

Building Reverse Engineering Tools with Software Components

by

Holger Michael Kienle

Dipl.-Inf., University of Stuttgart, 1999

M.Sc., University of Massachusetts at Dartmouth, 1996

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

© Holger Michael Kienle, 2006

University of Victoria

All rights reserved. This dissertation may not be reproduced in whole or in part, by
photocopying or other means, without the permission of the author.

Building Reverse Engineering Tools with Software Components

by

Holger Michael Kienle

Dipl.-Inf., University of Stuttgart, 1999

M.Sc., University of Massachusetts at Dartmouth, 1996

Supervisory Committee

Dr. Hausi A. Müller, (Department of Computer Science)

Supervisor

Dr. R. Nigel Horspool, (Department of Computer Science)

Departmental Member

Dr. Yvonne Coady, (Department of Computer Science)

Departmental Member

Dr. Issa Traoré, (Department of Electrical & Computer Engineering)

Outside Member

Dr. Dennis B. Smith, (Carnegie Mellon Software Engineering Institute)

External Examiner

Supervisory Committee

Dr. Hausi A. Müller, (Department of Computer Science)

Supervisor

Dr. R. Nigel Horspool, (Department of Computer Science)

Departmental Member

Dr. Yvonne Coady, (Department of Computer Science)

Departmental Member

Dr. Issa Traoré, (Department of Electrical & Computer Engineering)

Outside Member

Dr. Dennis B. Smith, (Carnegie Mellon Software Engineering Institute)

External Examiner

Abstract

This dissertation explores a new approach to construct tools in the domain of reverse engineering. The approach uses already available software components—such as off-the-shelf components and integrated development environments—as building blocks, combining and customizing them programmatically to realize the desired functional and non-functional requirements. This approach can be characterized as component-based tool-building, as opposed to traditional tool-building, which typically develops most of the tool’s functionalities from scratch.

The dissertation focuses on research tools that are constructed in a university or research lab (and then possibly evaluated in an industrial setting). Often the motivation to build a research tool is a proof-of-concept implementation. Tool-building is a necessary part of research—but it is a costly one. Traditional approaches to tool building have resulted in tools that have a high degree of custom code and exhibit little reuse. This approach offers the most flexibility, but can be costly and can result in highly idiosyncratic tools that are difficult to use. To compensate for the drawbacks of building tools from scratch, researchers have started to reuse existing functionality, leading towards an approach that leverages components as building blocks. However, this emerging approach is pursued in an ad hoc manner reminiscent of craftsmanship rather than professional engineering.

The goal of this dissertation is to advance the current state of component-based tool-building towards a more disciplined, predictable approach. To achieve this goal, the dissertation first summarizes and evaluates relevant tool-building experiences and case studies, and then distills these into practical advice in the form of lessons learned, and a process

framework for tool builders to follow.

The dissertation uniquely combines two areas, reverse engineering and software components. The former addresses the constructed tool's application domain, the latter forms the foundation of the tool-building approach. Since this dissertation mostly focuses on tools for reverse engineering, a thorough understanding of this application domain is necessary to elicit its requirements. This is accomplished with an in-depth literature survey, which synthesizes five major requirements. The elicited requirements are used as a yardstick for the evaluation of component-based tools and the proposed process framework. There are diverse kinds of software components that can be leveraged for component-based tool building. However, not all of these components are suitable for the proposed tool-building approach. To characterize the kinds of applicable components, the dissertation introduces a taxonomy to classify components. The taxonomy also makes it possible to reason about characteristics of components and how these characteristics affect the construction of tools.

This dissertation introduces a catalog of components that are applicable for the proposed tool-building approach in the reverse engineering domain. Furthermore, it provides a detailed account of several case studies that pursue component-based tool-building. Six of these case studies represent the author's own tool-building experiences. They have been performed over a period of five years within the Adoption-Centric Reverse Engineering project at the University of Victoria. These case studies, along with relevant experiences reported by other researchers, constitute a body of valuable tool-building knowledge. This knowledge base provides the foundation for this dissertation's two most important contributions. First, it distills the various experiences—the author's as well as others—into ten lessons learned. The lessons cover important requirements for tools as uncovered by the literature survey. Addressing these requirements promises to result in better tools that are more likely to meet the needs of tool users. Second, the dissertation proposes a suitable process framework for component-based tool development that can be instantiated by tool builders. The process framework encodes desirable properties of a process for tool-building, while providing the necessary flexibility to account for the variations of individual tool-building projects.

Contents	v
----------	---

Contents

Supervisory Committee	ii
------------------------------	-----------

Abstract	iii
-----------------	------------

Contents	v
-----------------	----------

List of Tables	ix
-----------------------	-----------

List of Figures	x
------------------------	----------

Acknowledgments	xii
------------------------	------------

Dedication	xiii
-------------------	-------------

1 Introduction	1
-----------------------	----------

1.1 Motivation	1
1.2 Problem	3
1.3 Approach	4
1.4 Validation	5
1.5 Outline	5

2 Reverse Engineering	7
------------------------------	----------

2.1 Background	7
2.2 Process	10
2.3 Tools	12
2.3.1 Repository	15
2.3.2 Extractors	16
2.3.3 Analyzers	18
2.3.4 Visualizers	20
2.3.5 Tool Architecture	22
2.4 Summary	26

3 Requirements for Reverse Engineering	27
---	-----------

3.1 Requirements	27
3.2 Requirements for Tools	29
3.2.1 Scalability	31
3.2.2 Interoperability	38
3.2.3 Customizability	46
3.2.4 Usability	52
3.2.5 Adoptability	58
3.2.6 Other Quality Attributes	68
3.2.7 Functional Requirements	71

3.2.8	Discussion	80
3.2.9	Contributions	81
3.3	Requirements for Tool Development Processes	82
3.3.1	Feedback-Based	84
3.3.2	Iterative	86
3.3.3	Prototype-Based	88
3.3.4	Other Requirements	89
3.3.5	Discussion	91
3.4	Research Approach to Identify Requirements	92
3.5	Summary	96
4	Software Components	97
4.1	Background	97
4.1.1	Component Definitions	98
4.1.2	Component Types	99
4.1.3	Software Reuse	106
4.1.4	Component Markets	107
4.1.5	Product Lines	108
4.1.6	Component-Based Software Engineering	109
4.2	Component Taxonomy	113
4.2.1	Origin	114
4.2.2	Distribution Form	114
4.2.3	Customization Mechanisms	115
4.2.4	Interoperability Mechanisms	118
4.2.5	Packaging	121
4.2.6	Number of Components	122
4.2.7	Other Taxonomies	125
4.2.8	Discussion	126
4.3	Summary	127
5	Building Reverse Engineering Tools with Components	129
5.1	Characteristics of Targeted Components	130
5.2	Catalog of Targeted Components	137
5.2.1	Visualizer Host Components	138
5.2.2	Extractor Host Components	150
5.3	Sample Tool-Building Experiences	156
5.3.1	Visual Design Editor	157
5.3.2	Desert	162
5.3.3	Galileo	166
5.4	Discussion	172
5.5	Summary	174

6	Own Tool-Building Experiences	175
6.1	Rigi as Reference Tool	177
6.2	REOffice	178
6.2.1	Excel and PowerPoint	179
6.2.2	REOffice Case Study	181
6.2.3	Conclusions	183
6.3	SVG Graph Editor	184
6.3.1	SVG for Reverse Engineering	184
6.3.2	SVG Integration in Rigi	187
6.3.3	SVG Experiences	190
6.3.4	Conclusions and Future Work	192
6.4	REVisio	192
6.4.1	Microsoft Visio	193
6.4.2	REVisio Case Study	194
6.4.3	Related Work	198
6.4.4	Conclusion	200
6.5	RENotes	200
6.5.1	Background	201
6.5.2	Rigi	204
6.5.3	Lotus Notes/Domino	204
6.5.4	RENotes Case Study	206
6.5.5	Conclusions	209
6.6	REGoLive	211
6.6.1	Related Research Tools	212
6.6.2	Adobe GoLive as a Host Product	213
6.6.3	REGoLive Case Study	216
6.6.4	Discussion of Tool Requirements	222
6.6.5	Conclusions	224
6.7	WSAD Web Site Extractor	225
6.7.1	Background	225
6.7.2	Fact Extraction for Reverse Engineering	227
6.7.3	Leveraged Functionalities	231
6.7.4	Case Studies	235
6.7.5	Experiences	238
6.7.6	Conclusions	239
6.8	Summary	239
7	Recommendations and Lessons Learned	241
7.1	Tool Perspective	241
7.1.1	Scalability	242
7.1.2	Interoperability	244

Contents	viii
7.1.3 Customizability	245
7.1.4 Usability	248
7.1.5 Adoptability	250
7.1.6 Discussion	251
7.2 Process Perspective	253
7.2.1 Process Framework	254
7.2.2 Work Product Template	257
7.2.3 Intended Development Process	258
7.2.4 Functional Requirements	262
7.2.5 Non-functional Requirements	263
7.2.6 Candidate Host Components	265
7.2.7 Host Product Customization	267
7.2.8 User Interface Prototype	269
7.2.9 Technical Prototype	271
7.2.10 Tool Architecture	273
7.2.11 Discussion	276
7.3 Summary	280
8 Conclusions	281
8.1 Contributions	282
8.2 Future Work	284
8.2.1 Reverse-Engineering Requirements	284
8.2.2 Tool-Building Approach	287
8.3 Parting Thoughts	288
Bibliography	289
Acronyms	343
Colophon	345
Creative Commons License	346

List of Tables

1	The five steps of EBSE	93
2	Reuse of Meta-Environment components	100
3	Summary of Sametinge’s component taxonomy	125
4	Summary of Morisio and Tarchiano’s framework	126
5	Characteristics of visualizers vs. extractors	138
6	Examples of visualizer host components for tool-building	139
7	Examples of extractor host components for tool-building	151
8	Characteristics of the discussed tool-building experiences	157
9	Summary of own tool-building experiences	175
10	Wong’s reverse engineering requirements	185
11	Fact extractor matrix	229
12	Summary of lessons learned and supporting case studies	252
13	Process characterization framework	279
14	Possible evaluation approaches for tool quality attributes	286

List of Figures

1	Dependencies of this dissertation's chapters	6
2	Reverse engineering activities	8
3	Reengineering in relation to forward and reverse engineering	11
4	Components of reverse engineering tools	14
5	A conceptual tool architecture	23
6	SHriMP visualization tool	25
7	Changes to TkSee resulting from usability evaluation	56
8	Storey's iterative tool-building process	87
9	Unix filters wrapped as ActiveX components	102
10	Compound document rendered in Lotus Notes	104
11	Library dependencies of xine	105
12	CBSE development cycle	110
13	Component taxonomy at a glance	113
14	Proactive vs. reactive parts of a component	119
15	Major themes of this dissertation	129
16	Tool-building target design space	137
17	A snap-together visualization	140
18	SHriMP Eclipse plug-in	142
19	Ephedra migration tool	145
20	ReWeb tool	147
21	BOX tool	148
22	Visual Design Editor	158
23	VDE dialog	159
24	VDE domain specification	159
25	VDE latency analysis	160
26	Desert editor	163
27	Desert's FOOD tool	164
28	Early version of Galileo tool	167
29	Galileo tool	168
30	Switching of active views in Galileo	168
31	Software structure graph in Rigi	178
32	Part of PowerPoint's object model	180
33	Software structure graph rendered in PowerPoint	181
34	RSF statistics computed in Excel and exported to PowerPoint	182
35	SVG graph	187
36	SVG document export in Rigi	188
37	SVG graph embedded in Web browser	191
38	REVisio showing a Rigi graph	195
39	REVisio showing Rigi pie charts	195

40	REVisio showing Visio's pan & zoom tool	196
41	REVisio tree layout of a small graph	197
42	REVisio radial layout of a large graph	198
43	REVisio CBO bar chart	199
44	Lotus Notes mail application	205
45	RENotes' layered architecture	207
46	Nodes in a sample RENotes database	208
47	Keyword search of a RENotes database	209
48	Visualization of a RENotes database	210
49	GoLive Design view	214
50	GoLive Links view	215
51	REGoLive Client view graph	218
52	REGoLive Developer view graph	219
53	Querying the WSAD link repository	232
54	EMF Web model	234
55	Creating an EMF model instance	235
56	Broken links in the ACSE home page	236
57	Reverse engineering of the ACSE home page	237
58	Relationships of development phases	255
59	A tool architecture in UML	274
60	Relationships of the process framework's work products	276
61	Goal model for software evolution	285

Acknowledgments

“So here you are now, ready to attack the first lines of the first page.”

– Italo Calvino [Cal81]

This dissertation happens to be rather lengthy. To compensate for this, I will keep the acknowledgments short.

Thanks to Hausi for getting me through all this. The fact that this dissertation came into existence—quite unexpectedly to me—testifies to his outstanding abilities as an advisor. Thanks to the committee members for taking the time to read and comment on the dissertation. Thanks to Crina for partial proofreading—and much more.

“Dem Handeln mit Computern liegt in direktem Sinne kein menschliches Urteilen, sondern nur ein Entscheiden zwischen einem mathematischen 0 oder 1 zu Grunde – zumindest, wenn man sich blind auf die Entscheidungen einer autonomen Maschine verlässt.”

– Joseph Weizenbaum, preface in [Kas03]

(A dedication has been intentionally omitted.)

1 Introduction

“Programs these days are like any other assemblage—films, language, music, art, architecture, writing, academic papers even—a careful collection of preexisting and new components.”

– Biddle, Martin, and Noble [BMN04]

1.1 Motivation

Tools and tool building often play an important role in applied computer science research. Tools are a fundamental part of software engineering research in general, and reverse engineering research in particular. The tangible results of research projects are often embodied in tools, for instance, as a reference or proof-of-concept implementation. For the research community,¹ the tool building experiences can be as valuable as the tool itself. Communicated experiences are a form of knowledge transfer among researchers in the community that can help, for instance, to point out potential pitfalls such as unsuitable technologies, or hard-to-meet requirements. Thus, studying and improving of tool building in the areas of software and reverse engineering promises to be most beneficial to the researchers involved.

Importance of tools
for research

Shaw has analyzed software engineering research methods and found several popular techniques that researchers use to convincingly validate their results [Sha01]. Among those techniques is the implementation of a (prototype) system. The character of this validation is: “Here is a prototype of a system that . . . exists in code or other concrete form.” A tool prototype serves, for example, to prove the feasibility of a certain concept, or as the basis for user studies to further enhance the tool. An analysis of the software engineering literature of six leading research journals found that 17.1% of the publications employ proof-of-concept implementations as a research method, placing it second only after conceptual analysis with 43.5% [GVR02]. Thus, tool building is a necessary, pervasive, and resource-intensive activity within the software engineering community. Nierstrasz et al., who have developed the well-known Moose tool, say that

Tools are used to
validate research

“in the end, the research process is not about building tools, but about exploring ideas. In the context of reengineering research, however, one must build tools to explore ideas. Crafting a tool requires engineering expertise and effort, which consumes valuable research resources” [NDG05].

Even though tool building is a popular technique to validate research, it is neither simple nor cheap to accomplish. Tool building is costly, requiring significant resources. This is especially the case if the tool has to be robust enough to be used in (industrial) user studies. Nierstrasz et al. identify one particular difficulty when building tools for the reengineering domain:

Tools are costly

¹A community is a group of people who share common goals, knowledge, and believes [Ves97].

“Common wisdom states that one should only invest enough in a research prototype to achieve the research results that one seeks. Although this tactic generally holds, it fails in the reengineering domain where a *common infrastructure* is needed to even begin to carry out certain kinds of research” [NDG05].

Often a tool is developed throughout the whole duration of a Master’s or Ph.D. thesis. Sometimes a significant part of the resources of an entire research group are devoted to building, evaluating, and improving a tool.

An example of such a significant academic tool-building effort is the Rigi reverse engineering environment, which has been under active development and then support for almost two decades [Mül86] [MK88] [MTO⁺92] [TWMS93] [Won96] [Mar99] [MM01a]. It has been used in several industrial-strength case studies [WTMS95] [Riv00a] [MW03]. Rigi had several major implementations.² The initial implementation was conducted on the Apple Macintosh as part of a dissertation (1984–1986). A subsequent version was based on SunView (1986–1988). Rigi was then ported to OpenLook, Motif, and Windows. Now, Rigi runs on various platforms and leverages Tcl to provide a scripting layer to automate recurring reverse engineering tasks. The GUI is implemented with Tk. The initial Tcl/Tk implementation involved at least three Ph.D. students, three research associates, and several Master’s students. To keep the tool appealing, new and emerging operating systems had to be supported as well as upgrades caused by new Tcl/Tk versions needed to be accommodated. As part of the evolution of Rigi, JavaScript is now offered as an alternative to Tcl scripting to lower Rigi’s entry barrier for new users [Liu04]. Supporting Rigi in a university research environment is difficult because of the relatively short stay of Master’s students. Furthermore, the tool is now no longer the focus of the main research efforts of the group, giving students little incentive to learn the tool and to understand its C/C++ implementation.

Rigi tool-building
effort

Tools developed in research often have a high degree of custom code—most parts of a tool are developed from scratch. This approach to tool building offers the most flexibility, but has a number of potential drawbacks. For example, building everything from scratch can be costly and can result in highly idiosyncratic tools that are difficult to understand by the user, but also difficult to maintain. If tools are built from scratch, a large part of the construction effort has to be devoted to peripheral code. For example, to effectively visualize and work with a reverse engineering graph, the user needs functionality for scrolling, zooming, loading/saving, cut/copy/paste, printing, and so on. Thus, only a small part of such a tool’s functionality is devoted to the actual research, such as a novel graph layout or clustering algorithm.

Tool building from
scratch

The drawbacks of building tools from scratch are slowly causing a shift in the way tools are now built by researchers. More and more tools are built using components. Components offer a different way of tool building, which replaces hand-crafted code with “pre-packaged” functionality. With components, the emphasis shifts from “tool-crafting” to “tool-assembling,” meaning that existing, pre-packaged functionality in the form of com-

Tool building with
components

²Personal communication with Hausi Müller in April 2005.

ponents needs to be assembled and customized rather than written from scratch. Finding appropriate host components that already provide a baseline infrastructure, and being able to customize them via adding functionality on top of them is critical for succeeding in the development effort. Tool-assembling promises many benefits, but requires different skills and methods, compared to traditional tool-building. In this dissertation, I take a broad view of what constitutes software components, defining them as “building blocks from which different software systems can be composed” [CE00]. Thus, a component can be a commercial off-the-shelf product, an integrated development environment, an object-oriented framework, or a dynamically loaded library.

Whereas there are examples of the use of components in the reverse engineering domain,³ in other areas of computer science the building of tools with components has advanced to a level where it is practiced routinely. An example of such a domain is compiler construction research, which has developed components for many compiler functionalities (e.g., scanning, parsing, symbol table management, and code generation). Many of these components are black-box and generative in nature [ACD94]. Using components for the construction of a compiler can greatly reduce the development cost and subsequent maintenance efforts; for instance, contrast writing a parser by specifying a grammar versus designing and implementing the whole parser from scratch. Generally, the use of components and the ideas of reuse are more pronounced in the *mature domains* [BCK98, p. 376] of computer science such as database management systems and compilers, which have progressed to the state of *professional engineering* (as opposed to mere *craftsmanship*) [Sha90].

Mature use of
components

1.2 Problem

Components promise to benefit tool building. As a consequence, reverse engineering research tools have begun to make the transition to components. However, the transition is currently made by researchers in an ad hoc fashion, driven, for example, by the desire to increase tool-building productivity, tool quality, or tool adoption. Often, this transition is made subconsciously, disregarding that the use of components represents a shift in the way tool building should be approached. The use of components fundamentally changes the development of tools and has unique benefits and drawbacks. However, this often is not realized by tool builders. Furthermore, the use of components in tool building is currently ad hoc, because of a lack of known experiences and guidelines. An example of emerging research in this area is Sommerville’s *software construction by configuration* [Som05a] [Som05b].

Impact of
components

It is my belief—which I have further confirmed with a literature search—that many software engineering researchers tend not to report their tool-building experiences in scientific publications. Indeed, Wirth has observed in 1995 that “there is a lack of published case studies in software construction” [Wir95].⁴ One reason for the lack of experiences might

Lack of tool-building
experiences

³A domain can be defined as an area of knowledge or activity [CE00, sec. 2.7.1].

⁴Notable exceptions that I am aware of originate from mature domains such as compiler construction and operating systems. Wirth has published a book about compiler construction that discusses the Modula-2 source

be the perception that they are not part of the publishable research results. For instance, the published research results for the Rigi environment do not contain any tool-building experiences, even though a significant body of research has been generated based on these tools (e.g., [TMWW93] [Ti194] [Ti195] [WTMS95] [SWM97]). I believe, however, that these experiences are valuable, could greatly benefit other researchers, and constitute a first step towards a better and more formal understanding of tool-building issues. A published body of experiences could, for example, lead to a better understanding of how different tool-building approaches affect the resulting tools' quality attributes and other properties; and help to distill guidelines, patterns, and methods for tool building.

There is a lack of understanding for both traditional and component-based tool-building. However, since tool building is costly and often a crucial part of research activities, a better understanding of tool building with components is an important contribution to the reverse engineering area in particular, and to the software engineering area in general. This dissertation aims to help tool builders by exploring the use and impact of components on tool building. The dissertation focuses on reverse engineering because the author has significant experience in this area and has participated in several tool-building projects and case studies. Equally important, reverse engineering is an area whose community places great importance on the building and evaluation of tools. Thus, advancing the state-of-the-art in tool building promises to have a significant impact on future research in this area.

Dissertation's aim

1.3 Approach

How can tool building with components be made more predictable and understandable? This dissertation cannot hope to give a comprehensive answer, but offers practical advice for reverse engineering researchers who want to use components in their tool-building efforts. More specifically, this dissertation

Research approach

- surveys the functional and non-functional requirements of reverse engineering tools and the tool-building process.
- discusses feasibility and implications of using components for building reverse engineering tools.
- proposes an approach for tool building with components, consisting of lessons learned and recommendations.

I bootstrap my results by drawing from a significant number of concrete tool-building experiences and investigations of tools—both my own as well as others.

Reverse engineering encompasses a broad spectrum of tools. As a consequence, reverse code of a complete compiler implementing a toy language [Wir86]. In their book, Wirth and Gutknecht concisely describe the design and implementation of the Oberon system (i.e., operating system and compiler) with many code excerpts from the actual system [WG92]. Tanenbaum has implemented a Unix clone called MINIX with the goal to provide students with a “working, well-documented operating system” [Tan87]; MINIX is documented in a book of about 900 pages including the complete C sources [TW97].

Scope of investigated tools

engineering tools and their functionalities and requirements are quite diverse. However, they typically all have a similar architecture (cf. Figure 4 in Section 2.3), including (1) a component that extracts information from certain sources (e.g., high-level programming language code or machine code), and (2) another component that visualizes information derived from these sources in some textual or graphical form. For example, the Rigi tool follows this architecture, having programming language fact extractors, and a graph visualization engine. Following this observation, the two main foci in this dissertation are fact extractors and (graph) visualizers.

1.4 Validation

As mentioned above, my results are derived from concrete tool-building projects and case studies. In fact, one contribution of this dissertation is a comprehensive analysis of existing work, with the goal of presenting it in the new context of tool building with components.

I draw from diverse personal tool-building experiences involving a number of different components. These experiences have been mainly conducted within the Adoption-Centric Reverse Engineering (ACRE) project⁵ of the Rigi group at the University of Victoria [MWW03] [MSW⁺02]. In this dissertation, the following case studies of tool construction are discussed:

Own case studies

- Graph visualization with Microsoft PowerPoint and Excel [WYM01] [Yan03]
- Graph visualization with Scalable Vector Graphics [KWM02]
- Metrics visualization with Microsoft Visio [ZCK⁺03] [Che06]
- Reverse engineering environment with Lotus Notes [MKK⁺03] [Ma04]
- Web site reverse engineering tool with Adobe GoLive [GKM05b] [GKM05a] [GKM05c] [Gui05]
- Web site fact extractor with IBM Websphere Application Developer [KM06]

The obtained tool-building experiences from the above case studies provide the necessary practical background to validate my results, and to distill recommendations and lessons learned.

1.5 Outline

The next chapter sets the stage by introducing the reverse engineering domain in some detail. Chapter 3 then discusses the requirements of reverse engineering tools. Chapter 4 gives background on software components and introduces a taxonomy for describing and comparing them. Chapter 5 explains my proposal of component-based tool-building for the reverse engineering domain. I first identify components with suitable characteristics

⁵<http://www.acse.cs.uvic.ca>

for my tool-building approach, and then describe concrete component-based tool-building examples. Thus, this chapter draws from background on both reverse engineering (cf. Chapter 2) and software components (cf. Chapter 4). Chapter 6 describes my own tool-building case studies.

Chapter 7 distills the related work of others and my own tool-building experiences into a number of lessons learned. The experiences that I gained from conducting my own tool-building case studies was instrumental to shape and understand the unique aspects of this tool-building approach. It also provides the means to judge the viability of the proposed tool-building approach by assessing the constructed tools with the tool requirements established in Chapter 3. Chapter 7 provides recommendation on how my proposed way of tool building can be accomplished in the form of a process framework, which can be instantiated by researchers to suit the particular needs of their tool-building project. Chapter 8 concludes the dissertation with a summary of its contributions, and proposed future work.

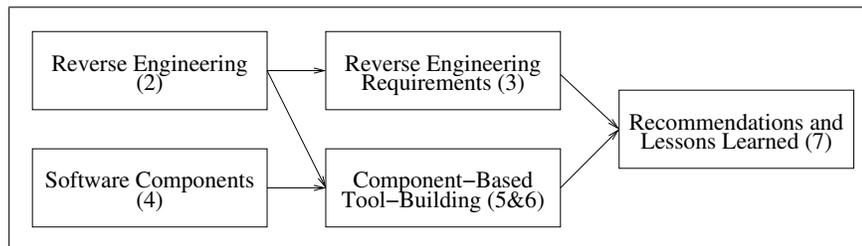


Figure 1: Dependencies of this dissertation's chapters

Figure 1 visualizes the dependencies of the chapters of this dissertation. Chapters 2 and 4 have been written so as to be mostly independent from each other, providing self-contained introductions to reverse engineering and software components, respectively. The case studies of component-based tool-building that are discussed in Chapters 5 and 6 require background in both reverse engineering and software components. Chapter 7 uses the identified requirements for the reverse engineering domain to guide the recommendations and lessons learned that are distilled from the case studies.

Dissertation's
dependencies

2 Reverse Engineering

“We all engage in reverse-engineering when we face an interesting new gadget. In rummaging through an antique store, we may find a contraption that is inscrutable until we figure out what it was designed to do. When we realize that it is an olive-pitter, we suddenly understand that the metal ring is designed to hold the olive, and the lever lowers an X-shaped blade through one end, pushing the pit out through the other end. The shapes and arrangements of the springs, hinges, blades, levers, and rings all make sense in a satisfying rush on insight. We even understand why canned olives have an X-shaped incision at one end.”

– Steven Pinker [Pin97, p. 21f]

Our tool-building domain is reverse engineering. Consequently, this chapter provides the necessary background on reverse engineering to better understand its process, tools, requirements, and research community.

2.1 Background

A broad definition of reverse engineering is the process of extracting know-how or knowledge from a human-made artifact [SS02]. An alternative definition is provided by the U.S. Supreme court, who defines it as “a fair and honest means of starting with the known product and working backwards to divine the process which aided in its development or manufacture” [BL98]. This process typically starts with lower levels of abstraction to create higher levels of understanding. Reverse engineering has a long-standing tradition in many areas, ranging from traditional manufacturing to information-based industries.

General reverse
engineering

In the software domain, Chikofsky and Cross define reverse engineering as follows [CC90, p. 15]:

Software reverse
engineering

“Reverse engineering is the process of analyzing a subject system to

- identify the system’s components and their interrelationships and
- create representations of the system in another form or at a higher level of abstraction.”

Research in reverse engineering is about tools and techniques to facilitate this process. The definition of Chikofsky and Cross is generally accepted by researchers in the field because it is flexible enough to encompass the broad range of reverse engineering activities, describing it as a process of discovery without explicitly stating its inputs and outputs [Sim03]. Inputs can range from source code (e.g., Cobol, Java, or VHDL) to videos of design meetings; outputs can be (annotated) source code, metric numbers, or architectural diagrams.

Reverse engineering of software falls into two distinct groups: *Binary reverse engineering* (or low-level reverse engineering) recovers source code from software that is only

Binary vs. high-level
reverse engineering

available in binary form,⁶ whereas *high-level reverse engineering* is typically concerned with the analysis of source code with the objective of recovering its design and architecture [Cif99]. This dissertation addresses high-level reverse engineering because most researchers work in this area. Consequently, the vast majority of publications in reverse engineering conferences and journals deal with high-level reverse engineering of software systems.

Reverse engineering is performed for a broad variety of reasons ranging from gaining a better understanding of parts of a program, over fixing a bug, to collecting data as input for making informed management decisions. Depending on the reasons, reverse engineering of a software system can involve a broad spectrum of different tasks. Examples of such tasks are program analyses (e.g., slicing), plan recognition, concept assignment, redocumentation, clustering, database or user interface migration, objectification, architecture recovery, metrics gathering, and business rule extraction [Til98a] [Rug96].

Reverse engineering
tasks

data gathering	system examination	static
		dynamic
		mixed
	document scanning	
experience capture		
knowledge management	organization	
	discovery	
	evolution	
information exploration	navigation	selection
		editing
		traversal
	analysis	types
		levels
		automation
	presentation	multiple views
		visualization techniques
		user interface

Figure 2: Reverse engineering activities ([Til00])

To accomplish a particular task, a reverse engineer identifies and manipulates *artifacts*. Tilley classifies artifacts in three categories [Til00] [Til98a]:

Reverse engineering
artifacts

data: Data (or facts) are the factual information used as the basis for study, reasoning, or discussion.

knowledge: Knowledge is the sum of what is known, which includes data and information such as relationships and rules progressively derived from the data.

⁶Software in binary form encompasses object code, bytecode format, binary code, executable code, and so on.

information: Information is contextually and selectively communicated knowledge.

Each type of artifact has a corresponding *canonical activity*: data gathering, knowledge management, and information exploration (cf. Figure 2). The latter activity is arguably the most important, because it contributes most to program comprehension; it encompasses information navigation, analysis, and presentation.

One important benefit of reverse engineering is that it can aid engineers to comprehend their software systems better. Program comprehension can be defined as “the process of acquiring knowledge about a computer program” [Rug96].⁷ Biggerstaff et al. state that

Program
comprehension

“a person understands a program when they are able to explain the program, its structure, its behavior, its effects on its operational context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program” [BMW93].

The activity of exploring and trying to understand a program has been likened to archaeology [RGH05] [HT02] [WCK99], detective work [KC99] [Cor89], and urban exploration of an unknown city or building [Moo02a]. Besides reverse engineering, Tilley identifies two other support mechanisms that aid program comprehension: unaided browsing, and corporate knowledge and experience [Ti198a]. With unaided browsing, a software system is explored manually (online or offline) without tool support. Corporate knowledge and experience, which is kept in the heads of the developers, is useful for program comprehension if available. It can be preserved through informal interviews with developers and mentoring.

Programmers use different comprehension strategies such as *bottom-up* [Shn80] (i.e., starting from the code and then grouping parts of the program into higher-level abstractions), and *top-down* [Bro77] [Bro83] (i.e., starting with hypotheses driven by knowledge of the program’s domain and then mapping them down to the code). There are many (personal) theories about what characterizes a programmer. Knuth, for example, believes that a programmer’s profile is “mostly the ability to shift levels of abstraction, from low level to high level. To see something in the small and to see something in the large” [Knu96].

Comprehension
strategies

Some programmers try to understand a program systematically in order to gain a global understanding, while others take an as-needed approach, restricting understanding to the parts related to a certain task [BGSS92] [ES98]. The latter approach has been also called just-in-time comprehension [SLVA97] [LA97] and just-in-time understanding [MJS⁺00]; its concept is nicely illustrated by Holt’s *law of maximal ignorance*: “Don’t learn more than you need to get the job done” [Hol01].

Just-in-time reverse
engineering

Reverse engineering is closely related to software maintenance, which is the process of modifying a software system or component to correct faults; improve performance or other attributes; adapt to a changed environment; or add functionality or properties. In

Software
maintenance and
evolution

⁷I use a general definition of program comprehension because it covers a wide variety of approaches. Other definitions restrict program comprehension to certain tasks such as design recovery (e.g., [PRBH91]).

this dissertation, I use the terms software evolution and maintenance interchangeably.⁸ A certain level of program understanding is necessary before a maintenance activity can be performed. The activity of trying to understand a system requires a significant amount of time—often more than half of the time of a maintenance activity is spent on program understanding [Cor89] [Sta84] [Sne97]. Thus, approaches that help program understanding, such as reverse engineering tools and techniques, have the potential to drastically reduce software development costs.

2.2 Process

When conducting a reverse engineering activity, the reverse engineer has to follow a certain process or workflow. To illustrate this workflow, I describe how a reverse engineer might obtain a system's *call graph*, which represents calls between program entities (e.g., procedures or files) [MNGL98] [MNL96].

Reverse engineering
process

The typical reverse engineering workflow can be structured into three main activities: extract, analyze, and visualize.⁹ These activities are conducted with the goal to synthesize and organize knowledge about the subject system. In the following, I briefly discuss the activities:

extract: A reverse engineering activity starts with extracting facts from a software system's sources.

Sources can be intrinsic artifacts that are necessary to compile and build the system (such as programming language source code, build scripts, and configuration files), or auxiliary artifacts (such as logs from configuration management systems, test scripts, user and system documentation, and videos of interviews with developers). For instance, the CLIME tool extracts facts from the following sources: Java code, Javadoc comments, UML class and sequence diagrams, JUnit test cases, and CVS logs [Rei05].

For a call graph, only facts from the source code itself need to be extracted. It is necessary to know the procedures (or methods) as well as the calls made within the procedures. Furthermore, the position of the artifacts within the source code (e.g., source file name, class name, and line number) is of interest.

analyze: Certain analyses are performed that use the facts generated in the extraction step.

Typically, analyses generate additional knowledge based on the extracted, raw facts. To obtain a call graph, the analysis has to match procedure calls with the correspond-

⁸There are currently no definitions of maintenance and evolution that are agreed on by all researchers in the field. Some view the terms interchangeably, while others have a more narrow view, applying evolution to only certain kinds of software-changing activities [CHK⁺01, p. 17].

⁹Moonen and Sim independently present a general software architecture or process of reverse engineering that is similar to the one proposed here, consisting of three phases: extraction, abstraction, and presentation [Moo02a, p. 11f] [Sim03, p. 100]. Reverse engineering is also described in terms of the *Extract-Abstract-View* [Kli03] [PFGJ02] [LSW01] or *Extract-Query-View* [vDM00] metaphor.

ing procedure definitions.¹⁰ With this information it is possible to construct a graph structure that represents the call graph.

Analyses can operate at the same level of abstraction as the original source, or provide some form of abstraction. For example, the CORUM II horseshoe model distinguishes between four levels of increasingly abstract software representations [KWC98] [WCK99]: source text, code-structure, function-level, and architectural. A call graph provides a basic function-level abstraction, omitting details of the actual implementation such as the employed programming language and the passed parameters. Call graph analysis can provide another form of abstraction by *lifting up* [WCK99] [Kri97] the call graph to the file level [KS03b].

visualize: Results of analyses are presented to the user in an appropriate form.

Generally, there are two main approaches to convey information to the reverse engineer: textual and graphical. In practice, often both approaches are combined.

A call graph, as any graph, can be expressed in textual form; however a graphical rendering with nodes (representing procedures) and arcs (representing procedure calls) is probably more intuitive and effective for the reverse engineer. If the call graph is presented with a graph visualizer such as Rigi, the reverse engineer can interactively explore the graph, for example, via filtering arcs and nodes, applying layout algorithms, and collapsing groups of nodes into compound nodes. Furthermore, it is possible to navigate from nodes and arcs to the corresponding source code that they represent.

It should be noted that the above workflow is idealized because it does not take into account the frequent iterations that a reverse engineer has to go through in order to obtain a meaningful result. For example, typically the extraction step produces facts that contain false positives or false negatives, which are detected at the analysis or visualization steps. This often forces the reverse engineer to go back and repeat the extraction to obtain a more meaningful fact base.

Process iterations

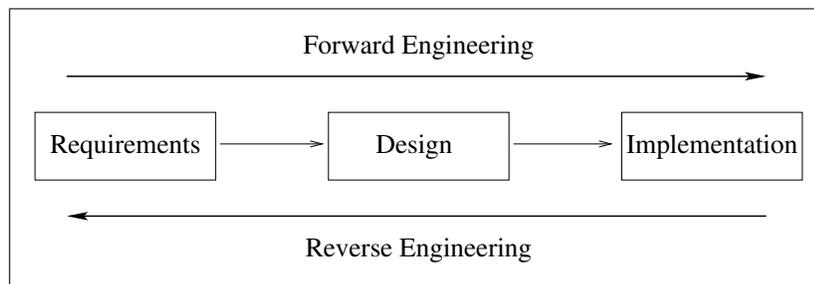


Figure 3: Reengineering in relation to forward and reverse engineering

¹⁰This matching is not necessarily trivial, for instance, for a call via a function pointer, or a dynamic method call.

The reverse engineering process is often embedded within the larger process of reengineering. Chikofsky and Cross state that “reengineering generally includes some form of reverse engineering (to achieve a more abstract description) followed by some form of forward engineering” [CC90, p. 15]. In contrast to reverse engineering, forward engineering changes the subject system. Figure 3 gives a graphical representation of the relationships. This dissertation is only concerned with the first step, reverse engineering.

Reengineering
process

2.3 Tools

Software systems that are targets for reverse engineering, such as legacy applications, are often large, with hundreds of thousands or even millions of lines of code. For example, the Rigi reverse engineering tool has been used on IBM’s SQL/DS database system [BH91], consisting of two million lines of PL/AS code¹¹ [WTMS95]. As a result, it is almost always highly desirable to automate reverse engineering activities.¹² This makes it possible to quickly reproduce steps in the workflow if the system changes. In principle, a reverse engineer could construct manually, say, a call graph without tool support via inspecting the entire source code.¹³ However, this activity is slow, error-prone, and needs to be repeated whenever the source code changes. Furthermore, sophisticated analyses can provide information about a system that is difficult or practically impossible to infer manually.

Necessity of tools to
automate

Consequently, the main focus of the reverse engineering research community is the construction of tools to assist the reverse engineer. Müller et al. state that

Tools aid program
comprehension

“many reverse engineering tools focus on extracting the structure of a legacy system with the goal of transferring this information into the minds of the software engineers” [MJS⁺00].

Thus, reverse engineering tools can facilitate program understanding. By analogy, program comprehension without reverse engineering tools is like urban exploration of cities without a navigation system.

Humans play an important role in reverse engineering. A significant part of the knowledge about a target system is either not considered (e.g., artifacts that are hard to extract such as documentation in natural language) or not accessible (e.g., the brains of developers) by reverse engineering tools. Thus, tools should allow input of human knowledge and make use of it. Sneed makes the following observations based on three reverse engineering projects, which were all based on automated tools:

Tools and human
knowledge

“It can be stated that automatic reverse engineering by means of an inverse transformation of code into specifications may have benefits to the user, espe-

¹¹PL/AS is a PL/1 dialect that stands for Programming Language/Advanced System. It is used within IBM.

¹²The authors of the Rigi tool state that “it took two days to semiautomatically create a decomposition using Rigi, but only minutes to automatically produce one via a prepared script. Either method would be much faster and use the analyst’s time and effort more effectively than would a manual process of reading program listings” [WTMS95, p. 53].

¹³In fact, programmers probably keep a (small) subset of the call graph in their memory when working with code.

cially when it comes to populating a CASE repository. However, there are also definite limitations. No matter how well the code is structured and commented, there will be missing and even contradictory information. Thus, it will always be necessary for a human reengineer to interact with the system in order to solve contradictions, to supply meaning and to provide missing information” [Sne95, p. 316f].

Thus, reverse engineers have to bridge the gap between knowledge provided by the tool and the actual knowledge required to solve a particular reverse engineering task.

Jahnke distinguishes tools in three groups according to how accommodating they are in considering human knowledge in the reverse engineering process [Jah99]: Human involvement

human-excluded: Human-excluded tools perform a fully-automated analysis (similar to batch-processing), producing static analysis reports without human intervention. This is typically the case for analyses that are related to traditional compiler optimizations such as the construction of call graphs or the inference of types (e.g., in COBOL [vDM98] [vDM00] and C [OJ97]).

human-aware: Human-aware tools perform automated analyses, whose results can then be (interactively) manipulated by humans. The Rigi environment is an example of a human-aware tool.

human-centered: Analyses of human-centered tools consider human knowledge up-front and during the entire reverse engineering process. An example of such an approach are Murphy and Notkin’s Reflection Models, which allows the reverse engineer to specify an architectural model of the target system up front and then to continuously refine it [MN97]. Baniassad and Murphy have developed a tool that allows a reverse engineer to propose a desired target structure of an existing software system in terms of conceptual modules [BM98]. The engineer can then perform queries to assess the impact of the proposed target structure. This leads to an iterative process of changing the conceptual models and assessing the impact with queries.

The amount of human involvement has to be balanced with the human effort that a reverse engineer is willing to expend to carry out an activity. Cremer et al. note that “if the legacy system is too large, interactive, human-centered re-engineering is too time-consuming and fault-prone to be efficient” [CMW02].

The reverse engineering community has developed many reverse engineering environments. Prominent examples of tools include Rigi, SHriMP, Moose [DLT00], Ciao/CIA [KCK99] [CFKW95] [CNR90], and Columbus [FBTG02] [FBG02]. Reasoning System’s Software Refinery is an example of a commercial tool that has influenced and enabled reverse engineering research (e.g., [BH91] [MGK⁺93] [MNB⁺94] [ABC⁺94] [WBM95] [YHC97] [FATM99] [NH00]). In Section 2.3.5, I briefly describe Rigi and SHriMP in more detail. Tool examples

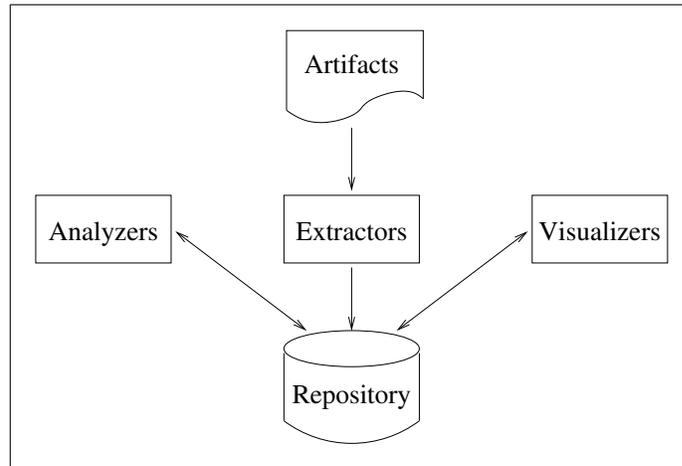


Figure 4: Components of reverse engineering tools

Most reverse engineering tools have a similar software architecture, consisting of several standard components. Figure 4 shows four types of components: extractors, analyzers, visualizers, and repositories; in the following, these are referred to as *tool component types*. The extractor, analyzer, and visualizer components reflect the reverse engineering workflow of extract, analyze, and visualize (cf. Section 2.2).

Tool component
types

An important commonality across all tool component types is that each utilizes a schema in some form. The purpose of a *schema* is to impose certain constraints on otherwise unrestricted (data) structures.¹⁴ An important design consideration for a schema is its *granularity*—it “has to be *detailed* enough to provide the information needed and *coarse grained* enough for comfortable handling” [Kam98]. Jin et al. [JCD02] distinguish schemas according to how they are defined (i.e., implicit vs. explicit), and where they are defined (i.e., internal vs. external). Implicit schemas are not explicitly documented, but rather implied by the data, or the way the data is interpreted. In contrast, explicit schemas are explicitly documented (e.g., via a formal specification). Internal schemas are hard-wired into components and thus are not required to participate in a data exchange, whereas external schemas are defined outside the components and thus need to participate in data exchange so that the components are able to interpret the data. Schemas are often discussed exclusively in the context of repositories. This is understandable because of the repository’s central role to facilitate data exchange. However, the remaining three component types also adhere to schemas, but this is often not recognized because these schemas are typically implicit and internal. For example, extractors, analyzers, and visualizers often use in-memory data structures (such as abstract syntax trees or control-flow graphs), whose schema is encoded as type definitions in the component’s source code.¹⁵ These

Schemas

¹⁴Schemas are also known as meta-models and domain models [Won99].

¹⁵An outstanding example of an extractor with an explicit schema is the Bauhaus reverse engineering tool, which uses a specification language called InterMediate Definition Language (IMDL) to define the schema of

components then have to transform their data from their internal representations in order to conform with the repository's schema.

In the following sections, I give an overview of the four tool component types.

2.3.1 Repository

The most central component is the repository. It gets populated with facts extracted from the target software system. Analyses read information from the repository and possibly augment it with further information. Information stored in the repository is presented to the user with visualizers.

Examples of concrete implementations of repositories range from simple text files to commercial databases. For instance, Rigi stores information in flat files. The ANAL/SoftSpec tool stores information in a relational database system (IBM DB2) [LSW01]. DISCOVER has a distributed database [Til97]. The RevEngE project uses an object-oriented database system (i.e., ObjectStore) as its persistent storage manager [MSW⁺94].

Repository types

Many reverse engineering tools store data in text files and define their own exchange format [Kie01] [Jin01]. Rigi defines the Rigi Standard Format (RSF) [Won98], which has been adopted by a number of other tools as well. RSF uses sequences of triples to encode graphs. A triple either represents an edge between two nodes or binds a value to a node's attribute. Furthermore, it is possible to assign types to nodes. Holt has developed the Tuple-Attribute language (TA) [Hol97], which is based on RSF. The GUPRO tool uses an XML-based exchange format called GraX [EKW99]. There has been a thrust in the reverse engineering community to define a standard exchange format, the Graph Exchange Language¹⁶ (GXL) [HWS00].

Exchange formats

Besides dedicated reverse engineering formats, there are general data exchange formats (e.g., XML, ATerms [vdBdJKO00], and GEL [Kam94]), and formats of general-purpose graph-drawing tools (e.g., dot, which is part of AT&T's graphviz package¹⁷ [NK94], Graphlet's GML [Him96],¹⁸ and EDGE's GRL [PT90]).

To extract information from a repository, there has to be a query mechanism. Querying is an important enabling technology for reverse engineering because queried information facilitates interrogation, browsing, and measurement [KW99]. For text files, general-purpose tools such as grep and awk can be used quite effectively [TW00] [WTMS95]. For database systems, its native query language can be used. For example, both the ANAL/SoftSpec and Dali tools use SQL to query their repositories [LSW01] [KC99]. The TA format has a query front-end (called Grok) that can, for example, compute the union of two relations or join two relations [Hol96]. Similarly, the GUPRO tool parses source into an graph structure, which is then queried via a domain-specific graph query language

Querying and APIs

its intermediate language IML [KM02a]. (An earlier version of Bauhaus had an implicit schema definition via hand-coded Ada data-structures—that is why Jin et al. classify IML as implicit rather than explicit [JCD02].)

¹⁶<http://www.gupro.de/GXL/>

¹⁷<http://www.graphviz.org/>

¹⁸<http://www.infosun.fmi.uni-passau.de/Graphlet/>

called GReQL [KW99] [LSW01]. The Jupiter source code repository system is based on an extended version of the MultiText structured text database system [CC01]. Jupiter provides a functional query language embedded in the Scheme programming language. XML-based data has the interesting property that it can be uniformly manipulated with XQuery¹⁹ and XPath²⁰. As an alternative to a stand-alone querying mechanism, reverse engineering tools also define application programming interfaces²¹ (APIs) to manipulate repository data [KCE00]. For example, Rigi transforms RSF into an in-memory graph structure, which can then be programmatically manipulated using Tcl scripts.

Data stored in a repository has to adhere to a certain schema. A typical example of a schema are the table definitions of a relational database; according to Jin et al.'s classification, database schema are explicit (e.g., documented in a file containing CREATE TABLE statements). In contrast, implicit schemas are often binary and similar to proprietary data formats (e.g., Bauhaus IML [CEK⁺00a]). Often the schema is explicitly documented with a (declarative) specification. This approach facilitates flexibility and extensibility. For instance, Rigi has separate files to hold the schema, while TA incorporates schema and data into the same file. XML files can have explicit, external schemas via an XML Schema²² or a Document Type Definition (DTD). JavaML is an example of an XML-based exchange format for reverse engineering of Java programs; its schema has been defined with a DTD [MK00].

Schemas

2.3.2 Extractors

Extractors populate the repository with facts about the subject software system. The extractor has to provide all the facts that are of interest to its clients (i.e., subsequent analyses).

Most research has focused on the extraction of facts from source code. Examples of other artifacts are software in binary form [CG95], database schemas and data [DA00], user interfaces [SEK⁺99] [AFMT95], reports [Sne04], and documentation [TWMS93] [ET94]. Source code is an example of an artifact that can be trusted because it represents the most complete and current information available. In contrast, documentation might be incomplete, outdated, or just plain wrong. Generally, the more formally defined the input, the more likely an extractor can be written for it (e.g., contrast a formal specification with one in natural language).

Extractor sources

Techniques used for data gathering from source code can be static (e.g., based on parsing) or dynamic (e.g., based on profiling). In the following, I focus on static techniques. Static extraction techniques use a number of different approaches [KL03]. Some extractors use compiler-technology to parse the source. This approach produces precise facts, but sources outside the parser's grammar cannot be processed. Many parsers are in fact based

Static extractor techniques

¹⁹<http://www.w3.org/XML/Query>

²⁰<http://www.w3.org/TR/xpath>

²¹The SEI defines an API as a technology that "facilitates exchanging messages or data between two or more different software applications. API is the virtual interface between two interworking software functions, such as a word processor and a spreadsheet" (qtd. in [dSRC⁺04]).

²²<http://www.w3.org/XML/Schema>

on compiler front-ends. For example, the C++ extractor CPPX is based on the GNU Compiler Collection (GCC) [DMH01], and Rigi's C++ parser is built on top of IBM's Visual Age compiler [Mar99]. However, there are also parsers that have been built from scratch such as the Columbus C++ parser [FBTG02]. Some tools such as A* [LR95], SCRUPLE [PP94], and tawk [AG06] [GAM96] provide query support to match parse trees or abstract syntax trees (ASTs). In contrast to parsing, there are lightweight approaches such as lexical extractors, which are based on pattern matching of regular expressions. Examples of this approach are LSME [MN96], MultiLex (which performs hierarchical matching via GNU `flex`) [CC00], SNiFF+²³ [AT98], TkSee/SN,²⁴ and Desert's so-called scanners [Rei99]. Lexical approaches are not precise, that is, they can produce fact bases with false positives (i.e., facts that do not exist in the source) and false negatives (i.e., facts that should have been extracted from the source) [MN96]. On the other hand, they are more flexible than parsers [Moo02b]. Typically, lexical extractors are language neutral and reverse engineers write ad hoc patterns to extract information required for a particular task. Island parsing [Moo01], similar to fuzzy parsing [Kop97], is an approach that combines both precise parsing and imprecise lexical matching.

Different extractor approaches have different properties. Sim distinguished between extractors that are sound vs. unsound, and full vs. partial [Sim03]. Generally, lexical approaches are unsound, whereas parsers are sound. Partial extractors such as LSME gather only selected information (e.g., which is of interest for a particular reverse engineering task). Parsers typically represent the complete parse and thus provide a full extraction. Lin et al. distinguish between four levels of extractor completeness [LHM03]. The highest level is source completeness, which makes it possible to reconstruct the exact source code (including comments and white spaces) from the fact base. At the lower end is a semantically complete extractor, whose fact base allows to reconstruct a program that behaves identically to the original source. All completeness levels imply a full extractor.

Extractor properties

Depending on the input, the development of an extractor can be a significant research effort. Obviously, different kinds of inputs require different extractors. But even for the same kind of input, different extractors might be required depending on the characteristics of the extractors themselves (e.g., speed vs. precision) and the desired facts (e.g., fine-grained vs. coarse-grained). Extraction problems for C and C++ have been reported repeatedly in the literature by researchers over the years. For example, at the SORTIE collaborative tool demonstration, three out of five teams had difficulties to parse the Borland C++ dialect of the subject system [SSW02]. Armstrong and Trudeau, who have evaluated several reverse engineering tools, observe that "in the extraction phase, several tools exhibit many parsing problems" [AT98]. Many of these difficulties are caused by irregularities in the source. Moonen identifies the following *irregularities* [Moo01]: syntax errors, completeness, dialects, embedded languages, grammar availability, customer-specific idioms, and preprocessing.²⁵

Extraction problems

²³http://www.windriver.com/products/development_tools/ide/sniff_plus/

²⁴<http://www.site.uottawa.ca:4333/dmm/>

²⁵Note that most of these problems are irregularities from the extractor's perspective because they are incom-

2.3.3 Analyzers

Analyzers query the repository and use the obtained facts to synthesize useful information for reverse engineering tasks. Reverse engineering research has developed a broad spectrum of automated analyses. Examples are analyses of static and dynamic dependencies; metrics; slicing; dicing; clone detection; cliché or plan recognition; clustering; and architecture recovery or reconstruction [MJS⁺00] [PRBH91]. Since there are far too many analyses to discuss, only a few representative analysis techniques are sketched in the following: program dependency information, clustering, and clone detection.

Many program analyses track dependencies between artifacts in source code. Generally, an artifact depends on another one if it directly or indirectly refers to it. For example, there is a dependency between a C function definition and its function prototype. Similarly, the use of a variable depends on the variable's definition. Many compiler analyses (such as definition-use chaining and alias computation) track dependencies at this level for subsequent code optimization [ASU86]. Other dependencies are function calls and file inclusions, which can be used to construct call and file-inclusion graphs, respectively. Not all dependencies are easy to identify. For example, type inference tries to infer types for weakly-typed languages such as COBOL, which lack explicit type information [vDM98] [vDM00].

Program
dependencies

A software system consists of interdependent artifacts such as procedures, global variables, types, and files. Clustering is an analysis technique that groups such artifacts into subsystems or (atomic) components according to certain criteria [Lak96] [GKS97]. Thus, clustering results provide one possible view of the system's architecture. Most clustering approaches group files, using dependency relationships (e.g., procedure calls), naming conventions, metrics, or measures of graph connectivity.

Clustering

Duplicated, or cloned, code in software systems is a common phenomenon.²⁶ Code clones are source code segments that are structurally or syntactically similar. A clone is often caused by copy-and-paste (-and-adapt) programming. Thus, one of the clones is usually a copy of the other, possibly with some small changes. Whereas clones are easily introduced during development, they can cause maintenance problems later on because (1) errors may have been duplicated along with the original code, and (2) a modification that is later made to one clone should be equally applied to all of the others. Besides assessing the amount of duplicated code in the system, clone detection is used for change tracking [Joh94] and as a starting point for (object-oriented) refactoring [BMD⁺00]. There is no universal definition of code clones; in fact, a clone is in the eye of the beholder. The precise definition of a clone depends on the actual clone analysis. Some analyses detect only textual clones (thus ignoring, for example, code segments that differ in their variable names, but are otherwise identical), others consider only whole procedures as clone candidates. The large number of clone detection approaches are interesting to study because they cover the entire design spectrum with respect to the types of facts, speed vs. precision, and visualization

Clone detection

patible with the extractor's assumptions about the source.

²⁶A duplication rate of ten percent is not uncommon [RDL04].

approaches.

Depending on the facts that analyses require, different categories of automated analyses can be distinguished: textual, lexical, syntactic, control flow, data flow, and semantic [CRW98] [Rug96]. For example, different approaches to detect software clones cover the entire spectrum of the above categories. At the textual level, clones can be discovered with simple string matching. Instead of looking for exact matches, white spaces and comments can be removed to catch slight variations (e.g., the level of indentation) in the code [DRD99]. This approach is simple to implement and mostly language independent. Lexical approaches can use regular expressions to group the source into tokens. For example, Johnson changes each identifier to an identifier marker to catch clones that have renamed variables [Joh94]. The syntactic level uses the structure of ASTs to identify clones [BYM⁺98]. At the next levels, control and data-flow information is considered. Mayrand et al. use metrics based on ASTs and simple control-flow information to establish similarities between C functions [MLM95]. Krinke uses a fine-grained program-dependency graph (PDG) to find clones via identification of similar subgraphs in a program's PDG [Kri01]. Identifying clones at the semantic level can be seen as looking for clichés or plans.

Types of facts

Jackson and Rinard present several dichotomies that help to further classify analyses [JR00]. In previous work, I have extended Jackson and Rinard's work [KM01]. The combined dichotomies are static vs. dynamic, sound vs. unsound, speed vs. precision, multi-threaded vs. single-threaded, distributed vs. localized, multi-language vs. single-language, whole-program vs. modular, robust vs. brittle, and fixed vs. flexible. In the following, four of the dichotomies are briefly discussed:

Analysis properties

static vs. dynamic Static analyses produce information that is valid for all possible executions, whereas dynamic analyses results are only valid for a specific program run. To assist the reverse engineer, both types of analyses are useful. Shimba is an example of a tool that combines static and dynamic analyses for the comprehension of Java programs [Sys00b] [SKM01]. Another example is dicing, which prunes a (static) program slice for a particular program execution [HH01].

sound vs. unsound Sound analyses produce results that are guaranteed to hold for all program runs. Unsound analyses make no such guarantees. Even though, such analyses are rather common in the reverse engineering domain. For example, many analyses ignore the effect of aliases,²⁷ thus yielding a potentially wrong or else incomplete result (e.g., for the construction of call graphs [MRR02] [ACT99] [TAFM97]). In fact, in an empirical study of C call graph extractors, five tools, which extracted information based on parsing the source code—a supposedly sound method—produced a variety of different results [MNGL98]. Still, even if the result of an analysis is unsound, it may give valuable information to a reverse engineer. Kazman and Carrière propose the composition of multiple unsound analyses (e.g., with weighted voting) to

²⁷Aliases occur when more than one expression can be used to refer to the same memory location. For example, in the C language `*p` and `x` are aliases after the statement `p=&x` has been executed [MRR02].

achieve a higher accuracy compared to the result of any individual analysis [KC99].

speed vs. precision Static analyses typically trade speed for precision or vice versa. This trade-off is well exemplified by the field of points-to and alias analysis, which has produced various different analysis techniques [Hin01]. Speed is often a problem for analyses that rely on data-flow information. Clone detection based on textual and lexical approaches typically scales well even to large inputs [Bak95] [DRD99]. In order to achieve good performance for syntactic clone detection, Baxter et al. had to implement efficient sub-tree matching based on hash-values [BYM⁺98]. Lastly, clone analyses based on PDGs have reported significant performance problems [Kri01].

multi-language vs. single-language Many analyses are tailored towards a specific programming language (e.g., pointer analysis or clustering for C). However, for a (global) analysis to be truly useful in a heterogeneous environment, language boundaries must be crossed. In the context of the Year 2000 (Y2K) problem,²⁸ Jones claims that about 30 percent of U.S. software applications contain at least two languages (e.g., C and Assembler, or COBOL and SQL) [Jon98, p. 49f]. The GUPRO project presents an approach to program comprehension in multi-language systems [KWDE98].

2.3.4 Visualizers

For software engineers to make the most effective use of the information that is gathered by extractors and augmented by analyzers, the information has to be presented in a suitable form by software visualizers.²⁹ Since the complexity of software makes it difficult to visualize, it is important that tools choose suitable techniques.³⁰ Particularly, different kinds of visualizations are more or less effective, depending on the comprehension task [MNS01, sec. 6.5]. Some information needs to be suitably condensed to give a birds-eye view of the subject system (e.g., call dependencies between files or modules rather than procedures), other information has to be presented in the most suitable representation for human consumption (e.g. call dependencies shown in tabular reports or directed graphs).

Generally, one can distinguish between textual and graphical software visualization. Examples of textual output are metric reports [WBM95] [SYM00] and re-formatted source

Textual and graphical
visualizations

²⁸<http://en.wikipedia.org/wiki/Y2K>

²⁹Mili and Steiner define software visualization as “a representation of computer programs, associated documentation and data, that enhances, simplifies and clarifies the mental representation the software engineer has of the operation of a computer system” [MS02].

³⁰Brooks draws a bleak picture when he states that “in spite of progress in restricting and simplifying the structures of software, they remain inherently unvisualizable” [Bro95, p. 186]. He elaborates, “whether we diagram control flow, variable scope nesting, variable cross-references, data flow, hierarchical data structures, or whatever, we feel only one dimension of the intricately interlocked software elephant. If we superimpose all the diagrams generated by the many relevant views, it is difficult to extract any global overview” [Bro95, p. 195]. This is true, however, researchers have developed visualizations that are believed to be effective for particular reverse engineering tasks, possibly because they abstract, prune, and summarize information appropriately.

code (e.g., to show software clones [BYM⁺98], control-flow diagrams [Sne01], or program slices [Wei84]). Reverse engineering information is often presented using graphs (e.g., visualization of module dependencies [FHK⁺97]), or diagrams (e.g., visualization of a system's structure with UML class or sequence diagrams [Gue04] [TP03] [Meh02] [Egy02] [SKM01] [KSS⁺02]), or Seesoft-like visualization of line-oriented source code statistics [ESS92]). There are also approaches to software visualization that use three dimensions (3D) [LS02] [MFM03] [HF02] [Rei95a], virtual reality [MLMD01], and haptic information [BL02].

Many reverse engineering tools use graphs to convey information [Kos03]. The Rigi environment represents software structures as typed, directed graphs and provides a graph visualizer/editor to manipulate the graphs. Rigi graphs can have different colors (to distinguish node and arc types), but all nodes and arcs have the same shape. Polymetric views utilize height and width of nodes, and width of edges to convey additional (e.g., metric) information [LD03]. Rieger et al. represent clones as graphs (nodes represent fragments of source code; edges represent duplication between two fragments), using polymetric views to represent metrics information. For example, in one view the nodes' width is used to convey the amount of internally copied code in a file, and the edges' width is used to convey the number of lines of copied code [RDL04]. The SHrIMP visualization tool has a view that represents graphs with a nested layout [SFM99b]. Leaf nodes can be opened up to reveal, for example, a text editor (e.g., to view source) or an HTML viewer (e.g., to view Javadoc) [MSM01] [SBM01]. There are a number of other visualizers similar to Rigi that have been developed by reverse engineering researchers; examples are CodeCrawler [Lan03], G^{SEE} [Fav01], LSEdit [SHD05], VANISH [KC96], Hy+ [CM93], G+ [CMR92], ARMIN [OT03] [KOV03], and SoftVision [TMR02b] [TMR02a]. Besides these editors, there are also general-purpose graph layouters and editors. The EDGE graph editor is an early example [New88] [PT90]. AT&T's graphviz provides an interactive editor, `dotty`, which can be customized with a dedicated scripting language, `lefty` [GN00] [NK94]. Various reverse engineering environments make use of AT&T's graphviz (e.g., Ciao/CIA [CFKW95] [CNR90], ReWeb [RT01b] [RT00], the reflexion model tool [MNS01] [MN97], CANTO [AFL⁺97], and Huang et al. [Hua04] [HHT01]). The daVinci Presenter, which originated at the University of Bremen, is another example of an interactive graph visualizer [FW94]. It supports nested graphs and allows folding and unfolding of subgraphs.

Graph visualization

In practice, tools often combine different visualization approaches. For example, CodeNavigator's user interface consists of the following views [Bro91]: (1) "lists are used to show traditional cross-reference information," (2) "directed graphs are used to show flow relationships," (3) textual annotations, and (4) "source code is displayed in a syntax-sensitive browser." Similarly, the Ciao system uses two main types of visualizations [CFKW95]: textual database viewers (to provide a human-readable output of information stored in a relational repository), and graph viewers (to render a visual representation of the relational information). Rigi combines graphical visualization with textual reports (e.g., a list of a node's incoming and outgoing arcs) to provide information about the graphs at

Mixed visualizations

different levels of detail.

The visualizations developed for clone detection serves to illustrate the spectrum of approaches. Clone detection techniques tend to produce a large amount of data, making it difficult for the reverse engineer to prioritize and assess the problem as a whole effectively [RDL04]. Researchers have developed several approaches to report clones with different visualization techniques at different levels of detail. The result of clone detection can report the location of every single clone; can group related clones into clone pairs, classes, or families [RDL04]; or can give a single number that estimates how much code could be saved by suitably eliminating clones [Bak95]. Clones can be reported textually using file names and line numbers, or graphically with a scatter-plot [DRD99] or polymetric view [RDL04].

Clone detection
visualization

Approaches also differ on how the visualized information can be manipulated. For example, one can distinguish between different levels of modifiability:

Manipulation of
information

static information: Static information is read-only such as a textual report or a static picture of a graph or diagram. Viewers for such information can offer navigational aids such as hypermedia to navigate between pieces of information [TWMS93] [Bro91] [Rai97].

generation of views from static information: Even if the underlying static information cannot be modified, viewers can offer customizable views to selectively suppress and highlight information. Views can be created by applying filters or layouts. Rigi, for instance, has filters to suppress the rendering of certain nodes and arcs. Viewers can collapse textual information and replace it with, say, an icon [CHBM98].

enhancement of static information: Some approaches do not allow the user to change the underlying information, but to enhance it. For example, a viewer might allow the user to add annotations to read-only entities. Rigi allows the user to group nodes into a parent node, facilitation the construction of a superimposed hierarchical tree-structure on the static node structure.

modification of information: An editor can allow the user to change and manipulate the underlying information. Note that changes are typically constrained in some way in order to ensure the information's consistency. For example, a graph editor that allows the user to delete a node, typically also deletes the node's attached arcs in order to keep the graph meaningful.

2.3.5 Tool Architecture

Most reverse engineering tools support the workflow explained in Section 2.2: extract, analyze, and visualize. Furthermore, the tools' software architectures often reflect this workflow as well. Figure 4 shows a generic, high-level reverse engineering architecture, consisting of four components: extractors, analyzers, visualizers, and a repository. This architecture exposes the *conceptual, or logical, structure* of the software, in which the

Conceptual
architecture

components (or units) are abstractions of the systems' functional requirements and are related by the shares-data-with relation [BCK98].³¹ However, note that the shares-data-with relationship does not necessarily imply well-defined or well-structured *interfaces*³² between units.

Bass et al. state that “this view is useful for understanding the interactions between entities in the problem space and their variations.” And indeed, Section 2.3 of this dissertation has shown the spectrum, or variations, of each component and how they interact. One can also look at the introduced conceptual architecture as a *reference model* for reverse engineering tools [BCK98, p. 25]. A reference model emerges through increasing consensus of a research community and thus indicates a maturation of the research domain. The domain of compilers provides an example of a widely known reference model [SG96] with the following functional units [ASU86, p. 10]: lexical analyzer, syntax analyzer, semantic analyzer, intermediate code generator, code optimizer, code generator, symbol-table manager, and error handler. Reference models in reverse engineering are important because they provide a frame of reference to guide researchers in understanding and implementing tools in the reverse-engineering domain.

Why conceptual architectures?

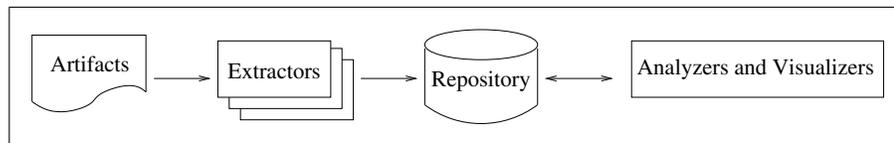


Figure 5: A conceptual tool architecture

The concrete architecture of a reverse engineering tool does not necessarily coincide with the conceptual tool architecture presented above. At one extreme, one could imagine a monolithic architecture that groups extraction, analysis, and visualization into a single component (without any interfaces). Many reverse engineering tools, especially the ones developed in the late 80s and early 90s, supported only a single programming language (e.g., MasterScope for Lisp, FAST for Fortran, and Cscope for C [CNR90]). Since these tools consisted of a single extractor, there was often a tight coupling between the extractor and the rest of the system [RPR93] [CNR90]. This rather tight coupling was often not intentional. Adding of a new extractor, which should have been a conceptually simple task, proved in practice to be quite difficult. This observation, led to the proposal of a clean separation between extraction and analysis by means of a language-independent, general schema that captures the semantics of multiple source languages [RPR93].³³ Whereas the

Concrete conceptual architectures

³¹There are many other architectural structures or views (e.g., Kruchten's 4+1 view model [Kru95]). In their book, Bass et al. identify the following additional views besides the conceptual, or logical, structure: module, process, physical, user, calls, and class structure [BCK98, sec. 2.5].

³²An interface is an “abstraction of a service that only defines the operations supported by that service (public accessible variables, procedures, or methods), but not their implementation” [Szy02, p. 556].

³³Note the similarity to the domain of compiler construction. UNCOL was an attempt to define a unified, executable intermediate representation for diverse programming languages [Mey97]. Vortex [DDG⁺96] is an

definition of such a general schema has proven to be difficult, most stand-alone reverse engineering tools now decouple the extraction from the analysis and visualization with a repository that supports a certain exchange format (cf. Figure 5). This decoupling has driven the reverse engineering community to work on standardized exchange formats—GXL is the most recent attempt—, so that facts can be shared among tools [HWS00].³⁴

Rigi is an example of a tool that decouples its extractors from the rest of the system via its exchange format, RSF. However, Rigi’s analyses and visualization are intertwined. Analyses (written in Tcl) are stored in separate text files, and use an API to access and manipulate Rigi’s internal, in-memory graph model. Analyses are invoked interactively from Rigi’s visualizer (e.g., via selecting an entry from a pull-down menu). Once an analyses is invoked, it can update the graph model; for instance, a clustering analysis can create new high-level nodes and group existing nodes into hierarchical structures. A changed state of the graph model is then immediately reflected by the graph visualizer. Consequently, analyses cannot be executed without running the visualizer itself. The integrated scripting support of Rigi allows lightweight customization as testified by a large number of reverse engineering applications that have been implemented on top of Rigi:

Rigi’s architecture

- Nimeta [Riv04, sec. 8.5] [RR02]
- jCosmo [vEM02]
- Shimba [SKM01] [SYM00]
- Bauhaus [Kos00] [CEK⁺00b] [Kos02]
- Dali [KC99] [O⁺B01]

Scripting also enables other tools to use Rigi as a server (or *engine*). RSF is a simple, yet powerful, text-based exchange format. It has been adopted by other tools besides Rigi. Many extractors have been written that are able to generate RSF in various domains (e.g., C, COBOL, PL/AS, C++, and Java).

SHriMP had several major implementations.³⁵ The first implementation was based on Rigi (with Tcl/Tk and C/C++, cf. Figure 6), followed by a Java port [WS00]. The Java version of SHriMP was then rearchitected in a component-based technology, JavaBeans, to facilitate closer collaboration between reverse engineering research groups [BSM02] [SBM01]. As a result, SHriMP can be used as a stand-alone tool, integrated within the Eclipse framework (Creole and Xia), or integrated within the Protégé knowledge-management tool (Jambalaya) [WMSL04] [LMSW03] [BSM02] [RLSB01] [MSM01]. SHriMP’s core architecture now consists of a number of JavaBean components [BSM02]

SHriMP’s
architecture

example of a compiler construction framework that defines an intermediate representation for several object-oriented languages (i.e., Modula-3, C++, and Cecil).

³⁴For the CASE domain, CDIF [Par92] and, more recently, XML Metadata Interchange (XMI) are standardized formats to exchange (meta-data) information between tools [TDD00b] [JS03].

³⁵Personal email communication with Casey Best of the Chisel group in October 2004.

server architectures (e.g., CodeNavigator [Bro91]), plug-in based architectures (e.g., Columbus), pipes-and-filters architectures (e.g., Ciao/CIA), frameworks (e.g., VizzAnalyzer [PLL04] [PS05]), and tiered Web applications (e.g., REportal [MSC⁺01], and the Software Bookshelf [FHK⁺97]).

2.4 Summary

In this chapter I introduced the domain of reverse engineering by motivating the use for it, explaining its role in program comprehension and maintenance, and describing a general process of how it can be accomplished. The architecture of reverse engineering tools was introduced along with a survey of its tool component types (i.e., repository, extractors, analyzers, and visualizers).

Whereas the reverse engineering domain exhibits commonalities with other computer science domains such as software engineering and compiler construction, it has its own unique requirements or rather emphasis on certain requirements. Domain knowledge is needed as a prerequisite to formulate the requirements of a domain. By providing a thorough description of reverse engineering, I have laid the necessary groundwork to understand the requirements of this domain, which is discussed in the next chapter.

3 Requirements for Reverse Engineering

“Tool research should not be entirely focused on new paradigms, but should address real user needs and expectations.”

– Kenny Wong [Won99]

This chapter discusses requirements of reverse engineering tools. Tool-building in the reverse engineering domain has its own characteristics, which may differ from other domains. For this reason, I explicitly give requirements that have been

- reported by other researchers in the reverse engineering area
- observed by myself while building tools for the ACRE project (cf. Section 1.4)

The identified requirements are useful to communicate assumptions about tool building in the reverse engineering domain. A requirement that has been reported (independently) by several sources is a good indication that reverse engineering tools should meet this requirement in order to be useful and fulfill the expectations of users.

This chapter has two main parts: I first discuss requirements of reverse engineering tools (Section 3.2), followed by requirements for a tool-building process for reverse engineering tools (Section 3.3).

3.1 Requirements

The tool architecture discussed in Section 2.3.5 describes the functionalities of a reverse engineering environment. This architectural view characterizes the system’s behavior,³⁶ capabilities, services, and purpose. However, besides a system’s functionalities, there are other, typically less palpable, system characteristics such as scalability, extensibility, and usability. In the following, I refer to the desirable characteristics of a reverse engineering tool or tool component type (i.e., repository, extractor, analyzer, or visualizer) as *requirements*.

Requirements
definition

Brown [Bro94], in the context of evaluating CASE environments, classifies requirements into

Requirements
classifications

- timeless (i.e., broad, context setting descriptions) such as “the environment shall be user friendly”
- technology-oriented (i.e., constraints on the architectural approach that is to be used) such as “the environment should have a central repository for all tool data”
- standards-oriented (i.e., definition of the major interfaces) such as the use of SQL for data management or CORBA for distributed messaging
- tool-oriented (i.e., mandating the use of particular components) such as “the environment shall use Microsoft Word as its document editor”

³⁶The behavior of a system can be expressed by specifying inputs and their expected outputs.

- user-oriented (i.e., description of typical usage via scenarios)

Another classification is the FURPS model [KS94] [Kru99, page 138f] used at Hewlett Packard, which stands for Functional requirements, Usability, Reliability, Performance, and Supportability. The benefit of these classifications is not so much that they allow a precise assignation of each requirement, but rather that they serve “as a template for requirements elicitation and for assessing the completeness of your understanding” [Kru99, page 139].

Even though I chose the term requirements, it is not meant to be interpreted in an overly restrictive sense. In the words of Carney and Wallnau [CW98], the term requirements

Requirements are preferences or desires

“has connotations that are often overly restrictive, and calling something a ‘requirement’ often implies a rather narrow understanding of notion of fitness for use. For example, sometimes the term is used in a legalistic sense: something is a requirement if and only if it is contained in a requirements document. The term often implies a degree of rigidity as well — in some situations the terms ‘preference’ and ‘desire’ might be more accurate than ‘requirement.’”

Furthermore, one cannot expect to meet all requirements equally well. In the words of Bass et al., “no quality can be maximized in a system without sacrificing some other quality or qualities” [BCK98, p. 75]. For example, Boehm makes the general observation that “added efficiency is often purchased at the price of portability, accuracy, understandability, and maintainability” [BKLW95, p. 13]. In the domain of compilers, there is a fundamental trade-off between compilation speed and quality of the generated code:

Dependencies of requirements

“Compiler speed can be achieved by reducing the number of modules and passes as much as possible, perhaps to the point of generating machine language directly in one pass. However, this approach may not produce a compiler that generates high quality target code, nor one that is particularly easy to maintain” [ASU86, p. 724].

Following this line of thought, a reverse engineering tool or development approach that is missing some of the requirements identified in Sections 3.2 and 3.3, respectively, can still be satisfactory.

Requirements that do not directly address the functionalities of a system are non-functional requirements, or *quality attributes* [BKLW95].³⁷ The quality of software can then be defined as the degree to which a system possesses desired combinations of quality attributes [BKLW95]. I mainly focus on quality attributes when discussing tool requirement because, as we will see, they equally apply to a broad range of reverse engineering tools or tool component types with varying functionalities. One can distinguish between

Quality attributes

³⁷I recognize that many quality attributes cannot be strictly seen in isolation because they are dependent on the behavioral aspects of the system (as explained in Clements’ side-bar *Non-functional Requirements Is a Dysfunctional Term* [BCK98, p. 76f]).

quality attributes that are observable via execution (e.g., performance, security, and usability) and quality attributes that can be reasoned about without execution (e.g., modifiability, portability, reusability, and testability).

Quality attributes often lack a precise definition. Examples of such attributes are modifiability and usability. A few quality attributes for particular, typically mature, domains have generally accepted definitions [BKLW95]. These definitions are often soundly grounded in mathematical formulae. For instance, one standard measure of system performance is latency (i.e., how long does it take to respond to a specific event?); the availability of a system is often measured as its mean time of failure; information retrieval measures the quality of query results via precision and recall; and so on.

Ambiguity of quality attributes

3.2 Requirements for Tools

Bass et al. state, “each domain has its own requirements for availability, modifiability, performance, security, or usability” [BKB02]. This is an important observation because it shows that the particular requirements of a certain domain can be better understood by starting from generic, domain-neutral requirements. Many domains share, say, a performance requirement. However, each domain instantiates this generic, high-level requirement differently, depending on the particular domain’s characteristics. Let’s consider portability as an example. The following domains instantiate this generic requirement quite differently:

Generic requirements

compiler construction: This domain traditionally places an emphasis on two types of portability: retargetability and rehostability [ASU86, p. 724]. A retargetable compiler is one that can be modified easily to generate code for a new target language. A rehostable compiler is one that can be moved easily to run on a new machine. The design and implementation strategies to achieve retargetability and rehostability differ significantly from each other.

operating systems: The first operating systems were tied to a particular machine and hence not portable. The Unix system showed that it was possible to implement an operating systems in a portable way using a high-level programming language, C,³⁸“making it easy to read, understand, change, and move to other machines” [Bac90, p. 4]. Thus, portability of an operating system is similar to rehostability of a compiler.

Web sites: A Web site is considered to be portable if it performs across platforms and is browser-safe (i.e., it looks and behaves the same on different Web browsers) [Ric03, sec. 2.4].

Thus, it is not sufficient to simply state the generic requirements for a domain without further elaborating on them—but that is what most reverse-engineering research does.

Besides generic requirements, domains have a number of unique requirements. These

Domain-specific requirements

³⁸The first versions of Unix were written in assembly language; by the Third Edition (1973) it was largely rewritten in C [Sal94].

requirements often reflect design and implementation knowledge of the domain. For example, programming language research has developed specific requirements (i.e., features deemed desirable) for a “good” programming language [Cez95, sec. 2.3.1]:

- simplicity (i.e., minimum number of rules to express algorithms for solutions)
- robustness (i.e., syntactic and semantic structures that are less error prone)
- rigor (i.e., exactitude through well-defined syntax, semantics, and pragmatics; complete, consistent, and free of ambiguities)
- abstraction (i.e., factoring out of recurring patterns)
- information hiding
- portability (i.e., machine independence allowing programs written on one computer to run on other systems)
- efficiency (i.e., speed of execution when run)
- and so on

In contrast, the domain of Web sites has the following requirements according to Powell et al.’s book *Web Site Engineering: Beyond Web Page Design* (discussed in [Ric03, sec. 2.4]):

- testable (i.e., the Web site is simple to test)
- maintainable (i.e., it is important that Web sites can be easily understood and extended)
- portable (i.e., the Web application performs across platforms and is browser-safe)
- reusable (i.e., it should be possible to reuse software modules or other work products of Web site development in more than one Web site)
- efficient (i.e., the response time for a requested Web page is short)
- well-documented (i.e., HTML and source code should contain comments, and the architecture and design of the Web site should be documented)
- and so on

Note that requirements can be contradictory (e.g., portability vs. efficiency), in which case reasonable trade-offs have to be found. An example of a requirement trade-off for reverse engineering analyses is speed vs. precision (cf. Section 2.3.3).

Whereas a reverse engineering tool’s functionalities is addressed by its conceptual architecture, it is the mapping of these functionalities to a concrete architecture and down to its implementation that determines the tool’s support for quality attributes. For example, the tool requirement to support multiple source languages may lead to a decoupling of the

Requirements drive
architecture

extractor components in reverse engineering tools, and a separation into front-ends and a common backend in compilers. Thus, the required quality attributes of a system often drive the decision to select a particular architecture or architectural style [BKB02] [SC97]. Conversely, the chosen architecture of a software system has a profound impact on its quality attributes [FB04] [SG96].

Several researchers have discussed requirements of tools in some detail. In his dissertation, Wong has distilled 23 high-level and 13 data requirements for software understanding and reverse engineering tools [Won99]. Tichelaar discusses six requirements for reengineering environments [Tic01, sec. 5.1]. Some researchers discuss requirements in the context of a certain type of tool. For instance, Hamou-Lhadj et al. discuss requirements for trace analysis tools [HLF04] [HL04], and van Deursen and Kuipers state requirements for document generators [vDK99]. Nørmark gives six requirements for elucidative programming [Nør00a]. Hainaut et al. analyze requirements for database reverse engineering environments [HEH⁺95] [HEH⁺96]. Other researchers address requirements of specific tool component types or functionalities (e.g., Koschke et al. give 14 requirements for intermediate representations in the context of reverse engineering [KGW98], and Ducasse and Tichelaar discuss requirements of exchange formats and schemas [DT03]). Tool requirements are also exposed by the criteria used in reverse engineering tool assessments [SK92], tool comparisons [BG97] [BG98] [AT98], and tool surveys [BK01] [Kos03].

Important related
work

In the remainder of this chapter, I discuss the requirements of reverse engineering tools in detail, based on a careful review of the relevant literature. To my knowledge, this is the first attempt of a comprehensive requirements survey in this domain. In the survey, where feasible, I identify whether a particular source has postulated the requirement for reverse engineering tools as a whole, or for a certain tool component type. However, unfortunately most researchers neither precisely define their postulated requirements, not explicitly state the applicable scope of them. In the latter case, I try to infer the scope out of the context. I also briefly discuss dependencies between different requirements when appropriate.

Survey of
requirements

3.2.1 Scalability

“Software developers frequently confront issues of bigness, aka scale. A harsh criticism of a solution to a software problem is the comment, ‘but it doesn’t scale.’”

– David West [Wes04]

As already mentioned in Section 2.3, reverse engineering tools might be used on subject systems of significant size. For example, one academic reverse engineering tool has been used on Microsoft Excel, which was reported to have 1.2 million lines of C code at the time [MN97].³⁹ A survey among users of software visualization tools has found that there is equal weight on the visualization of small, medium, and large target systems [BK01]. One third of the visualized systems were large (i.e., more than one million LOC), one

³⁹The size of a system is typically measured in lines of code (LOC). This measure is generally accepted even though it is not very precise.

third were medium (i.e., between one million and 100,000 LOC), and one third were small (i.e., less than 100,000 LOC). Bellay and Gall have evaluated four reverse engineering tools (Software Refinery, Imagix 4D, Rigi, and SNIFF+) using a 150,000 LOC embedded software system programmed in C and Assembler as a case study [BG98]. Even though this system is well below a million lines, they conclude that “many shortcomings of reverse engineering tools are due to the size of the case study system.” Baxter et al., discussing the requirements the Design Maintenance System (DMS) has to meet, make aware of the relationship between system size and processing time [BPM04]:

“A fundamental problem is the sheer size of large software systems. DMS is capable of reading literally tens of thousands of source files into a single session to enable analysis across entire systems. . . . Size translates into computational costs: 1 microsecond per source line means 2.5 seconds of CPU to do anything to 2.5 million lines.”

Whereas some tool implementations handle only limited input, serving often as a proof-of-concept prototype, realistic industrial-strength tools have to handle large target systems. Favre et al. state that “very often, a large number of unexpected problems are discovered when applying good tools at a large scale. This includes not only the size of the software but also the size of the company” [FES03].

Scalability as a general requirement for reverse engineering tools is discussed by a number of researchers:

General scalability
requirements

- Brown states for his CodeNavigator tool that it has been designed to “provide information about large-scale software systems” [Bro91]. Furthermore, addressing program understanding tools such as CodeNavigator in general, he observes: “to be effective, they must be able to handle systems of significant size” [Bro91, p. 367].
- In the context of legacy systems (which are typically large, highly complex, and mission-critical), Wong gives the following tool requirement: “Handle the scale, complexity, and diversity of large software systems” [Won99, Requirement 1].
- Based on their experiences with the Moose reverse engineering environment, Ducasse et al. state that “it should be possible to handle a legacy system of any size without incurring long latency times” [DLT00].
- Lethbridge and Anquetil have developed a list of key requirement for software exploration tools that support just-in-time comprehension. Among the requirements, they state that the system should “be able to automatically process a body of source code of very large size, i.e., consisting of at least several million lines of code” [LA97, Requirement NF1].
- ISVis is a visualization tool that supports the extraction of a subject system’s architecture, and provides guidance of where to insert an enhancement (called localization) [JR97]. Its authors state that “architectural extraction and localization are interesting

problems only if the system being analyzed is sufficiently large that an architectural overview is required to convey understanding. Consequently, it is important for extraction and localization technology to scale up to large systems.”

- In his dissertation, Tilley says “it is essential that any approach to reverse engineering be applicable to large systems. For example, effective approaches to program understanding must be applicable to huge, multi-million line software systems” [Til95, sec. 2.2.3].
- and so on [MMM⁺05] [BPM04] [LD03] [ALG⁺03] [KSSH03] [Lan03] [DT03] [FES03] [Bes02, sec. 3.4] [Tic01] [Til98a] [Sto98, p. 155] [Til97] [GA95] [Yan91].

Some researchers address scalability of specific reverse engineering tool component types. The granularity (or level of detail) of information about the system that needs to be extracted, stored, analyzed, and visualized is a major factor affecting scalability. A very detailed fact extraction (such as to accomplish *source completeness* [LHM03]) results in a larger database which in turn can decrease query performance. Detailed information is often extracted with the intent to perform sophisticated analyses (e.g., slicing or clone detection on PDGs), which take longer to perform and require more complex queries.

The amount of information that needs to be stored in the repository can affect scalability.⁴⁰ Cox and Clark say “a repository is scalable when there are no restricting limitations on the amount of extracted information or code that is stored” [CC01]. Tilley and Smith mention that the size and complexity of facts require scalable knowledge bases [TS96, sec. 3.2.3]. For example, a GXL file generated with the Columbus tool to represent the Mozilla 1.6 Web browser has a size of about 3.5 GB and contains about 4.5 million nodes [Fer05]. The Datrix representation of Vim 6.2 (220,000 LOC) has 3,008,664 relations and 1,852,326 entities [YDMA05]. Reverse engineering environments that use exchange formats as their repository should ensure that the format does “scale to large systems (e.g., 3 to 10 MLOC)” [BGH99]. Similarly, Wong requires for a scalable data format to “handle multi-million line software systems” [Won99, Data Requirement 9]. St-Denis et al. also list scalability among their requirements for model interchange formats (“can be used for real-life, e.g., large-scale applications”) [SSK00]. They discuss the following factors that may affect scalability: “the compressibility of the model interchange object, the possibility of exchanging partial or differential models and the possibility of using references (e.g., hyperlinks) to information instead of the information itself.” Another factor is incremental construction and loading of information to achieve resource optimization [DT03] [CNR90] [Kam98]. Jin et al. discuss scalability for schemas that are explicit vs. implicit and internal vs. external (cf. Section 2.3.1) [JCD02]. Hamou-Lhadj et al., discussing the requirements of trace exploration tools, state that input/output performance is critical since traces can become very large and users do not tolerate systems with a poor response time [HLF04]. They also note that “a general XML format, such as GXL, often requires more processing

Repository
scalability

⁴⁰This is especially the case for trace data of dynamic analyses, because a small program can, in principle, generate an infinite amount of trace data [JR97, sec. 5.1].

than a tuned special format. Performance of parsing XML poses an obstacle, especially for large data sets.”

Analogous to repository scalability, there is a scalability requirement for the in-memory data structures of reverse engineering components. Generally, it can be assumed that there is a linear correlation between the size of the target system and the information that needs to be stored [AG96].⁴¹ If the internal representation is fine-grained, it can consume a significant size. For example, the internal representation of Software Refinery’s C extractor is about 35 times the size of the source code [YHC97]. Atkinson and Griswold report that the AST of their analysis system consumes 414 MB and 18 million nodes for a target system with 40MB file space and one million lines of code [AG96]. With respect to the internal representation (IR) of an extractor or analyzer, Koschke et al. state that

- “the IR should be constructed efficiently. This property is a necessary condition for an overall efficient analysis. It is required to handle large systems in a reasonable time” [KGW98, Requirement (R4)]
- “traversals of the IR should be efficient; this is necessary because analyses usually imply traversing the IR at least once, in an iterative algorithm even many times” [KGW98, Requirement (R3)]

Ducasse and Tichelaar state that “the greater the level of detail, the higher the memory consumption and load time of information from databases or files and the slower the response times of tools that use the information” [DT03, p. 352].

One of the major activities in reverse engineering is querying [ZL00] [SL98] [SLVA97].⁴² In order to allow interactive manipulation of information, the repository and its query language have to be sufficiently responsive. Lethbridge and Anquetil state among their requirements for software exploration tools that they have to “respond to most queries without perceptible delay” [LA97, Requirement NF2]. Bull et al. say, “it is important for the tool not only to handle large amounts of data, but the tool must respond to queries on this data without perceptible delay” [BTMG02]. Meeting performance constraints can be difficult to achieve. For instance, an intelligent search algorithm for the TkSee tool has a considerable delay (“the average response time of the 34 samples queries was 37 seconds, which is relatively long compared with general user preferences of response time, i.e., 2 seconds”), which the authors classify as a major problem [LL01].

Query scalability

Since the extractor has to read in and process all sources of the target system, scalability is an important concern regardless, for example, of the granularity of the extracted information. As already discussed in Section 2.3.2, there is a broad spectrum of approaches with different performance trade-offs. For example, van Deursen and Kuipers observe that “lexical analysis is very efficient. This allows us to analyze large numbers of files in a short

Extractor scalability

⁴¹Koschke et al. make this an explicit requirement for intermediate representations (IRs): “The IR should be linear in size to the length of the source code. This property is particularly important for global analyses of large programs” [KGW98, Requirement (R5)].

⁴²I view searching as form of querying, albeit one that is typically less formal [SCHC99].

time, and also allows us to experiment with different lexical patterns: If a pattern does not yield the correct answer, the analysis can be easily changed and rerun” [vDK99]. Ferenc et al., discussing the Columbus tool, state “parsing speed” as a requirement for C++ extractors [FBTG02]. Bellay and Gall also give “parse speed” among their reverse engineering tool evaluation criteria [BG98]. They also give incremental parsing as a criterion, which “provides the ability to parse only the parts that changed, thereby reducing the parse time.” For performance reasons, extractors are often divided into scanner and parser [BPM04, sec. 4.3]. A related scalability concern is the question “how many models do you need to extract?” since each model might require a unique extractor [DT03].

Compared to the other tool component types, researchers have stated few general requirements for analyses. An explanation might be that the scalability of analyses is mainly dependent on the time complexity of particular algorithms. With respect to analyses for reverse engineering tools, Wong requires to “deliver diverse analyses” and to “support incremental analyses”, but he does not address quality attributes [Won99, tab. 6.1]. Koschke et al. state that “the IR should allow efficient control and data flow analysis” [KGW98, Requirement (R6)]. Generally, the more fine-grained the required facts, the more time-consuming the analysis. Flexible analyses can trade precision for scalability. For example, Fiutem et al. state for their Architecture Recovery Tool (ART): “To achieve scalability to large industrial size systems, special attention was also devoted to the speed of convergence of the iterative fixpoint method by conceiving a flexible analyzer that allows fine tuning of the trade-off between execution time performance and flow information precision” [FATM99, p. 359]. Researchers seem to discuss the scalability or performance only if there is a potential problem with the runtime of an analysis. For human-excluded analyses, there is a problem if the analysis cannot be executed in a nightly batch-run. In contrast, an analysis’ run-time of a human-aware tool has to be almost instantaneous. Nierstrasz et al. state for their tool that

Analyzer scalability

“one key technique that allows Moose to deal with large volumes of data is lazy evaluation combined with caching. For example, although Moose provides for an extensive set of metrics, not all of them are needed for all analyses. Even the relevant metrics are typically not needed for all entities in the system. Thus, instead of computing all metrics for all entities at once, our infrastructure allows for lazy computation of the metrics” [NDG05].

Moise and Wong discuss their experiences with the Rigi tool in an industrial case study. They have developed three specific analyses (written in Tcl) for clustering the subject system according to different criteria. They report for their analyses that “performance is becoming a serious issue with decomposition times running potentially into hours” [MW03]. Researchers have also discussed scalability and performance issues in the context of trace data compression [HLF04] [HL04] [HL02a].

A visualizer’s rendering speed has to scale up to large amounts of data. This is especially the case for interactive systems that allow direct manipulation of graphs or dia-

Visualizer scalability

grams.⁴³ For example, Czeranski et al. made the experience that “the Bauhaus GUI, which is based on Rigi, has a few unpleasant properties. It is relatively slow, which can cause noticeable waiting periods for large graphs, and hence sometimes disrupts the fluent use of the tool” [CEK⁺00b]. Armstrong and Trudeau state in their tool evaluation that “fast graph loading and drawing is essential to the usability of any visualization tool” [AT98]. Bellay and Gall use “speed of generation” of textual and graphical reports as an assessment criterion and experienced that “graphical views often need an unacceptable amount of time to be generated because of the layout algorithms” [BG98]. Contrary to Bellay and Gall’s findings, Moise and Wong note that their “case study showed that Rigi can deal with large graphs” [MW03]. In their survey, Bassil and Keller include “tool performance (execution speed of the tool)” as a practical aspect of software visualization tools [BK01, P5].

But besides rendering speed, there is another scalability issue:

“What’s the matter with the graphical interface today? The solution doesn’t scale. Making everything visible is great when you have only twenty things. When you have twenty thousand, it only adds to the confusion. Show everything at once, and the result is chaos. Don’t show everything, and then stuff gets lost” [Nor98].

The above quote from Norman on GUIs equally applies to the visualization of reverse engineering information [CMW02, p. 288] [Kos02, sec. 4] [MS02] [SFM99b] [ES98]. For example, hypertext navigation of reverse engineering information can suffer if the “information web” gets too large: “As the size of this web grows, the well-known ‘lost in hyperspace’ syndrome limits navigational efficiency” [TS96, sec. 3.3.1]. The authors of the REforDI reengineering environment say, “very soon we recognized that due to the limited perceptivity of human beings it is very important to be able to reduce and extend the amount of visualized information on demand” [CMW02]. Similarly, Reiss says that “practical program visualization must provide tools to select and display just the information of interest” [Rei95a]. To scale down the amount of visualized information, Storey recommends to “provide abstraction mechanisms” (e.g., which Rigi realizes via subsystem hierarchies) [Sto98, tab. 10.1]. Tilley states that “on very large systems, the issue of scale becomes very important. Visualization techniques are needed to sort and filter information” [Til97, p. 18]. Ferenc et al. also list filtering among the requirements for a reverse engineering framework [FBTG02]. The tool evaluations of Armstrong and Trudeau [AT98] as well as Bellay and Gall [BG98] have grouping of information among their evaluation criteria. Amyot et al. name “aggregation,” stating that it “permits the creation of groupings (and hence contributes to scalability in general)” [AMM03]. In fact, filtering and grouping/aggregating is among the standard repertoire of reverse engineering visualizers (e.g., Rigi [vEM02], Software Bookshelf [FHK⁺97], ISVis [JR97], Moose [DLT01], and

⁴³Storey et al., reporting on a user study with the SHriMP visualizer, found the following: “The single most important problem with SHriMP views was the slow response of the interface. Since SHriMP views are based on direct manipulation, users expecting immediacy were disturbed by the slow response” [SWF⁺96].

ManSART [YHC97]). Koschke explicitly identifies both types of scalability problems as major research challenges [Kos03]:

“Scalability and dealing with complexity are major challenges for software visualization for software maintenance, reverse engineering, and re-engineering. Scalability concerns both the space and time complexity of visualization techniques, as, for instance, automatic layouts for large graphs, as well as the need to avoid the information overload of the viewer.”

Besides performance, another related quality attribute is a tool’s robustness. Tilley and Smith state that “legacy software systems can contain millions of lines of source code. Because of this, support mechanisms need to be sufficiently robust to function effectively at this scale” [TS96, p. 23]. A scalable tool can also provide more flexibility for reverse engineers because efficient recomputation of information for a changed target system might be possible even if that system is large. Ducasse and Tichelaar explicitly identify several other quality attributes and features that scalability depends on: the structure of the tool’s exchange format, the level of detail at which the source code is modeled, incremental loading of information, and support for multiple models (because “the more models, the more information must be dealt with”) [DT03].

Dependency on other requirements

Most researchers discuss scalability in the context of computational performance and efficiency. An exception is visualization, where scalability is also discussed in terms of how to effectively display large amounts of data to the reverse engineer. Also, Favre et al. raise the issue of a tool’s scalability with respect to the number of (concurrent) users: “While a tool could perfectly suit the needs of a single user, its use by hundreds of software developers may unveil new issues” [FES03]. Scalability is typically discussed without giving explicit or quantitative metrics to measure them. Considering the large number of publications about reverse engineering tools, it is surprisingly rare to find quantitative statements about tools’ performance such as the following ones:

Discussion

- “CodeCrawler can visualize at this time ca. 1 million entities. Note that we keep all the entities in memory” [LD03].
- ISVis can analyze “program event traces numbering over 1,000,000 events” [JR97, p. 58].
- “The CharToon system consists of 46,000 LOC (without comments or empty lines) and 147 classes. The extraction step takes about 30 sec. on a computer with an AMD Athlon processor (1.2 Ghz) and 512 Mb main memory running Linux 2.4.9-12. The extracted source model contains 33,840 facts” [vEM02].
- “Mozilla compiled on our Linux system 4 in about 35 minutes, while the Acacia extraction took three and a half hours and the translation into TA took another three hours” [God01].

Furthermore, as the above examples suggest, each researcher reports measurements based on different criteria and metrics; even tool evaluations tend to not use objective measurements for comparisons among tools [AT98] [BG98]. There is an increased awareness in the software engineering community that approaches need to scale to industrial demands [FK00]. As a next step, the development of accepted criteria for the evaluation of tools such as benchmarks [SEH03] would be helpful to make performance and scalability measurements more meaningful.

3.2.2 Interoperability

“Building tools is expensive, in terms of both time and resources. An infrastructure for tool interoperability allows tool designers and users to re-purpose tools, which helps to amortize this cost.”

– Ferenc et al. [FSH⁺01]

In a small survey about negative aspects of reverse engineering tools, 6 software engineers out of 19 (31%) complained that tools are not integrated and/or are incompatible with each other [LS97] [LA97]—this was also the most frequent complaint!

Interoperability is the “ability of a collection of communicating entities to (a) share specified information and (b) operate on that information according to an agreed operational semantics” [LW04]. In other words, tools that interoperate enable them “to pass information and control among themselves” [Zel96].⁴⁴ Reverse engineering researchers have recognized the need for interoperability. Woods et al. [WOL⁺98] observe that

- “many tools have been written with closed architectures that hide useful internal products”
- “many external products are not produced in a useful form for further computation”

They further conclude that “the bottom-line is that existing program understanding tools and environments have not been designed to interoperate.” Other researchers have made similar observations [Bes02] [Per00] [EKW99] [Let98] [RW96]. The importance of interoperability is also recognized in other domains such as enterprise information integration [PR04] [SMC04], and portable intermediate representations in compiler construction such as ANDF⁴⁵ [Ope96].

Making tools interoperable yields a number of potential benefits. Interoperability enables code reuse in general because it becomes easier for a tool to utilize the functionalities of another one. As a result, reusing of existing functionality can “prevent repetitive ‘wheel-creation’” [KWC98]. Zelkowitz and Cuthill view interoperability as a way to improve automation in software engineering: “Tool integration ... is crucial to effectively provide automated support for activities. In order to automate activities with a tool set,

Benefits of
interoperability

⁴⁴A related concept of tool interoperability is *tool integration*, which is a measure of the extent to which tools are able to interoperate—or “agree” [TN92]. However, in practice the terms interoperability and integration are often used interchangeably.

⁴⁵<http://en.wikipedia.org/wiki/ANDF>

there must be a seamless way to pass information and control among the tools” [ZC97]. Furthermore, interoperability reduces the time and effort to (opportunistically) assemble a tool set that supports a particular reverse engineering task or process [Sto05] [Per00] [KWC98] [BGH99] [Man94]. For instance, the developers of the Dali tool say, that “one of our emphases has been to provide an open, lightweight environment so that tools can be integrated opportunistically” [KC99]. In an architecture reconstruction project, Gorton and Zhu have used five different reengineering tools. They say, “we discovered that the *tools complemented each other* in identifying reconstruction scope, critical architectural elements, potential design irregularities and creating useful architectural views for different evaluation tasks” (emphasis added) [GZ05]. Tool interoperability enables and promotes collaboration and community building among researchers; an example of such an effort is the *WCRE 2001 Collaborative Tool Demonstration* [SSW02] that involved six research groups to reverse engineer a legacy software system. Lastly, an interoperable tool can be used for purposes not envisioned by the original tool developers.

A number of researchers address interoperability as a general tool requirement:

General
interoperability
requirements

- Interoperability is among the tool requirements given by Lethbridge and Anquetil. They require from tools to “wherever possible, be able to interoperate with other software engineering tools” [LA97, Requirement NF4].
- Tichelaar, discussing requirements for reengineering environments, states that “a reengineering effort is typically a cooperation of a group of specialized tools. Therefore, a reengineering environment needs to be able to integrate with external tools, either by exchanging information or ideally by supporting runtime integration” [Tic01, sec. 5.1] [DLT01]. Similarly, Ducasse et al. say, “the environment should be able to operate with external tools like graph drawing tools, diagrammers (e.g., Rational Rose) and parsers” [DLT00].
- Wong addresses interoperability in the context of tool integration [Won99, p. 23]: “Tool integration is necessary to combine diverse techniques effectively to meet software understanding needs.”
- Interoperability is among the twelve requirements that Hainaut et al. identify for tools that aid in the reverse engineering of database systems: “A CARE tool must easily communicate with the other development tools (e.g., via integration hooks or communications with a common repository)” [HEH⁺95].
- The developers of the Columbus tool say that interoperability “is needed to connect the system with other tools. This can be done by furnishing a well defined interface (API), which allows access to the tool’s functionality” [FBTG02].
- Reengineering of SHriMP toward a component-based architecture was driven by the desire to simplify tool interoperability: “Before this reengineering effort, we were

unable to integrate SHriMP with other software engineering tools or use its visualizations for other purposes. This limited our ability to evaluate SHriMP’s effectiveness when used as part of a larger process” [Bes02].

- Knight and Munro describe their experiences in extending GraphTool to support GXL. For them “interoperability between tools means that independently specialized tools can be composed together to form sophisticated and powerful analysis suites for program comprehension, re-engineering, and maintenance” [KM02b].
- For their Dali architecture reconstruction tool, Kazman and Carrière say that “new elements must be easy to integrate into the workbench (openness)” [KC99].
- The designers of the Augur tool say, “we would like Augur to be broadly usable in real engineering practice. This means that it must be interoperable with a range of existing tools and infrastructures,” and further “Augur’s design emphasizes interoperability and extensibility so that it may be incorporated into existing development efforts without significant overhead” [FD04].
- and others [ADOV02] [ALG⁺03] [JR97] [BLM98] [Jin04] [Rei02] [FHK⁺97] [MMM⁺05] [DLT00].

For tools to interoperate, they have to agree on a suitable interoperability mechanism in some form or another. As a consequence, research papers often directly address the question of *how* to achieve interoperability, without explicitly stating it as a requirement first.

An early example of interoperability research is the Star approach to tool integration [Man94] [MHG94a] [MHG94b]. The Star system integrates tools with a set of translators. A separate reader/writer is written for each repository type (assumed to be text files). Readers transform repository information to a predefined Star in-memory representation; writers populate repositories with the information obtained from the Star in-memory representation. The Star system has a simple interactive user interface that allows users to select the source and target repository for translation. Thus, users drive the transformation and thus determine when information is exchanged between tools. This approach reduces the number of translators that need to be written because Star’s centralized in-memory representation allows to interconnect n different repository types with n reader/writer pairs; without this approach $n(n - 1)$ translators are necessary to directly connect each source to each target.⁴⁶ Star has been used in a case study to integrate various tools (e.g., text editor, graph viewer, interface viewer, and compiler) to realize a programming environment [MH95].

Star

Another approach that has been proposed to achieve interoperability among tools are the Common Object-based Reengineering Unified Model (CORUM) [WOL⁺98] and CORUM II [KWC98] [WCK99] frameworks. These proposals strive to provide a common

CORUM

⁴⁶This is an instance of the “ $n * m$ ” problem (with $n = m$) as it exists, for instance, in client-server interoperability (n clients want to connect to m servers) [Weg96], or compiler construction (support for n front-ends and m backends).

framework, or middleware architecture, for reverse engineering tools, encompassing standard APIs and schemas for ASTs, CFGs, call graphs, symbol tables, metrics information, execution traces, and so on.

A less ambitious approach compared to the CORUM frameworks is the community's efforts to define a common exchange format. Exchange formats enable interoperability between tools. A particular tool can utilize an exchange format to make information available to another tool. Panas et al. say "in order to have a properly working combination of two or more tools that have been developed independently, they must be able to exchange information. For this purpose we need a common data representation that is general enough to support a variety of data" [PLL04]. Exchange formats use simple files as a (temporary) repository for information transfer. The coupling between tools that exchange information is loose; an information producer need not to know its information consumers. Examples of exchange formats in the reverse engineering domain are RSF, TA, GraX, CDIF, and GXL (cf. Section 2.3.1); there are also a number of general-purpose data exchange and encoding formats such as XDR, ASN.1, SGML, and XML [Kie01] [SK01].⁴⁷ Effective interoperability of an exchange format can be accomplished if the format is an official or de facto standard. Bowman et al. state that the exchange format "should be a universal standard that is widely accepted" [BGH99, Criterion 7]. To get a format accepted among developers of diverse tools, it has to be *neutral*, that is "an exchange format representation that is independent of any particular tool, so that as many tools as possible can integrate with it" [JCD02]. The potential benefits of exchange formats are exemplified by the attendees of the *Workshop on Standard Exchange Format (WoSEF)*, who gave the following reasons for attending the workshop: they "wanted to be able to make complementary tools work together more smoothly" and "were tired of writing parsers/analyzers and wanted to avoid writing another one, in particular a C++ parser" [SK01]. Devanbu notes, "using a common data format across different tools saves time and storage space; the format needs to be built only once. For example, the parser need be run only once over the source code to build a persistent AST" [Dev99b]. For architecture reconstruction, Bowman et al. state, "lack of a standard exchange format does not prevent reuse between architecture-level frameworks, but it does make reuse difficult" [BGH99]. In Gorton and Zhu's reconstruction project, the RSF and GXL exchange formats were instrumental in achieving interoperability among the five tools [GZ05]. However, they were forced to write ad hoc scripts in Perl to translate between different tools' formats.

A file-based exchange format is a rather primitive form of data interoperability because there is no coordination mechanism for concurrent access. Thus, it is the responsibility of the tool user to assure data-consistency and to initiate data transfer from one tool to another. An example of a more sophisticated solution is a repository common to all tools (e.g., in the form of a database management system) such as proposed by the CORUM framework. Whereas a central repository has many benefits (e.g., data consistency), it

⁴⁷Before the advent of XML, researchers have also proposed to use HTML's meta tags to encode reverse engineering information [TS97].

is also a heavyweight solution. Brown cautions, “the common data repository is often a large, complex resource that must be controlled, managed, and maintained. This can occupy a great deal of time, money and effort” [Bro93]. Similarly, Wong concludes from the RevEngE project [MSW⁺94], which used the object-oriented Telos software repository [MBJK90], “there are significant difficulties in using and maintaining advanced integration technologies” [Won99, sec. 3.3]. He continues,

“advanced integration was difficult to implement and keep working due to tool and version changes. ... There are significant training costs that should not be overlooked in using advanced integration technologies and attempting tool integration. Integration technologies should be more transparent. The Rigi approach of using ‘traditional’ point-to-point integration via scripting seemed more efficient, flexible, and easier to understand” [Won99, sec. 6.3.1].

Exchange formats and common repositories are effective at communicating the structure (or syntax) of data. However, even if tools are able to read the data, it is of little use if they do not know how to interpret it. Schemas are a vehicle to convey semantic information about the data (i.e., its meaning and use). Interoperability among tools is much more effective if they agree on a certain schema [MW04b]. Godfrey puts it this way: “we feel that the particular syntax to define an exchange format is a small issue ... We consider the semantic model (design of the schemas) to be the most important issue” [God01]. A lesson learned by the developers of the Moose reengineering tool is to make the schema explicit; they state,

Schemas

“by making the [schema] explicit, we were able to develop tools that cooperate better, and so obtain benefits by combining results of multiple tools. Without this, each tool would have been a standalone prototype without any potential for supporting further experiments. By developing our own [schema] we were able to establish a minimal infrastructure for integrating [our] experimental tools” [NDG05].

Whereas syntax is domain-neutral, the schema models a particular domain or reflects an intended use. Thus, a single schema will not suffice. For example, there are schemas with different granularities to represent source code: fine-grained (e.g., a detailed C++ AST [FSH⁺01]), coarse-grained (e.g., the PBS high-level schema to model abstract architectures [BGH99]), and in between (e.g., the Dagstuhl Middle Model [LTP04]). Researchers try to establish standard schemas for other domains as well, for instance, execution traces: “There is also a need for a common [schema] for representing the execution traces of object-oriented systems in order to permit interoperability among tools” [HL04]. In practice, the diversity of tools makes it difficult to agree on schemas. Moise and Wong make the following observation:

“Often, an existing schema may not fit as-is to the particular software being analyzed or the tools being used. Consequently, schema reuse is not a simple task, and a proliferation of new schemas has resulted” [MW04b].

All schemas have in common that they have *weak semantics* [PR04], that is, meaning is derived from the names of schema entities and possibly informal documentation. For example, an entity called `linenumber` is presumably used on a source code fragment that exists at the given line number in a particular file. However, even if this assumption is correct, it is still not clear if line numbers are counted starting from zero or one, if the line number applies to raw or preprocessed source, if the line number denotes where the fragment begins or ends, et cetera.

Exchange formats specify the structure of the data and how it is stored on disk. However, how to actually read/write the exchange format and how to represent it in memory is not part of its specification [KCE00]. Thus, tools often implement their own readers and writers. These readers and writers have their own proprietary interface, reflecting the specific needs of a particular tool. Tools rarely can share an interface and its implementation. Interoperability can be achieved if tools agree on a standardized API to read, write, and manipulate data. A popular example of APIs that enable interoperability for relational data is Open Database Connectivity (ODBC) and Java Database Connectivity⁴⁸ (JDBC). For example, the G^{SEE} software exploration tool supports JDBC for data import [Fav01]. Another example of a standardized API in the compiler domain is the Ada Semantic Interface Specification (ASIS). ASIS exposes internal compilation information with a standardized interface.⁴⁹ Cooper states that this approach “promises to maximize interoperability between Ada CASE tools and Ada compilation environments” and to “promote standardization in software engineering environments, enabling data integration of Ada semantic information” [Coo97]. The Bauhaus reverse engineering tool uses ASIS to extract Ada source code facts for its intermediate representations: the RFG mid-level schema [Nei04], and the IML low-level schema [Mül05].

APIs

Dedicated frameworks for reverse engineering offer an infrastructure for tool integration via common data or control flows. Panas et al. have implemented an object-oriented tools framework in Java [PLL04]. Tools can implement certain interfaces to plug into the infrastructure. The CORUM frameworks are an effort to define common APIs for the reverse engineering domain. However, these efforts have not caught on in the reverse engineering community. Generally, the more sophisticated the interoperability mechanism, the more standardization and agreement among tools is necessary. Jin observes, “although the use of APIs significantly improves the speed and ease of interaction among tools, they still need to know how they can interact with each other. A tool must be aware of the requests it can make of another tool it interfaces with” [Jin04]. This form of tool interaction can be achieved by extending the API with message passing mechanisms based on a message bus (e.g., ToolBus [dJK03]), or point-to-point connections [Bro93].

One can attempt to draw an analogy between exchange formats (e.g., RSF) and APIs (e.g., ODBC) on how they achieve interoperability. Both abstract from the tools’ execution

⁴⁸<http://java.sun.com/products/jdbc/>

⁴⁹Note that the internal representation of two Ada compilers can differ widely, even if they both support ASIS. Thus, ASIS enables interoperability without unduly constraining the compiler’s implementation. Notably, the DIANA standard, who aimed for a standardization of the internal representation of Ada compilers, failed [Coo97].

environments. For example, RSF is stored in ASCII files, abstracting away different file systems. Similarly, ODBC abstracts from a concrete relational database. All data adheres to the same structure, or syntax: With RSF, data is represented in tuples, in ODBC data is represented in tables. The underlying model in RSF is a typed, directed, attributed graph; in ODBC it is relational algebra. Both approaches allow schema introspection: RSF has an (optional) schema description file; ODBC has catalog functions.

Interoperability can influence a number of other quality attributes. Interoperability has a beneficial impact on flexibility because components can be replaced more easily. For instance, a certain fact extractor can be replaced with another one that targets the same exchange format [FSH⁺01], or a JDBC-enabled tool can easily switch from one relational database system to another [Fav01]. An exchange format should have other qualities besides interoperability such as popularity, formality, and integrity [SSK00]. In particular, since an interoperable solution has to accommodate a perpetual stream of new tools with diverse requirements, it should be evolvable. According to St-Denis et al., an exchange format is evolvable if it “is easily adaptable to new requirements that may arise in the future” [SSK00]. Moise and Wong describe the following evolution scenario:

Dependencies of
requirements

“In reverse engineering a software system, one early step involves defining a schema and extracting the artifacts from the source code. These two tasks are mutually dependent and are iterative. As the source code is better understood, more facts are extracted, and the schema evolves to accommodate the new information. Often, an existing schema may not fit as-is to the particular software being analyzed or the tools being used” [MW04b].

Lastly, the solution should be lightweight and simple to realize because advanced tool integration technology can be brittle and difficult to maintain and learn [Won99, sec. 6.3].

I have tried to identify the most common and most widely discussed forms of interoperability. However, to focus the survey, it is not exhaustive. For example, a rarely used technique to achieve interoperability is through programmatic GUI manipulation. Bao and Horowitz’s approach uses a domain-specific scripting language to describe GUI events (e.g., mouse actions, window manipulations, and keyboard operations) to accomplish certain tasks [BH96].⁵⁰ Thus, a tool cannot tell the difference between human operation or manipulation through the scripting language. As a proof of concept, the authors demonstrate how a requirements elicitation tool called WinWin uses FrameMaker to render requirements documentation on-the-fly.

Discussion

Interoperability and integration of tools has been extensively addressed by researchers in the context of CASE tools and software development environments [Mey91] [BFW92]. Meyers identifies a number of requirements for tool integration: (1) it should be easy to write new tools for use in a development environment, (2) a new tool should be easy to add to the environment without the need to modify other existing tools that are already part of

⁵⁰This is accomplished by watching and manipulating communication events of tools that execute on the X Window System.

the environment, and (3) there should be no need for different tools to perform the same task (e.g., no more than one parser for a particular language) [Mey91]. He also discusses several approaches to system integration that have been already discussed here: shared file system (i.e., tools exchange data based on exchange formats), selective broadcasting (i.e., message passing among tools), simple databases (i.e., tools use a common database as repository), and canonical representations (i.e., tools share a common schema).

One approach pursued by the reverse engineering community to achieve interoperability is to agree on a common exchange format. This is exemplified by the thrust to establish GXL. There are also concrete proposals for schemas in a number of areas (e.g., C++ ASTs, mid-level architectural information, and trace extraction), but no proposal has achieved widespread use yet. The community has also considered more ambitious interoperability mechanisms such as a common repository and APIs. For instance, Sim acknowledges the usefulness of a standard exchange format, but argues to move towards a common API:

“Data interchange in the form of a standard exchange format (SEF) is only a first step towards tool interoperability. Inter-tool communication using files is slow and cumbersome; a better approach would be an application program interface, or API, that allowed tools to communicate with each other directly. . . . Such an API is a logical next step that builds on the current drive towards a SEF” [Sim00].

However, there is no indication that the reverse engineering community is devoting significant effort to realize this proposal. This is perhaps not surprising, considering that discussion on standard schemas has just begun. In a sense, agreement on an API is comparable to a simultaneous agreement of a standard exchange format along with a number of the most important schemas. Furthermore, exchange formats have their own benefits and it is not clear that APIs are necessarily a move in the right direction.

To my knowledge, there is another approach to interoperability that has not yet been considered by the reverse engineering community so far—service-oriented architectures (SOAs). SOA aims to integrate heterogeneous software systems with the use of middleware and services. Software systems expose their functionalities as services, communicating with each other via some kind of middleware [DG04]. Web services (based on XML, SOAP, WSDL, and UDDI) are a prime example of a SOA [LW04] [KSO02]. Reverse engineering tools could offer functionalities as Web services, and thus allow other tools to discover and call their services in a standardized form.⁵¹ Tool services would allow the development of new reverse engineering functionality via service composition, possibly even on demand [TBB03]. Existing tools could be reengineered towards a SOA (e.g., using SEI’s Service-Oriented Migration and Reuse Technique (SMART) [LMS06] [LMOS05]).

⁵¹Interestingly, researchers have explored the migration of legacy systems to offer Web interfaces and Web services [TGH⁺04]. However, researchers have not applied this approach to a certain class of (legacy) systems, namely their own tools!

3.2.3 Customizability

“It has been repeatedly shown that no matter how much designers and programmers try to anticipate and provide for users’ needs, the effort will always fall short.”

– Scott Tilley [Til95]

Customizability (or extensibility) is another important requirement for reverse engineering tools. Software Refinery is an early example of a customizable reengineering tool [NM93] [MNB⁺94]; for example, it allows customization of its parser, pretty printer, and GUI. Since most researchers do not distinguish between customizability and extensibility, I use both terms interchangeably. A concept related to customizability is *end-user programmability*, which allows the user of an application to tailor it programmatically to their needs [Til95, sec. 2.3].

As the introductory quote by Tilley suggests, reverse engineering activities are quite diverse and depend on many factors. As a result, reverse engineers have to continuously adapt their tools to meet changing needs. Thus, it is not sufficient for a reverse engineering tool to be general (i.e., it can be used without change in different contexts), it has to be flexible as well (i.e., it can be easily adapted to a new context) [Par79] [Par78]. In the context of data reverse engineering, Davis observes that “tools need [to be] customized to each project” [Dav01]. Similarly, Kazman and Carrière state in the context of architecture reconstruction, “because there is a great deal of variance ... we believe that no single collection of tools will suffice for all architectural extraction and analysis” [KC99]. Furthermore, customizability enables to meet needs that cannot be foreseen by tool developers, for instance, if a tool is applied in a new context or domain. Best states, “if the designer does not create an architecture that lends to extensibility, opportunities to use the tool in other domains can be missed” [Bes02]. Tilley characterizes such customizable tools as domain-retargetable [Til95, sec. 2.2.3]. Conversely, a tool that is not customizable is probably too rigid to meet the changing needs of reverse engineers except in a few well-understood circumstances. Markosian et al. say for reengineering tools, “in our experience, *lack of customizability* is the single most common limiting factor in using tools for software analysis and transformation” [MNB⁺94].

Many tool developers, including commercial ones that produce mass-market software, see customizability as an important requirement to satisfy their customers. The following researchers discuss extensibility as a general tool requirement:

- Buss and Henshaw discuss their experiences with the reverse engineering of IBM’s SQL/DS system. Among the lessons learned, they state that “any reverse engineering toolkit must be extensible to meet your problem domain needs,” and “since reverse engineering is an open-ended, context-dependent activity, it is imperative that its toolkits be similarly open-ended, flexible, extensible, and versatile” [BH91].
- In his dissertation about domain-retargetable reverse engineering, Tilley states, “a successful reverse engineering environment should provide a mechanism through

Benefits of
customizability

General extensibility
requirements

which users⁵² can extend the system’s functionality” [Til95, sec. 2.2.2]. Extensibility is also one of the quality attributes of Tilley’s Reverse-Engineering Environment Framework (REEF) [Til00] [Til98a].

- Bellay and Gall’s evaluation framework for reverse engineering tools contains a toolset extensibility criterion: “Tool extensibility is an important feature of many types of tools. This is also the case for reverse engineering tools, in which additional functionality often needs to be integrated to meet the specific constraints of a reverse engineering activity” [BG98]. The framework further distinguishes between extensibility of parsers, user interfaces, and tool functionality.
- Hainaut et al. give “functional extensibility” for CASE tools as a requirement, motivating it with “no CASE tool can satisfy the needs of all users in any possible situation” [HEH⁺95, sec. 10].
- Reiss has developed a tool, CLIME, to aid software maintenance by formulating constraints on development artifacts such as source code, UML design diagrams, comments, and test cases [Rei05]. Such a tool should be “adaptable to new design techniques and approaches” and as a result “must be open and extensible” [Rei02].
- In his dissertation on database migration, Jahnke states “customizability is a crucial requirement on CARE tools, because legacy databases differ with respect to many technical and non-technical parameters” [Jah99, p. 5].
- et cetera [LD03] [ALG⁺03] [DLT00] [Sto98, p. 154]. [FHK⁺97] [MNB⁺94] [Bro91] [Yan91] [BGV90]

Typically, tools enable customization of their functionalities via configuration files, built-in scripting support, or programmable interfaces. For instance, the Ciao tool leverages Unix pipes and command-line switches to achieve customizability. A particular visualization can be created by cascading and configuring individual filter programs [CFKW95]. The Rigi tool has a startup file to configure the used fonts, icons, text editor, et cetera. More advanced customizations are performed via Tcl/Tk scripting (cf. Section 2.3.5). The implementors of the Dali architecture reconstruction tool chose the Rigi tool for its extensibility via scripting:

Customization
mechanisms

“We needed to use a tool that provided both domain specific functionality—to do the necessary graph editing and manipulation—and extensibility, to integrate with the rest of the functionality of Dali. We are currently using Rigi for this purpose, since it provides both the ability to manipulate software models as graphs and extensibility via a control language based on TCL, called RCL (Rigi Command Language)” [KC99].

⁵²Note that in this scenario and the following ones the tools’ users are typically reverse engineers with considerable programming knowledge.

SoftVision is an open visualization toolkit whose functionality is very similar to Rigi's. The toolkit has a layered architecture consisting of a C++ core to improve performance and a Tcl/Tk layer [TMR02a]. The C++ core exposes an API to the scripting layer. Customization is accomplished either via the C++ API or Tcl scripting. The authors made the experience that “for most visualization scenarios imagined by our users, writing (or adapting) a few small Tcl scripts of under 50 lines was enough. This was definitely not the case with other reverse engineering systems we worked with” [TMR02a].

A repository consists of a schema and the data stored according to the schema. Each repository provides a rudimentary form of extensibility, because the data that is stored in the repository is not fixed, but customized by the applications that uses the repository. Thus, a more meaningful form of repository customizability is to look at the customizability of a repository's schema. In fact, customizability of a reverse engineering tool is often realized with an extensible schema. The developers of the Moose reengineering tool state, “the extensibility of Moose is inherent to the extensibility of its [schema]” [DLT01]. A number of researchers agree that an exchange format “should be extensible, allowing users to define new schemas for the facts stored in the format as needed” [Sim00] [BGH99, Criterion 8] [Mül98]. More specifically, Ducasse et al. require that “an environment for reverse engineering and reengineering should be extensible in many aspects: . . . the [schema] should be able to represent and manipulate entities other than the ones directly extracted from the source code (e.g., measurements, associations, relationships, etc.)” [DLT00]. According to Koschke et al., intermediate representations “should capture information provided by the user in addition to facts that are directly derivable from source code as needed by a compiler” [KGW98, Requirement R10]. Among the requirements that Ferenc et al. have identified for a C++ schema is the need for schemas to “be modular and easily extendible [sic]” [FBTG02]. Similarly, Riva states that an exchange format “should be easy to extend by the users themselves without the knowledge of complicated procedures” [Riv00b]. One of Wong's requirements for a reverse engineering repository is to “support dynamically evolvable schemas” [Won99, Data Requirement 1]. He further elaborates, “this flexibility to evolve schemas dynamically and incrementally is especially important in software understanding. New needs and concepts often arise over time.” The Software Bookshelf tool envisions a so-called builder, whose role is it to customize the tool according to the unique requirements of a particular migration project. An important requirement for the builder is a customizable repository “since the information needs [of users] . . . cannot all be foreseen, the builder requires powerful conceptual modeling and flexible information storage and access capabilities that are extensible enough to accommodate new and diverse types of content” [FHK⁺97, p. 567]. Event traces are an example of data obtained with a dynamic analysis. The ISVis tools supports the extension of trace data with new event types without having to change the tool itself: “As far as ISVis is concerned, events have types, and the exact nature of the type is unimportant to the pattern matching ISVis provides” [JR97].

Repository
customizability

One can distinguish between the following forms of schema extensibility:

fixed: Fixed schemas model a certain domain, which is not expected to change. For exam-

ple, the Bauhaus Resource Graph models the design level of procedural languages [CEK⁺00a]. A number of analyses and visualizations have been implemented based on this schema. Consequently, changes in the schema are expected to cause changes in the Bauhaus tool.

ad hoc: This approach allows to add information in an unstructured way, typically in the form of annotations or extensions (which can range from free comments in natural language to formal assertions) [KGW98, Requirement R10]. For example, software engineering environments allow tool-specific decorations of abstract syntax trees [Mey91], and UML allows to attach string tags to entities. Tools are then expected to ignore annotations that they do not know or expect. On the one hand, this approach allows to add new types of information without changing the schema. On the other hand, the new information is unstructured and cannot be checked against consistency rules.

domain-extensible: Domain-extensible schemas [Fav02] provide a core schema describing certain common domain features. The core schema can then be extended. The FAMIX schema of the Moose tool provides a language-independent representation of object-oriented features, the core model,⁵³ which can be extended with language-specific information [DLT00] [TDD00b]. Thus, “FAMIX allows for adding new abstractions and specific attributes to existing abstractions” [TDD00a]. Similarly to FAMIX, the Dagstuhl Middle Model allows extensibility via subclassing [LTP04]

domain-retargetable: Domain-retargetable schemas are domain-neutral per se, allowing the specification of any domain. For instance, the TA exchange format has been used to define schemas for architecture recovery at various levels of abstraction [BGH99] [LA97]. Similarly, RSF has schemas to represent software architectures [Riv00a], C++ [Mar99], Java [vEM02], Web sites [MM01a], C, COBOL, PL/AS, LaTeX, and so on [Til95]. The Rigi tool is a generic graph viewer that can visualize data that adheres to any RSF schema. Telos is an example of a domain-retargetable conceptual modeling language. It is used by the Software Bookshelf to define its data model. The authors of the Software Bookshelf note that as a result, “the information repository is easily extensible with new data or types of data” [FHK⁺97, p. 574].

An example of a customizable parser is Software Refinery’s DIALECT.⁵⁴ Newcomb and Markosian report their experiences with the migration of a COBOL payroll system [NM93]. They give a simple customization example of DIALECT: “the OS/VS compiler used for the payroll system allowed some periods at the end of sentences to be omitted; this syntax had not previously been handled by REFINE/COBOL.” Whereas parsers typically support only a single language and can be customized only with a few command-line

Extractor
customizability

⁵³FAMIX’s core model consists of the following entities: Class, Method, Attribute, Invocation, Access, and InheritanceDefinition.

⁵⁴However, since DIALECT is a LALR(1) parser, it probably needs expert knowledge to actually customize it.

switches,⁵⁵ many lexical analyzer do not target a particular language and can be extensively customized via pattern specifications. Cox and Clark make the following observation about their lexical extractor:

“Lexical tools are often faster to develop than parser based tools and, when developed using hierarchical pattern sets, can be *easily extended or adapted* for novel situations. Extension is performed through the addition of new lexical levels or additional patterns in an existing level” (emphasis added) [CC03].

Most program analyses are fixed in the sense that the reverse engineer cannot turn any knobs to influence the outcome of the analysis (e.g., one cannot trade speed for precision, or vice versa). For instance, there are many clone detection analyses, but few of them can be easily customized to control what constitutes a code clone and what does not. However, flexible analyses can be valuable because it allows the reverse engineer to instruct an analysis to focus its effort on information that is most relevant to a particular reverse engineering task [KM01]. Jackson and Rinard believe that software analyses should give the engineers more control, for instance, to customize the precision of an analysis: “Engineers need different degrees of precision in different situations, at different points in the program, and for different data structures. Applying a single analysis uniformly across the entire program is therefore counterproductive” [JR00]. Atkinson and Griswold have developed a whole-program analysis tool that allows the user to control the precision of its analysis as well as its termination criteria [AG96]. This avoids wasted resources caused by analyses that are more general than a certain reverse engineering activity actually requires. Jahnke has developed a graphical specification language, Generic Fuzzy Reasoning Nets (GFRN), that enables to customize the schema analysis process for database reengineering [Jah99]. He states, “the GFRN approach enables to realize a CARE environment that supports partial automation of the schema analysis process but provides a high amount of customizability and extensibility.”

Analyzer
customizability

In Bassil and Keller’s survey on software visualization tools, 45% of the respondents rated the “possibility to customize visualization” as “absolutely essential” [BK01, F29]. The developers of the Sextant software comprehension tool say, “we conclude that software exploration tools should be extensible to accommodate for domain-specific navigation elements and relationships as needed” [EHMS05]. Even though customizations seem important, Wang et al. say that “existing visualization tools typically do not allow easy extension by new visualization techniques” [WWB⁺03]. They have developed the EVolve software visualization framework, which “is extensible in the sense that it is very easy to integrate new data sources and new kinds of visualizations.” A new visualization is realized by extending the EVolve Java framework, which already provides abstractions for bar charts, tables, dot-plots, and so on. Similarly, Reiss has analyzed why software understanding tools are often not used and concludes that “the primary reason was that they failed to

Visualizer
customizability

⁵⁵Customizations of parsers is difficult to accomplish because it is necessary to understand the particular parsing technique (e.g., LALR or LL) as well as the grammar itself. Furthermore, grammars are often brittle and large. As a result, adapting a Cobol parser to handle a new dialect can take 3–5 months [LV01].

address the actual issues that arise in software understanding. In particular, they provided fixed views of a fixed set of information and were not flexible enough” [Rei01]. He also states that “since we cannot anticipate all visualizations initially, it should be easy to add new graphical objects to the system” [Rei93]. Among the requirements that Favre states for his G^{SEE} software exploration environment is the need for “customizable exploration” [Fav01]. Tilley states that “users need to be able to impose their own personal taste on the user interface . . . The goal of environmental customizability includes modification of the system’s interface components such as buttons, dialogs, menus, [and] scrollbars” [Ti198a, sec. 4.3.3.3]. As already explained, Rigi’s user interface is based on Tcl/Tk, which makes it easy to add new user interactions and to change existing ones. However, Rigi’s graph visualization cannot be easily changed (e.g., the shapes of nodes and the positioning of arcs are fixed). There are also general graph editors that allow various customizations via subclassing (e.g., TGE [KS90], and EDGE [New88] [PT90]), or dedicated specification languages (e.g., EDGE’s GRL). Storey et al.’s visualization evaluation framework addresses customizability of tool interactions:

“Effective interaction to suit particular user needs will normally require a high degree of customization. . . . Saving customizations and sharing customizations across team members may also be important” [SCG05].

Storey et al. have evaluated twelve different visualization tools, concluding that Advizor and Xia/Creole have high customizability; VRCS, Palantir, and Jazz have low customizability; and the rest having no support for customizability.

Whereas my survey suggests that researchers see tool customizability as an important requirement, many tools are lacking in this respect, especially extractors and analyzers. In fact, customizability of extractors and analyzers is often forced by scalability problems. For example, dynamic traces easily can become too huge to be efficiently stored, retrieved, and investigated. As a result, trace extractors can be customized for extracting only information at a certain level of granularity (e.g., functions, blocks, or statements), certain event types (e.g., function return or object creation), or parts of the execution (e.g., code of certain classes or packages) [HL04]. There is also the approach to have a fixed extractor or analysis with the idea to have its result then customized in a separate processing step (e.g., by a subsequent analysis and/or visualization).

Discussion

There are also analyses that are generic (i.e., they can operate independently of the actual data or schemas) and thus do not need to be customized.⁵⁶ A typical example of such an analysis is a graph layout or a textual differencing algorithm. While such generic analyses have the advantage that they are applicable across all schemas, they have the disadvantage that they cannot exploit the domain knowledge encapsulated by a particular schema.

⁵⁶These analyses typically exploit the fact that all schemas adhere to a common meta-schema (or meta-meta-model). For example, all RSF schemas are composed of nodes, arcs, and attributes, even though concrete schemas differ by the actual types of these entities.

3.2.4 Usability

“Nice tool, but it takes some time to get used to it.”

– user feedback for the sv3D visualization tool [MCS05]

Usability can be defined as the ease of use of a system for a particular class of users carrying out specific tasks in a specific environment [Hol05].⁵⁷ This definition emphasizes that usability depends on the context of use as well as the user. Definition

Usability encompasses a set of other quality attributes or characteristics such as [Hol05] [FB04]: Usability characteristics

learnability: The system should be easy to learn so that the user can rapidly begin working with the system.

efficiency: The system should be efficient to use, enabling a user who has learned the system to attain a high level of productivity.

memorability: The system should be easy to recall, allowing the casual user to return to the system without having to relearn everything.

satisfaction: The system should be pleasant and comfortable to use.

Among the goals of software engineering are the construction of systems that users find both useful and usable. The same is true for the construction of reverse engineering tools. Meeting the usability requirement of users has several benefits [FB04]. It improves the product, resulting in productive and satisfied customers; increases the reputation of the product and the developer, potentially increasing the product’s use; and decreases costly redevelopment caused by user complaints. Benefits

Researchers in the reverse engineering field have pointed out the importance of usability as follows: General usability requirement

- In a position statement for a WCRE panel, Müller et al. state, “reverse engineering tool developers not only need to understand the technology side but also need to take the business requirements and the application usability more and more into account” [MWS98].
- Walenstein says in his dissertation about cognitive support in software engineering tools: “The first rule of tool design is to make it useful; making it usable is necessarily second, even though it is a close second” [Wal02, p. 1].
- Discussing tool design and evaluation, Wong assures that “for program understanding tools, careful design and usability testing of the user interface is important” [Won99, p. 19].

⁵⁷Usability is a vague concept. Seffah and Metzker caution that “as a software quality attribute, usability has not been defined consistently by either the researchers and the standardization organizations or the software development industry” [SM04]. Folmer and Bosch survey a number of the different definitions [FB04].

- In a talk entitled *Creating Software Engineering Tools That Are Usable, Useful, and Actually Used* at the University of Victoria, Singer states: “Simply put, if a tool isn’t usable it won’t be used” [Sin04].
- Maccari and Riva have conducted an empirical evaluation of CASE tool usage at Nokia [MR00]. The respondents rated the modeling requirement to “be intuitive and easy to use” as highly useful (i.e., the median value of the responses was above four on a five-point scale). However, based on their experience with existing CASE tools such as Rational Rose, the respondents replied that this requirement is “insufficiently well implemented.” Maccari and Riva state that the result of their survey shows that “the ideal CASE tool should be a *graphical, easy to use* tool.”
- In a survey, participants rated the importance of requirements for software visualization tools [BK01]. The requirement “ease of using the tool (e.g., no cumbersome functionality)” was selected as the second-most important practical aspect, which 72% rated as very important (i.e., the highest value on a four-point scale) [BK01, P8].

There are two approaches how one can design for usability [FB04]:

Approaches to
usability design

product-oriented: Product-oriented approaches consider usability to be a product characteristic that can be captured with design knowledge embodied in interface guidelines, design heuristics, and usability patterns. For instance, Toleman and Welsh report on the evaluation of a language-based editor’s interface based on 437 guidelines [TW98].⁵⁸ This catalog covers functional areas such as data entry, data display, sequence control, and user guidance [SM86].

The evaluation of the usability of reverse engineering tools is often focused on the user interface. Toleman and Welsh say, “user interface design guidelines are an important resource that can and should be consulted by software tool designers” [TW98]. In Bassil and Keller’s survey, 69% believe that “quality of user interface (intuitive widgets)” is a very important requirement [BK01, P12]. Reiss, who has implemented many software engineering tools, believes that “having a good user interface is essential to providing a usable tool” [Rei05]. He provides a rationalization for the user interface choices of the CLIME tool, but does not support his decisions with background literature. For the BLOOM software visualizer, Reiss emphasizes that both usefulness and usability are important: “While it is important to provide a wide range of different analyses and to permit these to be combined in order to do realistic software understanding, it is equally important to offer users an intuitive and easily used interface that lets them select and specify what information is relevant to their particular problem” [Rei01]. Storey’s cognitive framework emphasizes usefulness aspects for comprehension tools, but also has a design element that requires to

User interface

⁵⁸An existing catalog of 944 guidelines was chosen, of which 46.3% were deemed applicable for the editor’s evaluation.

“reduce UI cognitive overhead” [Sto98, p. 149]. Storey et al. further elaborate on this design element, stating that “poorly designed interfaces will of course induce extra overhead. Available functionality should be visible and relevant and should not impede the more cognitively challenging task of understanding a program” [SFM99a].

Web interfaces exemplify the importance of judging usability based on the target users. It is widely believed that Web applications provide a simple and convenient interface because they are familiar to many users [TS96]. While this is most probably true for the users of reverse engineering tools, other users might struggle with this new interface paradigm, which differs significantly from graphical desktop applications. Dennis relates his experience of a new system with a Web interface:

Web interfaces

“While most users were quite enthusiastic about our Web groupware system, some weren’t. Those users had the same expectations for the Web system as they did for their Windows systems. They expected the standard interface concepts such as double-clicking to open topics and dragging-and-dropping to move items. The fact that the system used the Web was unimportant compared to the lack of standard Windows features” [BB98].

Design heuristics suggest properties and principles that are believed to have a positive effect on usability. Heuristics address issues such as consistency, task match, memory-load, and error handling. Bass et al. have collected 26 general usability scenarios [BJK01]. A scenario describes an interaction that a user has with the system under consideration from a usability point of view. Examples of scenarios are Aggregating Data (i.e., systems should allow users to select and act upon arbitrary combinations of data), Aggregating Commands (i.e., systems should provide a batch or macro capability to allow users to record, aggregate and replay commands), Providing Good Help (i.e., systems’ help procedures should be context dependent and sufficiently complete to assist users in solving problems), Supporting International Use (i.e., systems should be easily configurable for deployment in multiple cultures), Modifying Interfaces (i.e., system designers should ensure that their user interfaces can be easily modified), and Verifying Resources (i.e., systems should verify that all necessary resources are available before beginning an operation).

Design heuristics

To my knowledge, there is no catalog of design heuristics for the reverse engineering domain. However, researchers sometimes relate their experiences, providing tidbits of ad hoc usability advice. Examples of such tidbits, grouped by usability characteristics, are:

learnability: Respondents in Bassil and Keller’s survey view learnability as relatively less important; less than half see “ease of learning and installation of the tool” as a very important aspect [BK01, P7].⁵⁹ For search tools that are based

⁵⁹Unfortunately, this item in the survey combines two distinct attributes: learnability and ease of installation. The authors decided to group these together because both attributes can be considered as a necessary up-front investment to get productive with the tool (private email correspondence with Rudi Keller).

on pattern matching, Bull et al. require “an easy to specify pattern” [BTMG02]. This requirement trades improved query learnability (as well as simplicity and specification time) for less expressive power [PP96].

efficiency: According to Reiss, a tool’s usage should “have low overhead and be unintrusive” [Rei02]. Especially, it should not interfere with existing tools or work practices.

Reverse engineers often manually inspect, create and modify the data represented with an exchange format. To simplify this activity, these formats should be human readable and composable [Riv00b] [MM00].

memorability: The scripting interface of a tool should not overwhelm the user with too many commands. For instance, Moise and Wong made the experience that “the Rigi command library was difficult to learn and awkward to use with the sheer number of commands” [MW03].

satisfaction: In order to keep reverse engineers motivated, tools should be enjoyable to use. Especially, “do not automate-away enjoyable activities and leave only boring ones” [LS96].

Tools should be designed to “keep control of the analysis and maintenance in the hands of the [users]” [Cor03]. Otherwise the user may feel threatened and devalued by the tool.

process-oriented: Process-oriented approaches such as user-centered design consider usability as a design goal that can be achieved with a collection of techniques involving the end users (e.g., task analysis, interviews, and user observations). Singer answers the question of “How do you make a tool usable?” with the recommendation to “conduct pilot tests on [the] target user group,” and to “bring it to users early and often” [Sin04].

The SHriMP tool has a history of user studies, which also had the goal to improve the tool’s usability. For instance, a pilot study (involving 12 users who were video taped using think-aloud⁶⁰) has lead to recommendations for interface improvements to SHriMP (and Rigi) [SWF⁺96] [Sto98, ch. 8]. These recommendations were then used to redesign SHriMP’s interface [Sto98, ch. 9]. The redesign involved, for instance, a more effective navigation of the visualized graphs combining context+detail with pan+zoom, alternative methods of source code browsing, and the introduction of modes to reduce the cognitive overhead of the users during navigation. The new interface was then evaluated with another user study (which used videotaping and think-aloud, but also a questionnaire and an informal interview) [SWM97] [Sto98, ch. 10]. Besides SHriMP, the TkSee tool is also an example of a reverse engineering tool that has employed user studies (see below).

⁶⁰The think-aloud approach “requires subjects [to] verbalize their thoughts while performing a given task. The resulting stream of utterances helps indicate the way a subject is reasoning about how to perform the task” [OBB06].

Awkward usage scenarios or work patterns can also give hints on how to improve usability. For instance, observing the work of professional software engineers, Singer and Lethbridge found that they did “jumping back and forth between tools, primarily Unix command line (performing `grep`) to editor and back. This jumping involved the use of cut and paste to transfer data and was frequently awkward” [SL98]. This scenario points towards a better integration and interoperability of tools to improve usability.

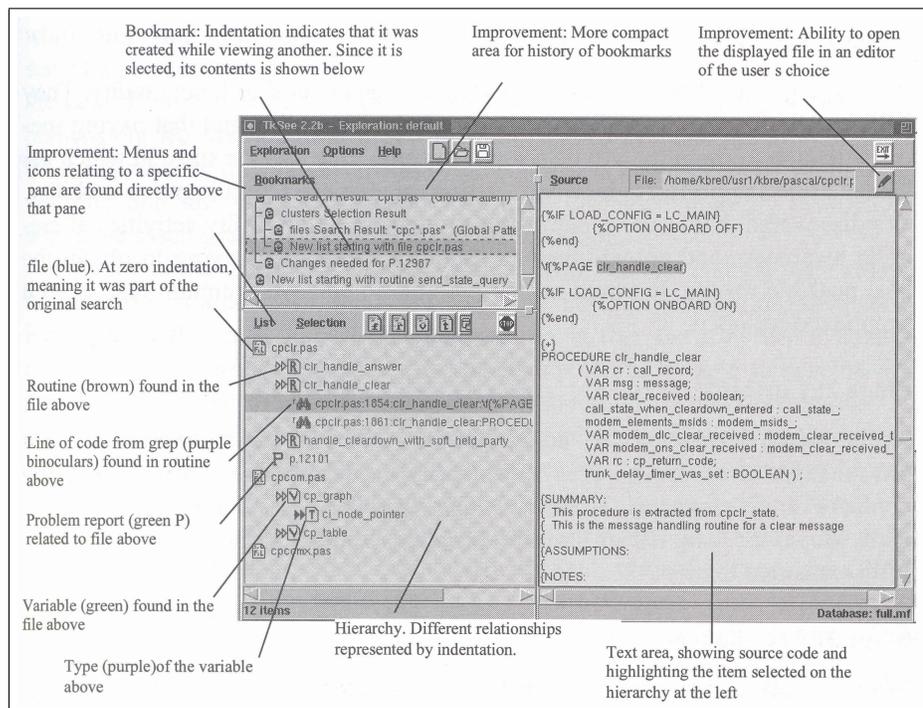


Figure 7: Changes to TkSee resulting from usability evaluation ([LH01, fig. 4.3])

The design and evolution of the TkSee search tool is an example of a tool-building effort that has combined both product-oriented and process-oriented approaches in the form of guidelines and user studies to improve the tool's usability. A product-oriented approach was followed by evaluation TkSee based on Nielsen's usability guidelines [LH01]. Three evaluators identified 114 usability problems. The types of problems found were poor or missing feedback (e.g., what has happened following an interaction), possible confusion about tool behavior, possible misinterpretation (e.g., meaning of labels or menu items), poor labeling, lack of labeling, lack of consistency, poor graphical design, unnecessary features, lack of needed features, lack of robustness (e.g., tool crashes or hangs), incorrect behavior, and non-optimal interaction. The usability analysis of TkSee showed that it can be important to have several evaluators with different backgrounds. One of the evaluators had a background in usability, but no background about the problem domain (i.e., program

comprehension). This person tended to find general usability problems related to feedback, labeling, and graphical design, et cetera. In contrast, another evaluator that was already knowledgeable about TkSee and the problem domain tended to point out missing features and incorrect behavior. TkSee was also evaluated with a user analysis involving videotaping and think-aloud usability testing [LH01, p. 83]. Eight participants found 72 problems, of which 53% had already been identified before by the evaluators. This shows that both product-oriented and process-oriented approaches are complimentary and that both should be used to evaluate a tool's usability. Figure 7 shows a screenshot of TkSee after its redesign, identifying some changes that were made as a result of the usability evaluation.

Tool developers often are not aware of the importance of usability or do not know how to achieve it. However, in order to systematically identify and improve a tool's usability problems, it is necessary to change tool developers' attitudes towards usability [Bev99]. Based on their experiences with the TkSee tool, Lethbridge and Herrera identify four stages that software development teams can go through in a transition towards increasing awareness of usability [LH01]:⁶¹

skepticism: At this stage, developers believe they can create systems with adequate usability without explicitly considering it in their development process.

curiosity: In this stage, developers start to realize that their system has usability problems. However, they are reluctant to address the issue because of fear of losing control over development.

acceptance: When this stage is reached, developers accept the assistance of usability experts and follow the experts' recommendations.

partnership: In this most advanced stage, usability is considered a critical design goal of the system. All stakeholders⁶² of system work together to achieve usability, following a disciplined process.

The TkSee team has reached the level of acceptance, even though there is a constant struggle to maintain this level because of staff turnover. The team is still far from the stage of partnership, but it is "questionable whether it could or should ever be reached in a university research environment" [LH01].

In 1991, Grudin observed that "resistance to unfriendly systems is growing. There is growing competitive pressure for usability in the marketplace, particularly in mature application domains" [Gru91]. More than a decade after this statement, it is questionable whether usability of software has improved drastically. Whereas the problem of usability seems to have more visibility, it is still difficult to overcome due to other, competing pressures such as feature creep and time to market.

Discussion

⁶¹An alternative scale is provided by the INUSE project, which distinguishes six levels: ignorance, uncertainty, awakening, enlightenment, wisdom, and certainty [Bev99].

⁶²A stakeholder is any person or representative of an organization who has a vested interest in the outcome of the project or whose opinion must be accommodated [Kru99, p. 239].

Usability is recognized as a problem by researchers, but it is often addressed in an ad hoc manner. Tool developers rarely discuss how they established the usability of their design. Toleman and Welsh testify, “in general, the design rationales for software tools that are available rarely indicate the basis for the design of the user interface” [TW98]. There is the underlying problem that researchers in reverse engineering have neither made an attempt to define nor clarified what they mean by usability. As a result, usability is often judged subjectively by the tool developers. Toleman and Welsh criticize that “software tool designers consider themselves typical users of the tools that they build and tend to subjectively evaluate their products rather than objectively evaluate them using established usability methods” [TW98]. Too often, usability is only superficially addressed. Lanza and Ducasse address the usability of their CodeCrawler visualization tool by saying that “our tool has been downloaded over 2,000 times and, although we haven’t performed a user survey yet, from personal and e-mail discussions with the users, we have learned that after a short learning time they know what they can get out of each view” [LD03]. Whereas these indications are encouraging for CodeCrawler indeed, they cannot replace a more formal usability assessment (such as exemplified by SHriMP and TkSee).

As a start, researchers should develop usability guidelines for the reverse engineering domain. Storey’s cognitive dimensions framework is mostly focused on improving the usefulness of a program comprehension tool, but not its usability. A complimentary framework that explicitly addresses usability might be helpful to improve the current practice of usability design and evaluation.

3.2.5 Adoptability

“Technologists tend to think that if they build a good thing, people will find their way to it and adopt it on their own, based on its inherent goodness...Wrong.”

– Lauren Heinz [Hei01]

For almost all new ideas, practices, technologies, tools, and other innovations in general,⁶³ there is the concern of how to get them adopted. As the above quote suggests, adoption of innovations cannot be taken for granted, regardless of the perceived benefits by its proponents. This painful experience has been repeatedly made by different innovators in diverse areas. For software engineering innovations, Finkelstein and Kramer summarize the made experiences as follows [FK00]:

Adoption problems
of innovations

“It has taken a long time for researchers to realize that we cannot expect industry to make very large big-bang changes to processes, methods and tools, at any rate without substantial evidence of the value derived from those changes.”

An example of a famous adoption problem in the software engineering area are CASE tools [JH98] [MS96] [Kem92]. Even though CASE tools were promoted as significantly

⁶³Roger defines an innovation as “an idea, practice, or object that is perceived as new by an individual or other unit of adoption” [Rog95, p. 11].

increasing software development effectiveness in terms of productivity and quality, many developers did not use these tools or abandoned them later on—leading to questions such as “Why are CASE tools not used?” [Iiv96].

The attempt to move towards a common exchange format for reverse engineering is an interesting example of the adoption of a new standard within a research community. The goal of a common exchange format is to simplify tool interoperability. However, to achieve this vision, a diverse group of stakeholder have to agree to adopt a standard first. Establishing a standard exchange format or schema is difficult because existing tools have to be modified for compliance, which may not be economical [FSH⁺01]. But without a critical mass, an exchange format does not make the transition to a standard exchange format. Adoption of a new format can be encouraged by addressing the functional and non-functional requirements of its users.⁶⁴ In other words, the exchange format should be an “early and clear win for adopters” [Hol00].

Common exchange
format

Perhaps the most import theory that is able to explain adoption is diffusion of innovations [Rog95]. The theory’s roots are in sociology, but there are hundreds of publications that have applied it to study the adoption of innovations in a vast number of fields: anthropology, (rural) sociology, education, public health, communication, marketing and management, geography, economics, and technology. An important adoption factor is the characteristics of an innovation as perceived by the potential adopter. Diffusion of innovations has identified the following characteristics as most significant [Rog95, p. 15]:

Diffusion of
innovations

relative advantage (+): Relative advantage is the degree to which an innovation is perceived to be better than what it supersedes. The immediacy of the rewards of adopting an innovation is also important and explains why preventive innovations have an especially low adoption rate.

Relative advantage can mean that reverse engineers notice that a certain tool allows them to perform certain tasks with more ease, in less time, or with higher job satisfaction.

compatibility (+): Compatibility denotes the consistency with existing values, past experiences and needs. Generally, innovations are understood by putting them in relation with the familiar and the old-fashioned.

The tool designers of `sgrep` try to improve adoption by making their tool compatible with a popular existing search tool, `grep`. They state, “`sgrep` is designed to be used in place of `grep`, so it is important that many of the design decisions found in `grep`, transfer over to `sgrep`” [BTMG02]. Since `grep` is a command-line tool, `sgrep` follows this pattern: “Although graphical user interfaces are often easier to use for novice users, we believe that *familiarity is more important than ease of use*,

⁶⁴Whereas it is clear that the users of an exchange format are researchers in the reverse engineering field, it is difficult to assess the requirements of this rather diverse community. Proposals for standard schemas face the same problem; a group of researchers proposing a schema for C/C++ note, “there is one fundamental issue that we have not yet resolved: who are the end users of this schema and what are their requirements?” [FSH⁺01].

for the kinds of tasks we envision for `sgrep`” (emphasis added). The TkSee search tool also maintains compatibility with `grep`. It returns hits as lines in a file, which are identical to a search with `grep`, but also has the ability to search for more specific software objects such as functions and variables. Since TkSee is targeted at developers in a Unix environment that know `grep` intimately, the developer can start with `grep`-like searches to familiarize themselves with TkSee and to gain confidence in its abilities before transitioning to more advanced tool functionalities. The TkSee developers state that “part of our design philosophy was to provide the developers a superset of functionality they had already” [LH01, p. 78].

complexity (–): Complexity is the difficulty of understanding and using an innovation.

Adoption of a tool can be increased by making it easier to use, or by providing training sessions and appropriate documentation. Complexity can be also reduced by identifying and eliminating unnecessary tool features.

trialability (+): Trialability denotes the degree to which an innovation can be experimented with, without committing to it.

The designers of the Software Bookshelf ease trialability of their tool by requiring that the user “should still be able to use tools already favored and in use today” [FHK⁺97, page 569]. The authors of a work practice study involving the TkSee tool believe that an important factor in the adoption of the tool by developers was that “we allowed them to continue their existing work practices (e.g., use of `grep`), rather than forcing them to adopt a radical new paradigm” [SL98]. Also, the TkSee search tool can be easily tried out because reverse engineers can readily switch between TkSee and other search tools that they were using before. Martin suggests that a search tool could offer different front-ends (e.g., a textual one similar to `grep` and a graphical one integrated into an IDE) to cater to different user types [Mar03]. A tool is easier to try out if it is easy to install. The REportal reverse engineering tool is implemented as a Web portal. Users can upload the source code that they want to analyze and then run analyses; thus “users are not required to install any software on their computers in order to use the portal services” [MSC⁺01].

observability (+): Observability is the degree to which the results of an innovation are visible to others.

For instance, if a tool is visibly used for other developers, it can have a beneficial impact on adoption.

These characteristics help to explain the rate of adoption. Adoptions that are positively related (“+”) with the above characteristics will be adopted more rapidly than other innovations. Besides these characteristics, there are other factors that determine adoption, for instance, communication channels, nature of the social system, activities of change agents, and individual/collective decision-making.

Developers of reverse engineering tools have mostly ignored the question of whether their tools are actually adopted by software developers and maintainers. For program understanding tools, Mayrhauser and Vans have observed expectations that users better adapt to a tool if they want to use it: “we still see attitudes reflected in tool builders’ minds that if we just teach programmers to understand code the way they ought to (i.e., the way their tools work), the understanding problem will be solved” [vMV93]. Thus, instead of lowering adoption barriers and increasing the users’ incentives to adopt, this attitude expects users to pick up a tool in spite of the raised adoption hurdles. In the last few years, however, the reverse engineering community has started to pay more attention to this question. This trend is exemplified by the following sources:

Adoption problems
of reverse
engineering

- In 1996, Rugaber and Wills already point out that there is an adoption problem of reverse engineering tools: “Reengineering research has had notably little effect on actual software reengineering practice. Most of the published papers in the field present techniques supported by prototype tools; few of which have actually been used on real projects” [RW96].
- Eight years later, the organizers of the *Fourth International Workshop on Adoption-Centric Software Engineering* come to a similar conclusion: “Research tools in software engineering often fail to be adopted and deployed in industry” [BLM⁺04].
- Lethbridge makes the general observation that “one of the beliefs that motivates software engineering tools builders is, ‘if we build it, they will come.’ Unfortunately, they often don’t come and we wonder why” [Let04]. Similarly, Wong stresses that it is not enough to devise a new technique or tool and “simply expect practitioners to pick it up and adopt it” [Won99, p. 93].
- In the context of software maintenance, Zayour and Lethbridge say, reverse engineering tools “have a major ‘low adoption’ problem among software engineers in industry” [ZL00].
- Software exploration tools use graphical presentation to visualize information about a software system. Even though researchers perceive these tools as valuable for reverse engineering and maintenance tasks, Storey reports that “despite the large number of software visualization tools, few of these tools are used in practice” [SFM99a]. Storey et al. use the adoption problem as motivation to propose a framework of cognitive design elements to guide tool design.
- In a roadmap paper for reverse engineering research for the first decade after the year 2000, Müller et al. state their believe that “perhaps the biggest challenge to increase effectiveness of reverse engineering tools is wider adoption; tools can’t be effective if they aren’t used” [MJS⁺00].
- In his dissertation, Wong demands from researchers to “address the practical issues underlying reverse engineering tool adoption” [Won99, Requirement 12]. Discussing

lessons learned from his Reverse Engineering Notebook, he says, “make adoption issues an integral part of reverse engineering tool research and design” [Won99, Recommendation 1].

There are few researchers who see the adoption of their tools as a first-class requirement for their research endeavor. A notable exception is the Adoption-Centric Reverse Engineering (ACRE) project, which explores the adoption problem of reverse engineering tools [Mül01].⁶⁵ It has initiated a series of four workshop on Adoption-Centric Software Engineering (ACSE 2001–2004) [BLM⁺04]. ACRE addresses the adoption problem with two lenses, cognitive support and interoperability. Using these two lenses, ACRE’s “main hypothesis is that in order for new tools to be adopted successfully, they must be compatible with both existing users and other tools” [Mül01]. Since most research tools only support a few selected program understanding or maintenance tasks, reverse engineers typically have to integrate them with other tools to use them effectively. ACRE proposes to investigate the use of data, control, and presentation integration technologies such as XML protocols, the GXL exchange format, Web browsers, ECMAScript, SVG, Eclipse, and Web services to make tools more interoperable and thus more adoptable. The other lens, cognitive support, refers to the means by which tools assist the cognitive work of their users (i.e., thinking and reasoning) [Wal02]. Examples of everyday tools that provide some form of cognitive support are shopping lists, address books, and pocket calculators. Without them, certain tasks would have to be performed with an increased cognitive load (e.g., in terms of memorization and computation).

Using a tool effectively and in a way that optimizes its cognitive support can require a significant learning effort. This is the case, for example, for CASE tools [Kem92]. Looking at their own tools, the principal investigators of the ACRE project say that they “had inadequate cognitive support to ease the complex software engineering tasks programmers perform” [Mül01]. The ACRE project proposes to build software engineering tools such that users can quickly optimize cognitive support based on prior tool experiences. This can be achieved by supporting strategic knowledge⁶⁶ that users already know, and by extending popular tools that are already used by software engineers such as office suites, tools for collaboration, and development environments. Intuitively, “users will more likely adopt tools that work in an environment they use daily and know intimately” [MWW03].

Lethbridge has explored adoption of program understanding tools by industry in the context of two of this tools, the TkSee search tool and the DynaSee trace visualizer [LH01] [ZL01] [ZL00] [Let00] [SL98]. Both tools are used by Mitel for the maintenance of telecommunications software. Zayour and Lethbridge point out that reverse engineering tools face an additional adoption hurdle because “their adoption is generally ‘optional’ ” in the sense that they are not a necessity to complete, say, a maintenance task. This distin-

⁶⁵The leading participants of this project are Hausi Müller, Margaret-Anne Storey (both University of Victoria), Kenny Wong (University of Alberta), and Joe Wigglesworth (IBM Toronto Lab).

⁶⁶Strategic knowledge is applicable across different tools, enabling users to transfer it from one tool to another [BJ97]. A simple example of strategic knowledge is the selection and repeated copying of an entity—it works for many diverse applications such as Word processors, spreadsheets, and CAD systems.

guishes them from other tools such as text editors and compilers. Zayour and Lethbridge propose a method based on cognitive analysis to improve tool adoption. Their model assumes that tool adoption is governed by the user's perception of ease of use, and perception of usefulness.⁶⁷ Note that these attributes are determined by the *perception* of the user and not as they are seen by experts. The model further assumes that a tool that reduces the cognitive load of users will positively influence both attributes and thus tool adoption.

In later work, Lethbridge considers adoption using three factors: costs, benefits and risks of tool use [Let04]. Potential adopters often do not perform a formal analysis of these factors, relying on their "gut feeling" instead. Examples of costs of use are (c1) purchasing of the tool, (c2) purchasing of extra hardware or support software, (c3) time to install and configure the tool, and (c4) time to learn the tool. Examples of benefits of use are (b1) time saved by the tool, and (b2) value of the increased quality of work done. Examples of risks are (r1) costs are higher than expected, (r2) benefits are less than expected, (r3) unintended negative side-effects (e.g., data corruption), (r4) discontinued tool support, and (r5) difficulty to revert to previous work environment. Discussing the factors, Lethbridge says that "in addition to perceiving costs and benefits differently, adopters will more intensively perceive the risks, and the more risks they perceive, the more their perceived benefits must exceed their costs of adoption to take place" [Let04]. In contrast, tool researchers tend to focus on costs and benefits, ignoring or down-playing the risks. The introduced factors can be used to assess tool-adoption scenarios. For instance, adopting a reverse engineering tool that is built on top of an office suite (as envisioned by ACRE) should have low purchasing costs assuming the office suite is already used (c1 and c2), a simple installation process if the tool is provided as a plug-in (c3), and favorable learning curve resulting in saved time (c4 and b1). On the other hand, updating the office suite might render the plug-in inoperative (r4) and users might become trapped in a certain data format (r5).

Tilley et al. have looked at the adoption of research-off-the-shelf (ROTS) software [TH06] [THP03]. They say,

"in our opinion, adoption is one of the most important, yet perhaps least appreciated, areas of interest in academic computer science circles. . . . Indeed, it can be argued that 'transitionability' as a quality attribute should receive more emphasis in most software projects" [THP03].

In applied fields such as software engineering, "it may be a measure of success for the results of an academic project to be adopted by an industrial partner and used on a regular basis." However, whereas adoption is a desirable (long term) goal for a research project, it is not a necessary criterion for success. This is caused by the academic reward structure, which emphasizes publications rather than workable tools. As a result, the adoption of ROTS software is complicated by lacking "completeness (e.g., a partial solution due to an implicit focus on getting 'just enough' done to illustrate the feasibility of a solution,

⁶⁷Perceived ease of use is "the degree to which a user believes that using the system will be free from effort;" perceived usefulness is "the degree to which a user believes that using the system will enhance performance" [MD97].

rather than going the ‘last mile’ to bring the prototype to market).” This is especially the case if the software is the result of a one-person effort produced as part of a Master’s thesis or dissertation. Additional complications for adoption are “understandability (e.g., a lack of high-quality documentation)” and “robustness, (e.g., an implementation that is not quite ready for prime time.” Huang et al. also look at the relationship between academia and industry: “Both parties know that they have a symbiotic relationship with one another, yet they seem unable to truly understand what each other needs” [HTZ03]. Industry has a potential interest in research results that can mature and then be integrated into existing processes to improve software development. To encourage interest from industry, researcher have to solve relevant problems and not work on problems that are “removed from the current needs of potential users.” Also, adoption of tools and techniques by industry could be encouraged with third-party case studies and quantitative data, but these are rarely available for ROTS software [TH02].

The adoption of an innovation can be explained by the cost that it causes the users and their organization. Patterson makes the point that research software is free of charge, but this does not mean that adoption and use of it has no cost—there is a difference between cost of purchase and the cost of ownership [Pat05]. Tilley and Distanto say, “ultimately, people will only adopt a technique if they see significant benefit in its use that far outweigh the costs associated with learning and using the technique in the first place” [TD04]. Storey et al. state that “the adoption of any tool has a cost associated with it. Economic cost is a key concern, in addition to other costs such as the cost of installing the tool, learning how to use it, and the costs incurred during its usage” [SCG05]. In their tool survey, Bassil and Keller did ask for the importance of the “cost of the tool” [BK01, P1]. Interestingly, 50% of the respondents from academia rated this aspect as “very important,” compared to only 32% of respondents from industry.

Cost

Researchers in the software engineering area have drawn from existing work to understand and improve adoption of tools, technologies, and methods. ACRE draws from ideas of cognitive science to understand, measure and evaluate the cognitive support of tools [Wal03b] [Wal03a]. Storey et al. include cognitive support in their evaluation framework of software visualization tools [SCG05]. Sun and Wong apply cognitive theories of human perception, especially Gestalt theory, to evaluate the SHriMP tool and to suggest improvements for it [SW04]. Lethbridge et al. incorporates elements of the technology acceptance model [MD97] [Dav89] and diffusion of innovation theory [Rog95] to explain tool adoption. Tilley et al. look at the adoption of research tools through the lenses of Moore’s *Crossing the Chasm* and Christensen’s *The Innovator’s Dilemma* [THP03]. Huang et al. look at the relationship between research and academia by modeling the situation using the Chinese philosophy of Yin and Yang [HTZ03]. Examples of other applicable theories and models are cognitive fit [Ves91] [GP96, sec. 2.1] [Wal02, sec. 3.2.4], consumer behavior theory [HL02b], technology transfer [Pfl99] [Aoy06], SEI’s Technology Transition Practices,⁶⁸ and technology readiness level [GGS⁺02].

Applicable theories

⁶⁸<http://www.sei.cmu.edu/ttp/>

Researchers have suggested many potential factors that affect tool adoption, summarized as follows:

tool: Researchers have mostly focused on factors that can be attributed to the tool itself.

To get adopted, tools have to be both useful and usable. Storey says, “although there are many software exploration tools in existence, few of them are successful in industry. This is because many of these tools do not support the right tasks” [Sto98, p. 154]. Bull et al. state, “in any field, ease of use and adaptability to the tasks at hand are what causes a tool to be adopted” [BTMG02]. Wong believes that “lightweight tools that are specialized or adaptable to do a few things very well may be needed for easier technology insertion” [Won96]. He also says, “by making tools programmable, they can more easily be incorporated into other toolsets, thus easing an adoption issue of tool compatibility” [Won99, p. 94]. The ACRE project identifies a number of factors that affect the adoption of reverse engineering tools: “Important barriers to adopting [software engineering] tools include the unfamiliarity with users, their unpolished user interfaces, their poor interoperability with existing development tools and practices, and their limited support for the complex work products required by industrial software development” [Mül01]. Devanbu points to inadequate performance of research tools, which are often not intended for large-scale applications [Dev99b]. In contrast to most research tools that require some effort and expertise for installation, popular tools are easy installed or even already pre-installed [Mar03].

user: For adoption, besides the characteristics of the tool, characteristics of the tool users play an important role.

When starting to use a tool, users often want positive feedback very quickly. Tilley makes the point that “it is an unfortunate fact that many developers will give a tool only a short window of opportunity to succeed. If they cannot get the tool up and running in 10 minutes and see real results, without looking at the manual, they will often abandon the tool” [Til98b]. Many users would not even consider a very small trial period no matter of the potential benefits that it promises—if users are happy with their existing tools they do not see the need of trying (yet) another one. Devanbu says,

“developers are pressured by schedules, and keenly aware of the need to meet cost, schedule or quality requirements. This engenders a conservative bias towards simple, and/or familiar tools, even if somewhat outdated. Builders of complex tools with steep learning curves (even ones promising significant gains) face the daunting hurdle of convincing busy developers to invest training time” [Dev99b].

The TkSee tool has collected some experiences with developers at Mitel. Less than half of the developers at Mitel used the tool for significant work over a two-year

period. For users who did not adopt, Lethbridge and Herrera found that “at some point during their learning attempts, many users had concluded that further learning was not worth additional investment of their very limited time” [LH01, p. 81]. Interestingly, user that did not adopt tended to state reasons that revealed misconception of the TkSee tool “either because it had not proved rapidly learnable or else because they had found some aspect of it difficult to understand” [LH01, p. 81]. Tk-See was introduced to the developers without “extensive documentation or training.” Lethbridge and Herrera elaborate on this point, “we feel sure that a more proactive training program might have helped increase adoption to some extent; however, we do not feel that more extensive documentation would have helped much - we hardly ever observed anyone look at the existing documentation” [LH01, p. 82].

organization: Often developers do not make the decision to adopt a tool all by themselves because they are constrained by their organization.⁶⁹ This is true for industry, which often mandates certain tools, as well as open source projects that assume a certain toolset (e.g., GCC, CVS, and Bugzilla).

From an organization’s perspective, “adopting a different toolset or environment discards the hard-earned employees’ experience” [Jaz04]. Furthermore, “changing tools requires changing processes, which is also an expensive undertaking. As a result, the state of tool adoption in industry is rather static and conservative” [Jaz04]. Devanbu also stresses that tools have to integrate smoothly into an existing process: “Software tools work best if their operation is well-tuned to existing processes. Thus, a tool that generates paper output may not be helpful in an email-based culture” [Dev99b]. Similarly, Wong says “software understanding techniques and tools need to be packaged effectively and made compatible with existing processes, users, and tools” [Won99, p. 93].

In a social system such as an organization, the adoption of innovations can be promoted by change agents or innovation champions [Rog95, page 398]. Lethbridge and Singer have rediscovered this approach for tool adoption of TkSee: “We had significant difficulty introducing new tools. One technique that seems to hold promise is to train a single individual (in our case somebody new to the organization) and have him or her act as a consultant for others” [LS97].

Whereas the diffusion of innovations theory has developed characteristics that hold across innovations and social systems, there might be also the need to consider domain-specific characteristics. Cordy reports his experiences with the technical, social, cultural, and economical drivers of the Canadian finance industry and financial data processing in general, which he gained during six year of project work with Legasys Corporation [Cor03]. In Cordy’s experience, resistance to tool adoption is strong because of unhappy past experiences with many inadequate and premature

⁶⁹Rogers defines an organization as “a stable system of individuals who work together to achieve common goals through a hierarchy of ranks and a division of labor” [Rog95, page 375].

CASE tools. As a result of the pressure of quick, low-risk enhancements of financial applications, for maintainers “only the source is real” [Cor03]. Thus, reverse engineering tools have to present results in terms of source, and not abstract diagrams. Also, robust extractors are needed because “having no answer is completely unacceptable, and programmers will rapidly drop any analyzer that fails to yield answers due to parse errors.” The decision to adopt a tool is not made by upper management, but individually by the maintenance programmers and their group manager. In order to convince programmers to adopt a maintenance tool, it is important that they do not feel threatened by it; the workflow of the tool should be such that “all control is left in the hands of the programmer.” Cordy says, “this philosophy of assist, don’t replace, is the only one that can succeed in the social and management environment of these organizations.” Cordy believes that

“by studying the maintenance culture of each industrial community, by treating their way of doing things with respect, and by working to understand how our techniques can best be fit into their existing working environment, we can both increase chances of adoption and enhance our own success” [Cor03].

There are many factors that influence the adoption of a tool—and many of these cannot be influenced by the tool developers directly. Rifkin reaches a similar conclusion when he says that

Discussion

“as designers of processes and tools that we want adopted by others, we should understand that there is only so much power in the technical content of our processes and tools” [Rif03].

However, tool developers should make an effort by leveraging the factors that they are able to influence to increase the likelihood of tool adoption. To increase the incentives of academic researchers who conduct applied research to focus more on the adoption of their proposed tools and techniques, it is necessary to change the academic reward structure. Researchers already have an incentive to raise adoption to a first-class requirement because an adopted tool has indirectly proven its usefulness. However, as Storey points out, the opposite is not necessarily true: “A lack of adoption is not enough to indicate that a tool is not useful as there are many barriers to adoption (e.g., seemingly trivial usability issues can impede usage of a tool)” [Sto05].

Besides reverse engineering, other computer science areas have discussed adoption of their tools and techniques (e.g., object-orientation [dCBB⁺93], product lines [Kru02], open source [WW01] [RE04], groupware [Gru94] [WFS02], and functional programming [Mor04]). It is encouraging that a growing number of researchers have started to realize that adoption is an important challenge that needs to be addressed. Unfortunately, these efforts are still immature. Suggestions to improve adoption are often based on guesswork without providing an underlying theory, or apply existing theories without empirical data.

3.2.6 Other Quality Attributes

My tool requirements survey has uncovered five major quality attributes (i.e., scalability, interoperability, customizability, usability, and adoptability), which have been presented in the preceding sections.

Besides these requirements, I also found a number of other quality attributes. However, these have been addressed in comparably less depth by tool researchers. Thus, I treat them in this dissertation as important, but minor requirements. The minor non-functional tool requirements are as follows:

Minor requirements

interactivity: Atkinson and Griswold make the observation that “program understanding tasks are interactive, unlike compilation, and an analysis such as slicing is often applied iteratively to answer a programmer’s question about the program” [AG96]. Since program understanding and reverse engineering are highly interactive and iterative activities, tools that want to support these activities should be interactive as well. In the context of database reverse engineering, Hainaut et al. require that “the tool must be highly interactive” [HEH⁺95]. Wong states that reverse engineering tools should “provide interactive . . . views, with the user in control” [Won99, Requirement 15]. The need for interactivity is most pronounced at the visualizer and is typically realized by means of direct manipulation [LD03].

However, tool interactivity has to be balanced with suitable automation, otherwise significant reverse engineering tasks might suffer from the large number of required user interactions. Cremer et al. summarize their reengineering experiences with the REforDI tool as follows [CMW02, p. 271]:

“The case study showed that neither pure interactive nor pure algorithm driven reengineering of complex legacy systems is feasible. If the legacy system is too large, interactive, humancentered re-engineering is too time-consuming and fault-prone to be efficient. . . . Therefore, we recommend to have both an interaction facility and a number of different re-design algorithms in order to obtain the best results possible.”

exploratory: Besides interactivity, a tool should also be exploratory. In fact, interactivity seems a necessary prerequisite for an exploratory tool. In the words of Hainaut et al., “the very nature of the reverse engineering activities differs from that of more standard engineering activities. Reverse engineering a software component, and particularly a database, basically is an *exploratory* and often *unstructured* activity” [HEH⁺95]. To accommodate this process, they state the requirement that “the tool must allow very flexible working patterns, including unstructured ones.”

Lethbridge and Singer report the following findings [LS97]: “There was strong evidence that software engineers in our study desire tools to help them explore software. They use such tools heavily already and want improvements.” Wong states

that “tools for software understanding must not impose a rigid comprehension strategy nor some awkward process” [Won99, p. 11]. Among the tool features that Storey recommends for program comprehension tools are various forms of “browsing support” [Sto05]. In her cognitive design framework, she also requires tools to “provide arbitrary navigation” [Sto98, p. 149]. Ducasse et al. say that “the exploratory nature of reverse engineering and reengineering demands that a reengineering environment does not impose rigid sequences of activities” [DLT00] [Tic01, p. 52]. They give examples of how this requirement can be met, for instance, with the rapid presentation of source code in both textual and graphical views, allowing actions on views such as switching between different abstraction levels, and easy querying of entities. Brown gives several design considerations for program understanding tools, among them “flexibility of end-user navigation” [Bro91]. His CodeNavigator tool “has a free-form navigation style that allows the user to control the scope and flow of investigations.” Similarly, Froehlich and Dourish say about their tool, “rather than encoding specific workflows, we provide a visual tool that allows developers to *explore views* of their system” (emphasis added) [FD04].

lightweight: Reverse engineering tools should be lightweight so as to minimize the time and effort for the user to get a first useful result. The initial result can then be iteratively refined with rapid feedback. Thus, lightweight also means that tools should be “easily repairable” [Won99, p. 63]. Such a lightweight tool also facilitates just-in-time comprehension (cf. Section 2.1).

In his dissertation, Wong proposes a lightweight tool, the Reverse Engineering Notebook; he states, “the Notebook architecture is centered on an extensible, general, lightweight tool” [Won99, p. 48]. His requirement for a lightweight tool is motivated by a past tool-building experience, RevEngE [MSW⁺94], whose “successes included verifying the need for lightweight technologies” [Won99, p. 72]. Wong also proposes to use a “lightweight [exchange] format” [Won99, Data Requirement 12]. RSF is an example of such a format; for instance, it can be easily understood and edited by humans, it is easy to search, and it does not require an explicit schema.

Most researchers address the lightweight requirement in the context of fact extraction. Simple lightweight tools such as `grep` and scripting languages are popular among reverse engineers, presumably because they are widely available, easy to master,⁷⁰ and can be quickly applied to any text-based source [TW00] [MJS⁺00] [Mar03] [HT02]. Hunt and Thomas report on the results of an OOPSLA workshop that explored lightweight techniques for program comprehension [HT02]; among other techniques, practitioners named “scripting languages for ad hoc programs to build static reports.” To reverse engineer the architecture of an industrial application, Riva uses several Perl scripts for analyzing the C source files (e.g., extracting

⁷⁰Compared to relational and Turing complete queries, lexical approaches trade lower specification time for less expressive power [PP96]. Since lexical patterns are less powerful they are typically also easy to specify [BTMG02].

information about function calls and file system structure) [Riv00a]. In a similar case study, O'Brien and Tamarree reconstruct the architecture of a J2EE application [OT03]. First, the Understand for Java tool is used to generate a number of report files, which are then translated with the help of Perl scripts into RSF for manipulation with the ARMIN graph visualization tool. Van Deursen et al. note that “often a pragmatic mix-and-match approach for data gathering is applied, combining the results from various extraction tools using scripting and glueing, for example, based on UNIX utilities such as `join`, `split`, `awk` and `perl`” [vDHK⁺04]. Lightweight Lexical Source Model Extraction (LSME) is an approach to generate fact extractors from lexical specifications [MN96]. The authors state that “the approach is lightweight in that the specifications are relatively small and easy to write” [MN96]. A call graph extractor for C can be specified in less than 25 lines. In contrast, “the effort required to produce a parser-based extractor for a desired source model, however, is typically high” [MN96]. Similarly, Moonen states that “a common approach for achieving lightweightsness is based on the use of lexical analysis. ... it often takes little time to develop solutions based on lexical tooling” [Moo02b].

multi-language support: Even though there are few reverse engineering tools that explicitly target systems composed of multiple (programming) languages,⁷¹ this can be a critical requirement for most industrial software, because “most of these real world software systems consist of several—sometimes many—languages. This especially holds if a system is rather old, surviving through generations of programming languages and hardware architectures” [KWDE98]. According to Lethbridge and Anquetil, a key requirement for a software exploration tool is to “process source code in a variety of programming languages” [LA97, Requirement NF3]. Consequently, the exchange format of a reverse engineering tool “should be capable of handling systems composed of multiple source languages” [Let98]. One approach to effectively support multiple languages is to integrate them into a common schema. The GUPRO project presents such an approach to program comprehension in multi-language systems [KWDE98] [Kam98]. The multi-language conceptual model (e.g., encompassing JVM-JCL, CSP, and Cobol) is composed of multiple single language models.⁷² The single language models are integrated into a multi-language model by means of generalization of similar concepts and identification of interrelations. There are a few other examples of reverse engineering tools that support multiple languages (e.g., FORTRESS [GPB04], Moise et al. [MW05] [MWHH06], MT [LCBO03], GRASP [HHBM97], and PolyCARE [Lin95]). Synytsky et al. describe how island parsing can be used to effectively parse multi-lingual documents in the Web domain [SCD03].

⁷¹Multi-language systems are also referred to as polylingual systems [GPB04] [GBP04].

⁷²These models are all coarse grained. Kamp gives the following rationalization: “Abstract syntax trees ... contain the whole information needed for any kind of static analysis but tend to become difficult to handle in multi language environments because of their huge number of entity and relationship types” [Kam98].

There are approaches that are multi-language in the sense that they are language-independent (e.g., textual clone analysis or analyses that work on any object-oriented language [TDD00a]). For example, one requirement of the FAMIX schema is to “support multiple object-oriented implementation languages. It must abstract from those languages to allow tools to be used without adaptation for the different supported languages.” [Tic01, sec. 4.1]. For the Maintainer’s Assistant, Yang requires that “it support software independently of its source programming language” [Yan91]. For database reverse engineering tools, Hainaut et al. require that “the specification model and the basic techniques offered by the tool must be DMS-independent” [HEH⁺95].

traceability: A requirement that is often implicitly assumed by tool builders is traceability of artifacts. Wong says that “traceability is needed to relate requirements, design, and implementation artifacts” [Won99, p. 12]. For example, the tool should enable a reverse engineer to trace tool artifacts such as graphical entities to the corresponding source code entities and vice versa. More generally, Hainaut et al. say that a “tool must ensure the *traceability* of the reverse engineering process” [HEH⁺95]. Storey proposes fourteen cognitive design elements that software engineering tools should have [SFM99a]. Presumably, traceability is implied by two of these (E6/7): “support the construction of multiple mental models” (e.g., using multiple textual and graphical views), and “cross-reference mental models” (e.g., via highlighting instances of the same entity in all views).

Traceability in a tool has to be supported by the underlying data model; this is expressed by one of Wong’s data requirements: “Represent the semantics of the inverse mappings and record instances of these mappings to enable traceability of software artifacts” [Won99, Data Requirement 5].

Besides these requirements, there are other tool characteristics that can be seen as requirements as well, for instance, computing platform [Til98a] [BG98] [TD04], cost (e.g., purchase price or licensing fees) [BK01] [Til98a], portability [BK01] [Dev99b], and robustness [SCD03] [SHE02].

3.2.7 Functional Requirements

The main focus of this dissertation is on quality attributes; however, while conducting my literature search, patterns of tool functionalities emerged that were repeatedly discussed by different researchers. These functionalities can be assumed to represent a consensus—or stable body of knowledge—of the discipline. This section briefly summarizes the requirements in the hope that other researchers may find them useful for their own tool research.

Tool builders have to evaluate the functional requirements of their tools carefully, because a tool can be only useful to its users if it provides the functionality that the users need to fulfill their tasks. In fact, the second-largest complaint in Lethbridge’s survey (cf. Section 3.2.2) was missing or wrong mix of tool features (15%) [LA97] [LS97]. As a

result, Lethbridge and Anquetil require tools to “incorporate all frequently-used facilities and advantages of tools that software engineers already commonly use” [LA97, Requirement NF6].

Many researchers believe that searching is an essential part of reverse engineering activities and thus should be well supported by tools:

Searching and
querying

- Storey states, “searching is one of the main activities in program understanding” [Sto98, p. 147]. Similarly, Zayour and Lethbridge say, “the primary activity in software maintenance involves searching” [ZL00]. Bellay and Gall believe that “the search facility of a reverse engineering tool is an essential part” [BG98].
- Evidence of the importance of searching is provided with a user study conducted by Storey et al., in which the authors observed that “in Rigi and SHriMP, the lack of a searching tool to find text strings in the source code definitely hindered the users” [SWM97]. Biggerstaff et al. report the following experience with the DESIRE program understand tool: “The various query facilities provide an armory of easy ways to analyze a system and are heavily used especially when porting code” [BMW93].
- In a survey conducted by Bassil and Keller, “search tools for graphical and/or textual elements (F14)” was rated as the most useful functional aspect of software visualization tools; 74% of the respondents classified it as “absolutely essential” [BK01]. Lethbridge and Singer report on a small-scale study of maintenance programmers: “After editors, the most heavily used tools as measured using automated logging were `grep`-like searching tools” [LS97].

Consequently, Lethbridge and Anquetil state the following requirement concerning searching:

“Provide search capabilities such that the user can search for, by exact name or by way of regular expression pattern-matching, any named item or group of named items that are semantically significant in the source code” [LA97, Requirement F1].

Ducasse et al. state that a reengineering tool should “provide a way to easily access and query the entities contained in a model” [DLT01]; the authors’ tool, Moose, supports querying with the Moose Finder and Moose Explorer. Effective searching (or querying) is required for both repository and visualizer. Often information from repositories can be extracted with a (formal) query language (cf. Section 2.3.1). Some tools even have a visual query interface (e.g., the BLOOM visualization tool [Rei01]). However, visualization tools often have rather limited searching capabilities that go no further than searching for the labels of graph entities.⁷³ Storey et al. have surveyed a number of visualization tools and come to the conclusion that there is need for improved querying support [SCG05].

⁷³Storey et al. report that whereas this “search on node labels was very useful” for the users of Rigi and SHriMP, there was also the problem that “some users mistakenly thought they were searching for strings in the code rather than searching for node labels in the graph” [SWM97, sec. 6.3].

Filtering of information in visualization tools can be seen as a rudimentary form of (structural) querying [SCG05]. It is an effective approach of graph visualizations to prune nodes or arcs (based on their types, or other criteria), and is supported by tools such as Rigi and SHriMP [SWM97] [SHD05]. According to Storey’s research, filtering helps to reduce disorientation effects by users [SFM99a, E14]. Maletic et al. describe requirements of visualization tools; one of these is to “filter out uninteresting items” [MMC02]. According to Riva’s experience, an important feature in Rigi is the “possibility to filter/unfilter information. This allows us to reduce the amount of visualized data and to limit our analysis” [Riv00a]. Other examples of filtering are pruning of traces [HL04] or call trees based on specific categories [vMV93].

Filtering

The research community has discussed extensively the functional and non-functional requirements for exchange formats. This discussion was started by the realization that a common exchange format would be beneficial for the whole community.⁷⁴ In the following, I briefly summarize the requirements for exchange formats reported in the literature:

Exchange format requirements

graph model: Wong recommends to “use a graph model with multiple node types, arc types, and attached attributes” [Won99, Data Requirement 2]. Examples of exchange formats that adhere to this model are RSF, TA, and GXL.

version control: One of Wong’s data requirements is to “provide version control over schemas and fact bases” [Won99, Data Requirement 8]. An exchange format does not need to provide version control, but should also not unduly complicate the use of it.

textual: An exchange format should be textual [Let98] or else ASCII-based [LLL00]. This simplifies processing and makes the format human-readable [SSK00, Requirement R12]. Riva notes that “a human readable format allows us to navigate the data with simple text editors and eventually repair corrupted files” [Riv00b].

file-based: Lethbridge recommends that “data should be stored as files” [Let98]. Riva made the experience with FAMIX that it is convenient when “all the extracted data are contained in one single file that is easy to archive and transfer” [Riv00b].

formality: An exchange format should be well-defined and formally documented to “eliminate the possibility of conflicting interpretations of the specification for encoding and decoding model data [SSK00, Requirement R5].

composability: The fact bases of an exchange format should be composable [Won99, Data Requirement 7]. For example, two RSF files can be simply appended to form a new, syntactically valid RSF file. A related requirement is that the exchange format “should be incremental, so that it is possible to add one subsystem at a time”

⁷⁴Examples of discussion forums are WoSEF (held at ICSE 2000) [SK01], a WCRE 2000 working session on exchange formats, and Dagstuhl Seminar 01041 on *Interoperability of Reengineering Tools* (held in January 2001) [Let01].

[BGH99] [HWS00]. Flat formats such as RSF are easier to compose than nested ones such as XML [DT03].

granularity: The exchange format should “work for several levels of abstraction” such as fine-grained AST-level and coarse-grained architectural-level [BGH99]. Similarly, Koschke et al. state that an intermediate representation “should support different levels of granularity from fine-grained to coarse-grained” [KGW98, Requirement R9]. Kamp says, “the repository should support the use of the *level of granularity* appropriate to the current program comprehension task” [Kam98].

neutrality: The exchange format should be neutral with respect to the stored information. For example, it should “work for several source languages” and “work for static and dynamic dependencies” [BGH99] [HWS00]. For St-Denis et al., “the neutrality requirement ensures that the model interchange format is *independent* of user-specific modeling constructs in order to allow a maximum number of model users to share model information” [SSK00, Requirement R4].

naming: Entities in the source model have to be represented in a suitable form in the exchange format. This mapping should be unambiguous (e.g., variables with the same name but in different scopes should be distinguishable from each others [Won99, p. 29]). This can be accomplished with unique (but artificial) identifiers or a unique naming scheme [DT03]. In the FAMIX model, “all the entities have a unique name that is built using precise rules” [Riv00b].

querying: Tichelaar et al. state that “a large portion of reengineering is devoted to the search for information. Therefore it should be easy to query the exchange format. Especially, processing by ‘standard’ file utilities (e.g. grep, sed) and scripting languages (such as Perl, Python) should be easy” [TDD00b].

processing: An exchange format should be easy to read, write, and process [TDD00b] [LLL00] [KGW98, Requirement R11].⁷⁵ This means, for instance, that the exchange format “should not involve complex nested structures (e.g., nested parentheses)” [Let98].

popularity: Even though not a technical issue, popularity and enthusiastic supporters are an important requirement since an exchange format has to facilitate data exchange between many and diverse (sceptical) stakeholders [SSK00, Requirement R6] [BGH99] [HWS00] [Hol00].

In the above discussion, I have omitted functional requirements that are tightly related to the stored data itself. For example, if information about source code is represented then the format “should support a mapping between entities/relations and the source code statements” [BGH99], or simply “should preserve a mapping to the original source code”

⁷⁵For instance, the DATRIX-ASG/TA format can be parsed with a Yacc grammar of ten rules [LLL00].

[KGW98, Requirement R7]. I consider such requirements as too specific for a generic exchange format. Koschke and Sim point out that there are different stakeholders for an exchange format such as tool users and tool builders [SK01]. The requirements of different stakeholders are not necessarily the same. For instance, a tool user might favor a format that is human-readable, whereas a tool builder is primarily interested in a format that can be parsed easily and efficiently.

Requirements for schemas have been mainly addressed in the context of exchange formats. Several of Wong's data requirements are targeted at schemas. According to him, a schema should "support aggregation, inheritance, hierarchy, and constraints" [Won99, Data Requirement 3]. Inheritance of schema entities is also proposed by Lethbridge [Let98] and supported by several exchange formats (e.g., TA and FAMIX). Inheritance facilitates extensibility because it allows to add new schema entities in a defined manner [DT03]. Lethbridge et al. state that "a [schema] needs to be robust as opposed to fragile. Robustness means the widest possible variety of tools can use it without the need for inconsistent variants" [LTP04, sec. 3.5]. Ferenc et al. say, "the schema should be independent of any parsing technology" [FSH⁺01] and Riva argues for a "separation between data and presentation" [Riv00b]. A lesson learned from the Moose tool is to "make your [schema] explicit" [NDG05]. Wong also says that the exchange format should "support introspection of schemas" [Won99, Data Requirement 11]. This means that a tool should be able to query the schema itself and not just the data. To facilitate introspection, a meta-schema is needed (which is either formally or informally defined). Ducasse and Tichelaar introduce meta-meta-models as an axis in their tool design space [DT03]. They state that while an existing meta-schema has the benefit that it is already pre-defined and agreed upon by all tool users, it is less flexible and constrains extensibility.⁷⁶

Schema requirements

I have found only a few general requirements for analyzers. Wong states that "analysis tools need to be incremental and work on partial systems," which leads to his requirement to "support the incremental analysis of software" [Won99, Requirement 10]. This is especially relevant if most of the information that the analysis uses does not change frequently [Rei05]. Researchers also voice the opinion that analyses should address both static and dynamic properties of software, for instance, in the domains of constraint checking [Rei02]; reverse engineering of net-centric [TD01] and object-oriented [SKM01] applications; and architecture reconstruction [GZ05]. For example, after a case study using several reverse engineering tools, Gorton and Zhu conclude that "more work needs to be done on capturing dynamic information" [GZ05]. The developers of the REforDI system relate this experience: "The dynamic aspects of a system (runtime behavior etc.) are not considered in the REforDI project. But, we found that in order to understand a system this information is as important as information on the static parts of a system" [CMW02, page 288]. Also, analyses should support a history mechanism to undo and redo its effects. This is especially important for transformations tools because the made changes may not have the desired ef-

Analyzer requirements

⁷⁶For example, Rigi's schema is constrained by the fact that it must contain a `level` arc to model hierarchical graphs. Rigi's meta-schema is constrained because it allows only nodes, arcs, and attributes with a fixed semantics.

fect. The need of an undo or history mechanism is mentioned for schema transformations of databases [HEH⁺95, p. 140], code refactorings [MT04] [Yan91], and design recovery [JWZ02].

Researchers have also identified functional requirements for visualization tools. I summarize them briefly by grouping them into the following requirements:

Visualizer
requirements

views: Visualizer for reverse engineering typically provide different views of the target software system (e.g., to satisfy the need of different stakeholders and to emphasize different dimensions of the data such as the time dimension or level of abstraction) [Kos02] [RCdBL03]. An example of integrated views is provided by Kazman’s SAAMtool, where

“any node within any view can have links to nodes in one or more of the other views. Currently the tool supports a dynamic model, a code view, and a functional decomposition. Using these views one could see, for example, not just a code view of a product, but could select elements from this code view and use the set of selected elements to limit what parts of the dynamic view are viewed” [Kaz96].

Wong requires visualizers to provide views that are consistent and integrated [Won99, Requirement 15]. He further requires to “integrate graphical and textual software views, where effective and appropriate” [Won99, Requirement 20].⁷⁷ The developers of the Rigi system advocate *dynamic views* [WTMS95]. In contrast to static views, which only provide information that has been obtained at a certain point in time, dynamic views synchronize and recompute their information when the underlying data source changes.⁷⁸ Blaine et al.’s software visualization taxonomy asks, “to what degrees can the system provide multiple synchronized views of different parts of the software being visualized?” [PBS93, C.4]. Storey says, “programming environments should provide different ways of visualizing programs. . . . orthogonal views, if easily accessible, can facilitate comprehension, especially when combined” [Sto05]. Similarly, Meyers states that “there is a crucial need for programming environments that facilitate the creation of multiple views of systems” [Mey91].

Koschke has conducted a survey about software visualization; he summarizes the responses as follows: “Most prominently, the need for multiple views in software maintenance, reverse engineering, and re-engineering is pointed out. Given multiple

⁷⁷There is the believe that graphical views of software are superior to textual ones, however this is not necessarily the case. For instance, Murphy and Notkin observed in a case study involving their reflexion model technique that “surprisingly, the engineer drove almost all the investigation of the reflexion model and the source code from textual information. Thus, it might be important to rethink the general belief that graphical interfaces to reverse- and reengineering tools are the best approach” [MN97]. Cordy’s experience is that “source code is the only real medium of understanding for programmers of the financial world (and thus graphs may not be a good choice for presenting analysis results to this community)” [Cor03].

⁷⁸In Rigi dynamic views are (re-)constructed with Tcl scripts. So-called live documents are a generalization of Rigi’s dynamic views [WKM02].

views, integration, synchronization, and navigation between these views while maintaining the mental map are important aspects” [Kos03]. Bassil and Keller’s survey of software visualization tools consider “synchronized graphical and textual browsing,” which rated 62% of the respondent as “useful, but not essential” [BK01, F7]. Hainaut et al. state the requirement that views should address different levels of granularity as well: “Multiple textual and graphical views, summary and fine-grained presentations must be available” [HEH⁺95].

abstraction: Most visualizations are based on an underlying graph model. Since the resulting graphs can become too complex to effectively visualize (cf. Section 3.2.1), even for small software systems, abstraction mechanisms are needed. Von Mayrhauser and Vans conclude from their studies that “maintenance programmers work at all levels of abstraction” [vMV93] and consequently stress the “importance of the availability of information about the program at different levels of abstraction” [vMV95a]. Hierarchical graphs are a popular abstraction mechanism [JMW⁺02]. They allow grouping of nodes to create higher levels of abstractions, resulting in a layered view [BG98]. Ducasse and Tichelaar believe that “grouping several entities together is an important technique in reengineering, primarily to build higher-level abstractions from low-level program elements or to categorize elements with a certain property” [DT03]. According to Kazman, “a tool for software architecture should be able to aggregate architectural elements recursively” [Kaz96]. Rigi is an example of a tool that fulfills this requirement. It has “support for abstraction activities (grouping/collapsing) with an easy to use GUI” [Riv00a]. Similarly, the SHriMP tool fulfills the requirement to “provide abstraction mechanism” by providing “subsystem nodes and composite arcs in the nested graph” [SFM99a, E3]. There are many more examples of tools that provide similar abstraction mechanisms (e.g., EDGE [New88] [PT90], daVinci [FW94], and Balmas’ work [Bal01] [Bal04]). In Bassil and Keller’s survey about software visualization, “hierarchical representations (of subsystems, classes, etc.)” was the third-most useful functional aspect, rated by 57% of the respondents as “absolutely essential” [BK01].⁷⁹ Lastly, abstractions are also possible in the presentation of dynamic information. For instance, tools such as ISVis allow the reverse engineer to “view the trace content at different levels of abstraction (e.g., object interactions, class interaction, etc.)” [HL04].

code proximity: Code proximity means the ability of the visualizer to provide easy and fast access to the original, underlying source code [LD03]. Most reverse engineering tools fulfill this requirement (e.g., Rigi, SHriMP, and CodeCrawler), but differ in their capabilities. For instance, Rigi supports the navigation from a node in the graph to a source file and line number, while SHriMP can visualize the source di-

⁷⁹Interestingly, in the same survey the aspect “abstraction mechanisms (e.g., display of subsystem nodes, and of composite arcs in graphs) (F12)” was ranked comparably low by participants; less than 25% saw it as “absolutely essential” [BK01].

rectly within the node. The importance of this requirement becomes apparent by the number of researchers who have commented on it, for instance:

- One of the criteria in Bellay and Gall’s reverse engineering tool evaluation addresses code proximity as follows: “Point and click movement from reports to source code allows to access the source code representation of a specific entity in a report. The browsing of the source code can be seen as switching the abstraction level and is done often during code comprehension” [BG98].
- Ducasse et al. state, “to minimize the distance between the representation of an entity and the actual entity in the source code, an environment should provide every entity with a direct linkage to its source code” [DLT01].
- Wong says that “links between the source code and the repository are needed” [Won99, p. 16].
- Bassil and Keller’s survey did ask participants for the importance of having “easy access, from the symbol list, to the corresponding source code” [BK01, F5]; more than 35% of the participants ranked this as “absolutely essential.”
- In Koschke’s survey, respondents pointed out that “the connection between source code views and more abstract views derived from these needs to be maintained in both directions” [Kos03].
- The TkSee search tool provides search results of different kinds (e.g., files, functions, variables, and lines in a file). For each kind of search result, the tool user has “instant access to the source code” [LH01, p. 78].

Code proximity is just a special case of a certain abstraction mapping. There are other mappings (e.g., between the application and implementation domains), which are equally important but less well explored by researchers [MJS⁺00].

automatic layouts: Many visualization are less effective or practically impossible to construct without automatic layouts. Even if automatic layouts cannot be perfect, they provide a good starting point for further, manual refinement. Researchers have developed layout algorithms to present graphs in more meaningful ways [SFM99b]. For instance, Riva states about the Rigi tool that it has “support for automatic layout of graphs according to different algorithms. This allows us to look at the data from different perspectives” [Riv00a]. Different layout approaches can represent the same data quite differently (and influence their interpretation [Pet95]). For instance, hierarchical graphs can be visualized in a single window as a nested view (e.g., SHriMP (cf. Figure 6)), as a tree view (e.g., Rigi), or in separate views showing a single nesting only (e.g., Rigi) [SFM99b]. There are also 3D layouts such as Reiss’ Plum framework [Rei95a].

Layouts often strive to optimize certain properties of the graph [ME02] [TBB88]. Bellay and Gall say, “special layout algorithms can make a graph more readable;

for example, minimizing the crossings of the arcs” [BG98]. The SHriMP layout adjustment algorithms allows to apply a layout without adversely affecting the layout of the original graph [SFM99b].

Bassil and Keller’s survey asked for the importance of “automatic layout capabilities” as one of the functional aspects of visualization tools. About 40% of the survey’s respondents rated this as absolutely essential [BK01, F10]. In Koschke’s survey, 78% of the users who visualize graphs are using automated layouts. This seems to support the claim that automatic layouts is an important requirement. However, respondents complained about deficiencies such as “lack of incremental, semantic, and scalable layouts, bad space consumption, and lack of user control” [Kos03]. Koschke also raises the issue that layout algorithms should take semantics of nodes and edges into account (e.g., in a UML class model one would expect that all inheritance links point in the same direction) [Kos02]. Koschke recommends that the reverse engineering community should collaborate more actively with graph drawing researchers [Kos03].

undo: Since users perform interactive manipulations that change the visualization, there should be an undo mechanism that allows them to revert to previous states. Ducasse et al. name a history mechanism, which should contain all previously performed steps of graph operations [DLT01]. In the LSEdit graph editor, “any edit that can be done can be also undone. Any edit that has been undone can also be re-done” [SHD05]. Maletic et al. say visualizers should “keep a history of actions to support undo, replay, and progressive refinement” [MMC02]. Bassil and Keller’s survey of software visualization tools consider both the “possibility to record the step of an arbitrary navigation” and the “possibility to return back to proceeding navigation steps” [BK01, F22 and F23]. Based on their research in program comprehension, Mayrhauser and Vans believe that a tool should have a “history of browsed locations” [vMV93].

miscellaneous: Other requirements mentioned by researchers are the use of colors [PBS93, C.2.1.1] [BK01, F32], the ability to place annotations on graphical entities [SCG05] [BG98] [SFM99a] or code [vMV93], and to manipulate graphs, specifically zooming [BK01, F11] [DLT01] [SCG05] [MMC02], panning [SFM99b], deleting/editing of view entities [DLT01] [BG98], and saving [BK01, F24] [MMC02] [SFM99a]. Researchers also see a potential benefit of using animations for information visualization [Kos02] [BK01, F33] [Won99, Requirement 22] [PBS93, C.2.2].

It is not surprising that many of the identified features overlap with general requirements for graph editors. Karrer and Scacchi state twelve requirements for an extensible tree/graph editor; many of these directly apply to the domain of software visualization [KS90].

3.2.8 Discussion

A number of observations can be made based on the conducted survey in this chapter.

Requirements are often discussed without citing related work,⁸⁰ or mentioned without giving a detailed explanation or rationalization. Particularly, it is often not clear where requirements come from—have they been derived based on observation of users (e.g., shadowing or monitoring [SL96]), questioning of users (e.g., surveys), experiences of the developers, personal intuition, et cetera? Identified requirements are useful, regardless whether they originated in empirical research, or practical experience and intuition [Sto05, sec. 3.3]. However, researchers should nevertheless clearly state how the requirements originated.

Requirements elicitation is ad hoc

Researchers do not discuss the applicable scope of a stated requirement. For instance, is the requirement believed to apply to all software systems, to the domain of software or reverse engineering, to certain kinds of tools, or to one tool in particular?

Requirements are not scoped

A requirement is often discussed in isolation, without addressing dependencies or trade-offs with other requirements. Approaches such as Mylopoulos et al.'s NFR-Framework could be used to make such dependencies more explicit [MCN92] [CNY95]. Perhaps surprisingly, only Lethbridge and Anquetil explicitly separate their requirements into functional and non-functional ones [LA97]. Such a distinction makes it easier to judge the scope and applicability of requirements. Furthermore, they explicitly identify requirements that their tool does not address (yet), thus making their tool's limitations more explicit.⁸¹

Requirements are unstructured

Most of the requirements in the survey are discussed at an abstract level. Whereas such requirements are useful, they do not allow to measure objectively to what extent a tool fulfills a requirement, or to compare tools quantitatively. This is understandable for a requirement such as usability (cf. Section 3.2.4), which, according to Brown's classification (cf. Section 3.2), is a *timeless requirement*. An open research problem for such requirements is it to "operationalize these requirements into concrete, measurable criteria" [Bro94]. However, other requirements such as scalability are easier to quantify. Researchers should take this opportunity to precisely state the metric and the experimental data that makes them believe that their tool is, say, scalable.

Requirements are ill-defined

Whereas tool requirements seem relatively stable, they are not fixed. Changes in development and maintenance processes and in the characteristics of software need to be reflected in the requirements for reverse engineering tools. Thus, researchers should continuously reevaluate their assumptions. For example, a previously neglected requirement that is starting to receive more attention by researchers is multi-user support. Bellay and Gall argue that multi-user support in reverse engineering tools is "not of such importance as in development tools because the application normally does not change and only one person may reverse engineer it" [BG98]. However, this view seems dated considering the large amount of commercial and open source software that is developed and maintained in a distributed, collaborative work-style. Koschke states,

Requirements can change

⁸⁰There are a few notable exceptions that adequately cite related work (e.g., [DT03] [BTMG02]).

⁸¹For instance, they state, "the system is not required, at the present time, to handle object oriented source code" [LA97, Acceptable Limitation L2].

“large maintenance and reverse engineering projects require team-work and, hence, visualizations need to support multiple users that may work at the same system at the same time at possibly different locations” [Kos02].

The emerging importance of this requirement is also reflected in Storey et al.’s software visualizer framework, which has a dimension to distinguish the team size that a particular tool targets [SCG05].

I believe that my survey suggests that tool requirements is a worthwhile research topic that still needs further exploration. Even though it is by now well understood that a thorough requirements elicitation should be conducted before starting to implement a system, it appears that reverse engineering tools are often built without a clear understanding of neither the general requirements of the reverse engineering domain, nor the specific requirements of a particular kind of tool.

3.2.9 Contributions

The requirements exposed by my survey can contribute to advance tool-building in several ways. This section briefly enumerates these contributions.

During tool development, the survey can simplify the elicitation of requirements. For example, as suggested by Kruchten, a list of requirements can be used as a checklist to make sure that no important requirements are missed, thus increasing the confidence of the developers [Kru99, p. 139]. In fact, according to Boehm, making developers “aware of quality concerns, that by itself helps improve overall software quality” (qtd. in [CNY95]). Wong states that his 23 “high-level requirements help to characterize the fundamental sub-problems and issues that need to be effectively addressed in building [reverse engineering] tools” [Won99, p. 6]. My survey is more comprehensive and points out several shortcomings that could be addressed by the next generation of reverse engineering tools. Lastly, requirements can be also an important driver when deciding on the tool’s architecture (cf. Section 2.3.5) and development process.

Tool development

For tool evaluations, the survey can be used to pick suitable requirements for measuring and comparing tools. An evaluation can pick and informally discuss a set of accepted requirements to convince fellow researchers of a tool’s adequacy or superiority compared to another tool—Shaw refers to this research approach as narrative evaluation based on a qualitative model (“Given the stated criteria, my result . . . accounts for the phenomena of interest. . .”) [Sha03]. More formal tool evaluation approaches such as benchmarks can pick a set of requirements and employ a *task sample* (i.e., a representative sample of the tasks that a class of tools or techniques is expected to solve in actual practice) [Sim03]. For example, Sim has developed a benchmark that measures the accuracy and robustness of C++ extractors [SHE02].⁸² Similarly, the scalability of tools could be measured by

Tool evaluation

⁸²Rugaber and Wills already proposed benchmarks in 1996: “Coordinated and measurable progress in our field depends critically on establishing standard data sets and benchmarking tasks that can be used to quantitatively evaluate and compare reengineering tools and techniques” [RW96]. It has taken a considerable time for the first benchmark proposals to emerge (e.g., C++ extractors in 2001 [SHE02], and clone detection in 2003 [LLWY03]).

applying a certain task (e.g., construction of a call graph) to certain subject systems of varying sizes and complexities.

Lastly, (idealized) requirements can drive tool research. For example, Wong states that addressing all of this 23 requirements represents “a significant research challenge” [Won99, p. 20], and recommends to “summarize and distill lessons learned from reverse engineering experience to derive requirements for the next generation of tools” [Won99, Recommendation 3]. Similarly, Hamou-Lhadj et al. have identified requirements for trace exploration tools and in doing so “have uncovered a number of requirements that raise very interesting research challenges” [HLF04]. Storey et al. generally believe that

Tool research

“surveys in any research discipline play an important role as they help researchers synthesize key research results and expose new areas for research” [SCG05, page 200].

The survey in his chapter reflects the current state of tool requirements and thus can serve to identify open and new requirements for future research directions.

3.3 Requirements for Tool Development Processes

This section introduces requirements for a software process⁸³ to develop reverse engineering tools. Tool development in an academic research environment often is ad hoc and unstructured.⁸⁴ As a result, tools are developed without following a well-defined process. However, researchers have reported some experiences that allow to distill properties that an appropriate development process should probably possess. I also draw from informal discussions with other researchers, and from my own experiences of developing reverse engineering tools as a Ph.D. student both at the Bauhaus group (University of Stuttgart) and the Rigi group (University of Victoria).

The identified requirements can, on the one hand, provide important background information and constraints for developing an appropriate tool-building method, and, on the other hand, serve as evaluation criteria to judge the efficacy of a proposed tool-building method.

Benefits

The development history of the SHriMP tool is one of the few more comprehensive sources that allow me to infer requirements for tool-building. The SHriMP project, which has spun out of Rigi, is an example of building a tool that has been continuously refined and evaluated, following an iterative approach of designing and implementing reverse engineering tools, which has been proposed by the team’s leading professor [Sto98, sec. 11.1.1]

SHriMP

⁸³Software processes consist of a set of tools, methods and practices that are used to produce a software product [LY01]. In other words, a process is the “sum of all software engineering activities encompassing the whole software life cycle” [CCM05].

⁸⁴Tools are often constructed by students that have only a few years of programming experience. They typically work alone or in small teams with informal communication flow and without an explicit process. Furthermore, they often work not closely supervised and are evaluated based on their finished product, not on how they have constructed it.

[SFM99a].⁸⁵ Evolving a tool such as SHriMP is a major research commitment, involving several students at the Ph.D. and master level at any given time. SHriMP had several major implementations.⁸⁶ The first implementation was based on Rigi (with Tcl/Tk and C/C++) [SFM99a], followed by a Java port [WS00]. The Java version of SHriMP was then rearchitected in a component-based technology, JavaBeans, to facilitate closer collaboration between reverse engineering research groups [BSM02] [SBM01]. As a result, SHriMP can be used as a stand-alone tool, integrated within the Eclipse framework (Creole and Xia), or integrated within the Protégé knowledge-management tool (Jambalaya) [WMSL04] [LMSW03] [BSM02] [RLSB01] [MSM01]. Wu and Storey have published the results of their first Java port of SHriMP; they state that “during the development of this prototype, we took into consideration the knowledge from previous prototypes and empirical evaluations” [WS00]. Even though they do not provide further details, we can infer that their development process is (1) prototype-based, (2) iterative, and (3) based on feedback from empirical evaluations.

The TkSee search tool is an example of a tool that has been improved based on the feedback obtained from developers in industry. TkSee is written in Tcl/Tk and C++. Early versions of TkSee were delivered to users at Mitel in 1996. At the time, the tool was used by relatively few user, which used only a small subset of its features. Generally, users were reluctant to adopt new tools.⁸⁷ Lethbridge and Herrera describe TkSee’s development process as follows [LH01, p. 89]:

TkSee

“TkSee had been developed in a university research environment following an informal and opportunistic development process. Features had been added by students and researchers when they had had bright ideas they wished to experiment with, hence it lacked complete documents describing its requirements, its design (except that of its database architecture [LA97]) and how to use it.

There was considerable staff turnover among TkSee developers because many were students. Also, almost none of the staff had any training in user interface design and usability. The newer staff was often not able to understand the tool or the purpose of certain features. They did not appreciate why certain user interface decisions had been made, or even that certain decisions were deliberate, hence, they tended to make changes that were poor from a UI perspective and that led to usability problems.”

To improve TkSee, feedback was obtained from field observations as well as formal user studies using think-aloud protocol and videotaping. Communicating the identified problems to the tool developers was carefully planned [LH01, p. 88]:

⁸⁵During its initial development, SHriMP could take advantage of the experiences and lessons learned with Rigi. This knowledge transfer was possible because SHriMP’s leading professor was involved in Rigi’s development as a Ph.D. student. (Personal communication with Hausi Müller in April 2005.)

⁸⁶Personal email communication with Casey Best of the Chisel group in October 2004.

⁸⁷Many of the Mitel developers “have not even adopted Emacs, and prefer to use more primitive editors they know better” [SL98].

“We prepared a report that summarized and categorized the problems and walked through this with the developers - showing them video clips of the problems to emphasize certain points and convince them of the seriousness. Following this, the developers implemented most of the recommended changes.”

Similar to SHriMP, we can conclude that tool development had at least one iteration that improved the tool based on feedback from a user study. Furthermore, the use of Tcl/Tk would facilitate rapid prototyping of TkSee.

Unfortunately, most researchers do not report at all about their tool development process. It seems that researchers often develop their tools by themselves and are the only users of the tool during and after development. Tools are then evaluated with a case study or anecdotal evidence.

The following sections discuss the requirements that I have been able to identify for a tool-building process in an academic research setting.

3.3.1 Feedback-Based

“As soon as the end users see how their intentions have been translated into a system, the requirements will change. ... Users don’t really know what they want, but they know what they do not want when they see it.”

– Philippe Kruchten [Kru99, p. 54]

Many ideas for improvements of software systems originate from their users. There is evidence that software development projects are the more successful, the more *user-developer links* they use [KC95]. These links are defined as the techniques and/or channels that allow users and developers to exchange information; examples are surveys, user-interface prototypes, requirements prototypes, interviews, bulletin boards, usability labs, and observational studies.

As with many other software development projects, often the researchers developing a research tool have a poor initial understanding of its requirements. Requirements elicitation is difficult, for instance, because the target users are ill-defined and the tool’s functionality might be not fully understood yet. To alleviate this problem and to bootstrap the first release, Singer and Lethbridge propose to first involve reverse engineers (who will later use the tool) to understand their particular needs and problem domain before starting the design and implementation of the reverse engineering tool [SLVA97]. This can be achieved with questionnaires, (structured) interviews, observation sessions, (automated) logging of tool use, and so on [LS97]. For instance, Lethbridge and Singer have used a simple, informal approach to elicit initial tool requirements from future users [SL98]:

Feedback from target users

“For the first release, we brainstormed a group of software engineers for their needs, and then designed, with their continued involvement, a tool called SEE (Software Exploration Environment).”

Once a research tool has reached a first stable release, feedback should be obtained. Lethbridge and Singer follow a process with two main phases: (1) study a significant number of target users to thoroughly understand the nature of their work, and then (2) develop and evaluate tools to help the target users work better. Importantly, the second phase involves “ongoing involvement with [target users]” [LS96]. User feedback can be obtained, for instance, with (longitudinal) work practice studies of individuals or groups [SLVA97] [SL98] [vMV95a] [Wal03b]. Such studies observe and record activities of reverse engineers in their normal work environment, working on real tasks. The designers of the Augur software exploration tool, for instance, report that “to gain some initial feedback on Augur’s effectiveness, we have conducted informal evaluations with developers engaged in active development” [FD04]. Hundhausen and Douglas have used the finding of an ethnographic study with students in a course (involving techniques such as participant observation, semi-structured interviews, and videotape analysis) to redefine the requirements for their algorithm visualization tool [HD02].

Wong states that “case studies, user experiments, and surveys are all necessary to assess the effectiveness of a reverse engineering tool” [Won99, p. 93]. Many researchers conduct informal case studies of their tools by themselves,⁸⁸ considering themselves as typical users [TW98]. However, it is not clear whether this approach can generate the necessary feedback to further improve a tool. A more effective approach is user studies. In the best case, feedback is provided by the actual or potential future users of the tool; however, obtaining feedback from this type of user is often impossible. As an alternative, researchers use other—presumably “similar”—subjects such as computer science students. For example, Storey conducted a study with 12 students to assess a new visualization technique introduced with the SHriMP tool. Observations and user comments resulting from the study generated several improvements [SWF⁺96, sec. 5.3]. Storey explains her strategy as follows:

Feedback through
user studies

“We are currently planning further user studies to evaluate the SHriMP and Rigi interfaces. Observations from these studies will be used to refine and strengthen the framework of cognitive design issues which will, in turn, be used to improve subsequent redesigns of the SHriMP interface” [SFM99a].

In another user study involving students, researchers did ask 13 questions about their visualization tool, sv3D, with the goal “to gather feedback from first time users about some of the sv3D features” [MCS05]. They conclude that the users’ “answers provided us with valuable information that supports the implementation of new features in sv3D.”

To summarize, researchers have tried to obtain feedback from different types of subjects (e.g., professionals who will use the tool, students who substitute for “real” users of the tool, and, last but not least, themselves) as well as with different methods (e.g., case studies, field studies, surveys, and formal experiments [KPP⁺02] [ZW98] [Bas96]).

⁸⁸In a survey of twelve visualization tools, two were evaluated with user studies and eleven with a case study [SCG05].

3.3.2 Iterative

“You should use iterative development only on projects that you want to succeed.”

– Martin Fowler, qtd. in [Lar04, p. 17]

Iterative software development processes—as opposed to processes that follow a waterfall or sequential strategy—are an approach to building software in which the overall life-cycle is composed of a sequence of iterations. Boehm’s spiral model [Boe88], the Rational Unified Process (RUP) [Kru99], and Extreme Programming [Bec00] are examples of iterative software development processes.⁸⁹ Developing software iteratively is among the six best practices identified by Kruchten [Kru99, p. 6]. In fact, many processes are iterative—the processes of reverse engineering software provide several examples (e.g., iterative hypotheses refinement in program comprehension [vMV95b], iterative domain modeling in knowledge organization [TS96, 3.2.2], and iterative extraction of facts [KS03a]).

Sequential vs.
iterative development

The waterfall model provides minimal opportunity for prototyping and iterative design [Gru91], but “is fine for small projects that have few risks and use a well-known technology and domain, but it cannot be stretched to fit projects that are long or involve a high degree of novelty or risk” [Kru99, p. 75]. As illustrated by Rigi and SHriMP, tool-building projects can run for several years. Since research tools typically explore new ideas and strive to advance the state-of-the-art, they are risky, potentially involving new algorithms and techniques. Thus, the waterfall approach is not suitable for the development of reverse engineering research tools.

Iterative software development has several benefits compared to the waterfall model [Kru99, p. 8]; especially important benefits in the context of tool building are:

Benefits

risk mitigation: Misconceptions are made evident early in the life cycle when it is comparably cheap to react to them.

user feedback: User feedback is encouraged, making it possible to elicit the system’s real requirements.

process improvement: Software developers can incorporate made experiences and lessons learned into the next iteration, improving their development approach.

executable releases: Each iteration results in an executable release that can be evaluated.

Iterative development is most suitable to cope with requirements changes [McK99] [Kru99, p. 74] [LN04]. As in other software projects, requirements in tool-building can change frequently (e.g., because of user-feedback⁹⁰ and modified hypothesis).

⁸⁹Even though the importance of iterative development has been recognized only relatively recently, it was already used in the early 1960s for software construction [LB03]. According to a survey of software practitioners, one third of projects still use the waterfall approach [LN04].

⁹⁰Schrage says that “the first client demo renders 40 percent of the listed requirements either irrelevant or obsolete—the problem is, we don’t know which 40 percent” [Sch04].

A particular example of the iterative nature of tool development is provided by the development of a schema and corresponding fact extractor. A schema embodies a certain domain (e.g., C++ code or software architecture), which has to be interactively refined. Changes in the schema trigger changes in the fact extractor:

“In reverse engineering a software system, one early step involves defining a schema and extracting the artifacts from the source code. These two tasks are mutually dependent and are iterative. As the source code is better understood, more facts are extracted, and the schema evolves to accommodate the new information” [MW04b].

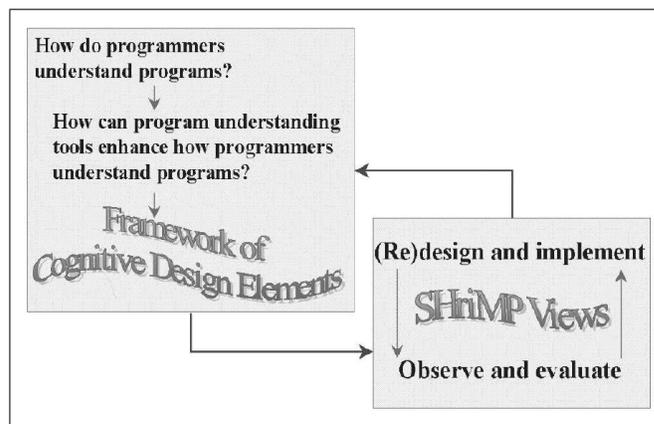


Figure 8: Storey’s iterative tool-building process ([Sto98])

As an outcome of the SHriMP tool, Storey proposes an iterative approach for designing and implementing reverse engineering tools [Sto98] [SFM99a]. It is a process consisting of “several iterative phases of design, development and evaluation” [Sto98, sec. 11.1.1]. Figure 8 depicts a rendering of the process iterations, illustrated with SHriMP as the subject. There is an “iterative cycle of design and test” that aims at improving a tool [SFM99a]. The (initial) design of the tool is guided by the cognitive design elements framework, which provides a catalog of issues that should be addressed by software exploration tools. Testing of tool design and implementation can be accomplished by user studies. This cycle is embedded in a larger iterative cycle for improving the framework. When adopting Storey’s process, the larger cycle could be omitted or replaced with a different framework.

Storey’s process

Similar to Storey, Wong argues for an iterative approach when building reverse engineering tools. He explains the process with his proposed Reverse Engineering Notebook:

Wong’s process

“The Notebook research itself undergoes continuous evolution. The evolution follows a spiral model . . . Each iteration of the Notebook involves several states, including studies with users, evaluating alternative technologies, considering compatibility constraints and adoption risks, validating the product, and planning for the next phase” [Won99, p. 99].

3.3.3 Prototype-Based

“We prototyped our application in Visual Basic, expecting to rewrite it in C,’ said a friend of mine recently. ‘But in the end we just shipped the prototype.’”
 – Jon Udell [Ude94]

The construction of prototypes is common in engineering fields (e.g., scaled-down models in civil engineering) [AH93]. For software construction, prototyping is the “development of a preliminary version of a software system in order to allow certain aspects of that system to be investigated” [Sch98]. The prototype typically lacks functionality and does not meet all non-functional requirements. Kail and Carmel distinguish between user-interface prototyping (which exposes users to a demo or early version of the system to uncover user-interface issues), and requirements prototyping (which exposes users to a demo or early version of the system to discover system requirements) [KC95]. Rapid prototyping is the use of tools or environments that support prototype construction [AH93]. Examples of technologies that facilitate rapid prototyping are weakly-typed (scripting) languages with support for GUI construction (e.g., Tcl/Tk [Ous98] [Sch98] and Smalltalk [Sch96] [GB95] [AH93]), tools for rapid application development (e.g., IBM’s Lotus Notes and Magic SME’s Magic [Zub97]), and domain-specific prototyping environments and languages (e.g., Matlab [Bro04] [EB01] and the Computer Aided Prototyping System (CAPS) [LSP⁺03] [LS92] [AH93]).

A throwaway prototype produces a cheap and rough version of the envisioned system early on in the project [AH93].⁹¹ This prototype is useful to obtain and validate user requirements and is then thrown away once the initial requirements have been established. Exploratory prototypes are typically throwaway prototypes that are “designed to be like a small experiment to test a key assumption involving functionality or technology or both” [Kru99, p. 161]. As opposed to the construction of a throwaway prototype, an evolutionary approach to prototyping continuously evolves the prototype into the final product. An example of a particular approach to evolutionary prototyping is the building of a skeletal system [BCK98, sec. 13.2]. This approach first defines the system’s architecture and then implements each component of the architecture to the extent that it demonstrates its behavior and can be executed; initially most parts of the system are not fully functional (or “stubbed out”); this can be realized via a “canned” sequence of actions. Subsequent development selectively replaces stubs with real code.

Throwaway vs.
evolutionary
prototype

Using prototypes in software development has a number of benefits. They are effective for defect and risk avoidance, and for uncovering potential problems (e.g., verification of the proposed system architecture, trying out a new technology or algorithm, identification of performance bottlenecks, or exploration of alternative designs) [Boe99] [LSP⁺03]. Yang has used rapid prototyping to develop a reverse engineering tool called the Maintainer’s Assistant; he describes several benefits:

Benefits and
drawbacks

“There are several advantages of rapid prototyping which can be taken to de-

⁹¹A throwaway prototype need not be executable—it can be a paper simulation or storyboard [Spa00] [Cam02].

velop the Maintainer's Assistant itself. For instance, the system can be developed much faster by rapid prototyping, so that it can speed up the implementation. The user is involved in the process, so that he or she (mainly the builder in our system) can determine if the development is moving in the right direction. Rapid prototyping produces a model of the final system, so that the user, as well as builder, can see the final system in the early development stage" [Yan91].

Prototyping has potential drawbacks also [AH93]. In contrast to a description of a system's behavior, executable prototypes are well suited to represent what a system does. However, they are less well suited to capture design rationale (i.e., why was the system built and why in this way)—this should be documented explicitly [Sch96]. There is also the threat that a prototype that was originally planned to be thrown away is evolved into the final product [GB95]. Evolutionary prototyping is then "merely the official name given for poor development practices where initial attempts at development are kept rather than thrown away and restarted" [LN04].

Prototypes are well suited for applied research—Chirouze et al. go as far as stating that "the idea of rapid prototyping and researching is intrinsically linked" [CCM05]. Wong proposes an iterative development strategy for the building of reverse engineering tools. In this approach,

Reverse engineering
tool-building

"each iteration builds an evolutionary prototype that is intended to be a solid foundation for the next iteration" [Won99, p. 99].

Prototypes have been successfully used in the building of software engineering tools (e.g., using Visual Basic [HNHR00], and Tcl/Tk [SWF⁺96]). A tool prototype can serve as an initial proof-of-concept. Such a prototype does not need to fulfill certain requirements. For instance, a prototype can demonstrate the usefulness of a novel algorithm without meeting the scalability requirements of a production tool. However, if user studies are based on the prototype, it might have to meet much higher quality standards. For example, Storey et al. report that

"The first prototype of the SHriMP interface was implemented in Tcl/Tk. Tcl/Tk is a scripting language and user interface library useful for rapidly prototyping graphical interfaces. However, its graphics capabilities are not optimized for efficiently displaying the large graphs typical of software systems. The second prototype has been implemented using Pad++, a graphics extension for Tcl/Tk. Pad++ is highly optimized for efficiently displaying large numbers of objects and smoothly animating the motions of panning and zooming" [SFM99a].

3.3.4 Other Requirements

Based on my experiences, I believe that the following requirements should be considered for a tool-building process as well:

lightweight: The notion of a lightweight process is not clearly defined. However, evidence of a lightweight process is that it strives to minimize (intermediate) artifacts such as vision statement, use-case model, and status assessment, recognizing that these artifacts are not the goal of a process, but a means to an end. Martin says, “a good process will decrease the demand for such intermediate artifacts to the barest possible minimum” [Mar01]. Extreme Programming has four core values from which twelve practices are derived; one of these values is simplicity [Bec00]. In this context, Martin says, “A process that is too complex will fail. . . . Anything that cannot be completely justified, is eliminated. A process description should always look too small” [Mar01]. Similarly, the SEI’s Personal Software Process Body of Knowledge states that “a process description should be brief and succinct” [PMCS05, Key Concept 1.1.1]. A goal of configuring RUP is that the process “must be made as lean as possible while still fulfilling its mission to rapidly produce predictably high-quality software” [Kru99, p. 31]. Cockburn introduces two factors that determine a method’s weight: *method size* (i.e., “number of control elements, including deliverables, standards, activities, milestones, quality measures, and so on”) and *method density* (i.e., “the detail and consistency required in the elements”) [Coc00].⁹² He says, “a relatively small increase in methodology size or density adds a relatively large amount to the project cost” [Coc00]. Agile methods are typically more lightweight than plan-driven approaches [BT03] [ASRW02].

adaptive:⁹³ A process should be flexible enough to accommodate changing requirements of the system under construction. However, evolving requirements—or other changes in business, customer, or technological needs—might make it necessary to adapt the process itself during the development effort. The SEI’s Personal Software Process Body of Knowledge states that “a quality process must be easy to learn and *adapt to new circumstances*” (emphasis added) [PMCS05, Key Concept 5.1.4]. Fowler describes the adaptive nature of a process as follows [Fow03]:

“A project that begins using an adaptive process won’t have the same process a year later. Over time, the team will find what works for them, and alter the process to fit.”

A process can be adapted after an iteration as the result of a process review.⁹⁴

product quality attributes: Processes can specifically target certain quality attributes of the final system. For example, user-centered design focuses on usability. Göransson et al. believe that “for the successful integration of usability it must be incorporated in

⁹²In later work, Cockburn has renamed density to ceremony [Coc03, p. 42].

⁹³Note that this dissertation distinguishes between adaptive and configurable processes. A configurable process makes it possible to make certain changes to a process before development starts; an adaptive process allows ongoing changes during development. Abrahamsson et al. distinguish between processes that are universally predefined (i.e., fixed) vs. situation appropriate (i.e., adaptive) [AWSR03].

⁹⁴How to change and evolve processes is also addressed by the field of method engineering [RTR⁺00].

the software development process” [GLG03]. Instead of proposing their own unique usability method, they extend RUP with a new usability design workflow. A similar case as for usability can be made for another quality attribute—adoptability (cf. Section 3.2.5).

Even though I could not find explicit evidence for these process requirements in the reverse engineering literature, I believe that they should be part of an effective tool-building approach.

3.3.5 Discussion

“If you believe that one size [solution] fits all, you are living in a pantyhose commercial.”

– P.J. Plauger, qtd. in [GV95, p. 64]

Many reverse engineering tools have been built by researchers and many issues have been explored. Requirements for tools have been identified and processes for using the tools have been proposed. For example, for the Reverse Engineering Notebook, Wong discusses tool requirements, identifies user roles (i.e., a builder to construct the Notebook architecture, a mediator to populate the Notebook with relevant information, and an end user accessing the Notebook),⁹⁵ and gives relevant tasks for the different roles (e.g., a builder integrates analyses and visualizations, a mediator extracts software facts from the target system, and an end user explores the target system with analyses and views).

Researchers have also started to think about how reverse engineering tools should be built—but examples are scant. Lethbridge and Singer focus on requirements elicitation from future users of the system (cf. Section 3.3.1). Storey proposes an iterative cycle of tool design and tool evaluation (cf. Section 3.3.2). Yang advocates to use rapid prototyping (cf. Section 3.3.3). Most of these ideas are incomplete and lack necessary details. For example, Storey and Wong have proposed an interactive approach to tool-building, but they do not further elaborate their approach into a full process—for that it would be necessary to describe *who* is doing *what*, *how*, and *when*.⁹⁶ Furthermore, proposed approaches are often lacking in longitudinal experiences. Wong’s dissertation, for instance, has only executed part of the first iteration: “At present, requirements, concepts of operation, and design for the Notebook have been determined. . . . the next phase is one of more user observation, implementation, and evaluation of the Notebook approach” [Won99, p. 99]. Storey seems to implicitly follow her proposed tool-building process in subsequent incarnations of the SHriMP tool, but unfortunately she has not published her experiences.

Lack of full process

It seems not realistic that a single researcher can define a full process as part of a dissertation. What is needed is a concerted effort of the researchers involved in reverse engineering tool-building. As a first step, researcher have to describe their tool-building approaches

⁹⁵Wong’s user roles correspond to builder, librarian, and patron in the Software Bookshelf [FHK⁺97].

⁹⁶RUP, for instance, addresses these questions with its modeling elements *workers*, *artifacts*, *activities*, and *workflows*, respectively [Kru99, p. 35].

(i.e., *how* has the tool been built as opposed to *what* does it do) and the experiences that they have made. This could then lead to the formulation and subsequent refinement of a full process.

How should such a process look like? One might argue that an existing process can be readily used out-of-the-box. However, it is important to develop a process that addresses the specific requirements of the reverse engineering tool-building domain. A starting point is provided by the process requirements identified in this survey; a process should enable feedback, be iterative, and involve prototyping. A process that addresses the requirements of a particular domain is a *strong* method. In contrast, a *weak* method is domain-independent.⁹⁷ Even though there are no classical definitions of what constitutes strong vs. weak approaches, this dichotomy is nevertheless a useful vehicle to raise awareness that one-size-fits-all approaches have serious shortcomings [GV98] [VG98] [Gla04].

Strong methods

I believe that researchers should strive to develop a dedicated process for the building of reverse engineering tools. Such a process should accommodate the requirements identified above. However, a method need not be jump-started from scratch. It could incorporate techniques from existing methods, or be obtained using a configurable process. An example of such a configurable process is RUP.

The identified process requirements have the potential to positively influence each other, in the best case leading to a positive feedback loop. For example, prototyping has a positive influence on feedback because it encourages stakeholders—especially future users of the system—to provide feedback. Frequent user feedback and the resulting changes in requirements are best accommodated with an iterative process. An iterative process, in turn, should produce an executable system at the end of each iterative development cycle, which can be accommodated with evolutionary prototyping.

Positive feedback
loop

3.4 Research Approach to Identify Requirements

Often frameworks are proposed, or surveys and reviews are conducted without explicitly stating the process of their creation. This makes it difficult for other researchers to judge their scope, applicability, and validity. The surveyed requirements in this chapter have been identified following a process proposed by Kitchenham et al. called evidence-based software engineering (EBSE) [KDJ04] [DKJ05]. Besides EBSE, the approach taken by other reviews and surveys has been studied (e.g., [JBC⁺02] [NW02] [MT02] [FB04]).

Evidence-based
software engineering

The five steps needed to practice EBSE are summarized in Table 1. Whereas the goal of EBSE is quite broad,⁹⁸ its process is applicable to guide the survey. In the following,

⁹⁷This distinction is inspired by research in problem-solving. Strong methods address a specific type of problem, whereas weak methods can be applied to diverse types of problems [GV98]. Vessey gives the following analogy: “A strong method, like a specific size and type of wrench, is designed to fit and do an optimal job for one kind of task; a weak method, like a monkey wrench, is designed to adjust to a multiplicity of problems, but is not designed to solve any of them optimally” [Ves97].

⁹⁸EBSE’s goal is it “to provide the means by which current best evidence from research can be integrated with practical experience and human values in the decision making process regarding the development and maintenance of software” [KDJ04].

Step	EBSE [KDJ04]	Requirements survey
1	Converting the need for information into an answerable question.	What (non-functional) requirements should reverse engineering tools and their development processes satisfy?
2	Tracking down the best evidence with which to answer that question.	Literature search of selected publications (with author's domain knowledge).
3	Critically appraising that evidence for its validity, impact, and applicability.	Review and classification of requirements.
4	Integrating the critical appraisal with our software engineering expertise and with our stakeholders' values and circumstances.	Application of the survey to assess reverse engineering tools.
5	Evaluating our effectiveness and efficiency in executing Steps 1–4 and seeking ways to improve them both for next time.	Assessment of survey coverage; feedback from other researchers.

Table 1: The five steps of EBSE and how they are implemented by this dissertation's requirements survey

I briefly discuss each of EBSE's steps and how it was implemented by this dissertation's survey:

Step 1: As the first step an answerable research question is defined. The question should be general enough so that a sufficient body of work applies. My research questions are:

Research question

What (non-functional) requirements should reverse engineering tools satisfy?

and

What requirements should a tool-building process for reverse engineering tools satisfy?

Step 2: This step involves identifying the relevant work that helps in answering the question posed in Step 1. The survey follows the ideas of *systematic reviews*, which “is a means of identifying, evaluating and interpreting all available research relevant to a particular research question” [Kit04]. Kitchenham et al. point out that “an important issue for any systematic review is to use an appropriate search methodology with the aim of achieving as complete (and, therefore, unbiased) a survey as possible.

Identification of publications

Furthermore, it is an important part of a systematic review to describe the search method” [KDJ04].

I have used scientific databases (mostly IEEE Xplore and the ACM Digital Library, but also Wiley, Springer Online, and Elsevier ScienceDirect) to search for relevant articles. Examples of used keywords are `requirements`, `tools`, and `reverse engineering`. However, since these keywords are general and ambiguous without context, most searches returned a large number of false positives. A further difficulty was that relevant articles often had no or few citations to other relevant articles. In retrospect, my domain knowledge and experience in the reverse engineering field—mostly gained over the years by conducting research and reading articles in the area as well as attending conferences and having discussions with other researchers—was invaluable for the survey.

I also identified relevant sources for further consideration, skimming titles and abstracts of

journals: *IBM Systems Journal*, *Journal of Automated Software Engineering*, *Journal of Software Maintenance*, *Journal of Software Maintenance and Evolution*, *Journal of Visual Languages and Computing*, *ACM SIGSOFT Software Engineering Notes*, *IEEE Transactions on Software Engineering*, *ACM Transactions on Software Engineering and Methodology*, and *Software Process: Improvement and Practice*.

conference proceedings: CASCON, CASE, CSM, CSMR, ICPC, ICSE, ICSM, IWPC, IWPSE, VISSOFT, WCRE, and WPC.

software engineering dissertations: [Fer04] [Hua04] [Jah99] [Köl00] [Kos00] [Mar02] [Moo02a] [Ric03] [Riv04] [Sto98] [Sys00a] [Tic01] [Til95] [Wal02] [Won99] [Zay02].

Since there are no specific rankings of conferences or journals for reverse engineering research, relevance was judged subjectively.

Step 3: To structure the survey, the identified requirements gathered from the sources in the previous step were classified as follows:

Classification of requirements

quality attributes versus functional requirements: The quality attributes are discussed in Sections 3.2.1–3.2.6, whereas the functional requirements are summarized in Section 3.2.7. The survey focuses on quality attributes because they are more generally applicable and more useful for tool evaluations.

general versus component-type-specific requirements: Whenever possible, the survey distinguishes between requirements that are applicable to reverse engineering tools as a whole, and requirements that should be fulfilled by a particular tool component type (i.e., repository, extractor, analyzer, or visualizer).

major versus minor requirements: Quality attributes are classified as major (Sections 3.2.1–3.2.5) or minor (Section 3.2.6). This distinction is mostly made by the identified amount of supporting literature. Hence, a minor requirement does not necessarily imply that it is less important to satisfy compared to a major requirement.

Step 4: To ensure a wide applicability of the survey, I did strive to summarize the requirements that are reported in the literature without introducing a bias. A critical appraisal is provided for each major quality attribute as well as in the discussions in Section 3.2.8 and Section 3.3.5. Application of findings

In this dissertation the survey is applied to assess reverse engineering tools that have been built from components (cf. Chapter 7). Otherwise, the survey currently has no explicit stakeholders; I plan to disseminate the survey and hope for other researchers to provide feedback and report on the usefulness of the survey for their own research. Furthermore, Section 3.2.9 identifies how the survey can be applied to contribute to tool development, tool evaluation, and tool research.

Step 5: It is difficult to evaluate literature surveys of this kind because an exhaustive search is impossible [NW02] [TM04]. One possible approach is to assess a survey’s completeness. I believe that the survey’s coverage of sources is representative of the domain of reverse engineering research. Whereas the survey certainly has missed sources, it succeeded to identify the critical tool requirements. Evaluation of research method

I encourage other researchers to identify missed sources and to critique the survey. To my knowledge there are no comparable surveys of this depth and scope that have been conducted for the reverse engineering domain; this makes it impossible to evaluate surveys via comparative methods such as meta-analysis [KDJ04].

Our survey is timeless in the sense that it does not explore the reported requirements over time [LL04]. If there is sufficient data, it might be interesting to provide a temporal view of the emergence and changes in perception of the discussed requirements.

I feel that EBSE has been beneficial for this survey by providing a guiding framework. However, EBSE gives few recommendations about the individual steps, and since EBSE is a relatively new approach few experiences have been published so far. I hope that my interpretation of the steps has preserved the EBSE’s intentions.

Kitchenham et al. note that there is “little appreciation of the value of systematic reviews” in the field of computer science [KDJ04]. Guidelines for software engineering researchers on how to conduct systematic reviews are just beginning to emerge [Kit04]. As a result, surveys have variable quality. Kitchenham et al. recommend that systematic reviews should be adopted as soon as possible by the software engineering community—including dissertations [KDJ04]. I hope that this survey is a step towards a more systematic approach and has raised the bar for future surveys.

3.5 Summary

This chapter has used a literature survey to identify the requirements for reverse engineering tools as well as for a tool-building process. For reverse engineering tools, the survey has identified five major quality attributes (i.e., scalability, interoperability, customizability, usability, and adoptability), several minor quality attributes, and a number of functional requirements. For a tool-building process, three requirements (i.e., feedback-based, iterative, and prototype-based) could be identified in the literature. The approach of evidence-based software engineering has been used to guide the literature survey. The identified requirements are used later in Chapter 7 to inform recommendations and lessons learned for component-based tool-building.

This and the previous chapter have covered one major theme of this dissertation, reverse engineering. The next chapter introduces the other major theme, software components.

4 Software Components

“Software engineering is increasingly focused on system composition challenges, as opposed to writing particular blocks of code that implement algorithms, data structures, or other functions.”

– Greg Olsen [Ols06]

In this dissertation, I propose to use of software components to build reverse engineering tools. Building a tool out of components is radically different than building one from scratch. Consequently, Section 4.1 starts out with relevant background surrounding software components—history, definitions, rationale, techniques, and process.

I do not propose to use *any* kind of component for tool construction. Instead, to scope my proposed tool-building approach, I focus on *certain kinds* of components. To clearly denote these targeted components, I propose a taxonomy to characterize components and component-based systems in Section 4.2.

4.1 Background

“More than 99 percent of all executing computer instructions come from COTS products.”

– Basili and Boehm [BB01]

Even though this is often not explicitly recognized, software components have played an important role during software construction for a long time.⁹⁹ For instance, existing software functionality is packaged into components in the form of operating systems, mathematical libraries, and database systems to enable reuse by various clients. Component-based systems are composed out of software components. However, this does not mean much: “The phrase component-based system has about as much inherent meaning as ‘part-based whole’ ” [BBB⁺00]. Hence, the purpose of this section is to sharpen our understanding of software components.

The notion of component is quite fuzzy, which can be partially attributed to a number of different viewpoints, for example:

Component
viewpoints

software reuse: This area of research has a broad notion of component, encompassing any reusable asset.

software architecture: This area views components as design abstractions (e.g., patterns or reference architectures).

object-orient programming: Some researchers in the object-oriented community have originally equated components with classes and objects. This view has later broadened to (white-box) object-oriented frameworks.

⁹⁹Hierarchical systems structure that organized systems into layers or levels (as first demonstrated by Dijkstra’s THE operating systems in 1968 [Tic92]) as well as Parnas’ criteria on decomposing systems into modules [Par72] point towards the composition of systems from modules or else components.

commercial products: Practitioners using commercial off-the-shelf (or shrink-wrapped) products for software constructions equate components with such products.

In the remainder of this section, the issues surrounding software components and component-based systems are explored in more detail. I first provide a working definition of what a component is, and then discuss software reuse, component markets, product lines, and component-based software engineering.

4.1.1 Component Definitions

“Trying to come up with a general classical definition for software components is not only futile, but harmful.”

– Czarnecki and Eisenecker [CE00]

There is no single accepted definition what constitutes a component. This dissertation strives for a rather broad definition, because—as the examples given in Section 4.1.2 illustrate—components for tool-building can be drawn from a broad spectrum. I follow Czarnecki and Eisenecker, defining (software) components as

Working definition

“building blocks from which different software systems can be composed” [CE00].

Another definition that would be equally suitable is Meyer’s definition of a (reusable software) component as “an element of software that can be used by many different applications” [Mey97, p. 1200]. An example of a popular component definition that is less broad compared to the ones above is the one given by Szyperski [Szy02, p. 195]:

Other definitions

“A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.”

Both Sametinger [Sam97, sec. 6.1.2] and Szyperski [Szy02, ch. 11] provide collections of existing definitions—taken together they give more than 20 different ones. The earliest definition is given by Booch in 1987, who defines a reusable software component as “a logically cohesive, loosely coupled module that denotes a single abstraction” [Sam97, p. 70]. There are many more definitions in the literature (e.g., Jacobson et al. [JGJ97, p. 85], Broy et al. [BDH⁺98], Kruchten [Kru99, p. 236], Meyer [Mey99] [Mey03], Hopkins [Hop00], Sparling [Spa00], and Graham [Gra01, p. 362] [Gra01, p. 757]).

All of these definitions cannot hope to say precisely what a component is and what it is not; hence, they should not be taken as a *classical definition*.¹⁰⁰ Furthermore, the notion of components also depends on the application domain (e.g., components for embedded systems [MAFN04] [Crn04]). Many of these component definitions have in common that they emphasize the following characteristics:

Component characteristics

¹⁰⁰A classical definition provides a single set of necessary and sufficient properties [CE00, p. 9].

composability: Components need to be composable at system build-time, deployment, and run-time. Composability suggests a certain degree of independence and replaceability of components (regardless whether they are developed in-house or procured).

explicit interface: Each component should have a well-defined, well-documented, and explicitly defined interface (or service). This allows other components (or clients) to (re)use the component.

coherent functionality: A component should encapsulate a coherent set of functionalities (or services), fulfill a clear purpose, and offer a certain value.

4.1.2 Component Types

The definitions of software component vary in their specificity. Authors often give definitions with specific component types already in mind. For example, Szyperski advocates components that are binary and directly deployable without modifications. Other authors such as Meyer have a broader notion of component, also including object-oriented libraries in source form.

In the following, I briefly identify several types of components:

off-the-shelf products: Examples of such products are large commercial applications such as Microsoft Office, Internet Explorer, and Outlook. These products can be purchased and then used as-is. This type also includes open source counterparts with similar functionalities such as OpenOffice, Mozilla, and Ximian Evolution. Operating systems such as Microsoft Windows and GNU/Linux are also off-the-shelf products.

Although often the case, these products are not necessarily GUI-based or intended for end users. An example of off-the-shelf products that are not targeted towards end users are database management systems such as IBM DB/2.

integrated development environments: Examples of integrated development environments (IDEs) or software engineering environments (SEEs) [HOT00] are Eclipse,¹⁰¹ Microsoft Visual Studio, IBM Visual Age, TogetherSoft Together, and Rational Rose. Examples from academia are Reiss' Field and Desert environments [Rei90] [Rei96] [Rei99] as well as CWI's ASF+SDF Meta-Environment [vdBdJJK03] [vdBHdJ⁺01]. Many of these are also off-the-shelf products, but intended for software development activities.

In the following, I briefly discuss the ASF+SDF Meta-Environment, which supports automated software engineering activities such as source-to-source transformations and (interactive) program analyses. The IDE consists of a number of independent components that are connected by a message-bus component called Tool-

ASF+SDF
Meta-Environment

¹⁰¹Eclipse has broadened its scope with Version 3.0 from IDE to an open platform for building applications of all sorts. This new vision is called the Eclipse Rich Client Platform (www.eclipse.org/rcp/).

System	ATerms	ApiGen	SDF	BOX	ASF	TIDE	ToolBus
Chi	•	•					
Stratego	•		•	•			
Lucent	•						
haRVey	•		•	•			•
ELAN	•	•	•		•		•

Table 2: Reuse of Meta-Environment components by various projects ([vdBV05])

Bus [dJK03]. The ToolBus allows heterogeneous components in a possibly distributed system to communicate by sending messages to each other (which leads to a message-oriented middleware architecture). There is a broad range of available components [vdBV05]: ATerms (an untyped data exchange format that can be used to send messages and data over the bus), ApiGen (typed ATerms), SDF (context-free language parsing), BOX (declarative pretty printing), ASF (transformations of programs) RScript (relational calculus), and TIDE (debugging). Besides these components, the Meta-Environment incorporates external components such as editors (Emacs and Vim), and AT&T graphviz for graph layouts. The components of the Meta-Environment can be independently reused and composed to build new systems. Table 2 shows that this promise is realized in practice. It gives a number of systems from different research groups along with the reused Meta-Environment components.

domain-specific tools: There are many small to mid-size programs that handle specific tasks. For example, Unix¹⁰² has various small programs for effective manipulation of textual data. Another example are text editors such as Emacs. However, the Emacs editor includes functionalities typically also found in integrated development environments such as syntax highlighting, pretty printing, spell checking, compilation, and debugging [Cur02]. Of specific interest to reverse engineering are graph visualizers and graph editors such as the AT&T graphviz package.

application generators: Application generators generate code for (parts of) a software system [Kru92, sec. 7]. The generation is typically based on a specification language, from which components with certain interfaces are generated so that they can communicate with the rest of the system. Application generators can be seen as a form of generative reuse, discussed in Section 4.1.3. Ideas from application generators can be also found in the OMG’s Model Driven Architecture¹⁰³ (MDA) [MCF04].

Typical examples of this component type are scanner and parser generators such as `lex` and `yacc`, respectively. An example of a more advanced generator is the LISA compiler-compiler, which generates a compiler or interpreter from a specifi-

¹⁰²I use “Unix” as a euphemism for Unix-like systems such as AIX, Solaris, and GNU/Linux.

¹⁰³<http://www.omg.org/mda/>

cation based on attribute grammars [HPM⁺05]. The ASF+SDF Meta-Environment offers similar functionality but is based on algebraic specifications. Other examples are Meta-CASE tools, which provide facilities for specifying and generating CASE tools (e.g., IPSEN [KS97], MetaEdit+¹⁰⁴ [KM97], and the Generic Modeling Environment¹⁰⁵ [LMB⁺01]), and fourth-generation languages (4GLs) such as (relational) query languages and report generators [Cez95, p. 472].¹⁰⁶ Generative approaches have also been used to automatically produce parts of reengineering environments [CMW02] [RS04].

(off-the-shelf) components: In contrast to full products, (off-the-shelf) components provide more limited functionality and cannot be directly executed. Instead, a number of such components are integrated to form a component-based system.¹⁰⁷

When building a component-based system, is it is not practical to integrate any number of diverse components and make them interoperate—Bass et al. state: “components are often *almost compatible*, where *almost* is a euphemism for *not*” [BCK98, p. 338]. In practice, there are several established “wiring” standards¹⁰⁸ for components, simplifying the task to make components interoperate [Szy02, ch. 12]: Sun’s JavaBeans and Java Enterprise Beans (EJB), Microsoft’s Component Object Model (COM) and Distributed COM (DCOM), and OMG’s Common Object Request Broker Architecture (CORBA) [Lon01b] [PS98] [Lew98] [Szy98]. Less well-know examples are GNU Bonobo¹⁰⁹ [Mee01], Mozilla’s XPCOM¹¹⁰ [vGBS01], and Oberon Microsystem’s Component Pascal¹¹¹ [Szy02, page 549].

Wiring standards

Microsoft’s ActiveX controls are (visual) components based on COM that can be embedded into HTML pages and visualized with Internet Explorer. Adobe’s SVG viewer is an example of an ActiveX control. ActiveX controls can be nested—Internet Explorer itself is also an example of an ActiveX control. ActiveX components have no matching interfaces, which makes it necessary to glue them together programmatically, usually with Visual Basic¹¹² [vOB02, p. 334].

ActiveX

¹⁰⁴<http://www.metacase.com/>

¹⁰⁵<http://www.isis.vanderbilt.edu/Projects/gme/>

¹⁰⁶4GLs are programming languages that require little programming knowledge from users compared to fully-fledged programming languages. They are often declarative (allowing the user to express *what* is to be done rather than *how* it is to be done).

¹⁰⁷According to Sullivan, “to *integrate* means to put parts together into a functioning whole” [Sul01].

¹⁰⁸Historically, the origin of such standards is rooted in binary calling conventions to achieve interoperability of function calls across programming languages, and then later conventions for remote procedure calls to achieve interoperability across different machines.

Wiring standards are closely related to the concepts of component kit (i.e., a set of components along with standard protocols for connecting them together) [Gra01, p. 757], and component model (i.e., a set of component types, their interfaces, and, additionally, a specification of the allowable patterns of interaction among component types) [BBB⁺00].

¹⁰⁹<http://developer.gnome.org/arch/component/bonobo.html>

¹¹⁰<http://www.mozilla.org/projects/xpcom/>

¹¹¹<http://www.oberon.ch/>

¹¹²http://en.wikipedia.org/wiki/Visual_Basic

Plug-ins (or add-ins) for particular (off-the-shelf) products and IDEs are another example of off-the-shelf components [Bir05] [LvdH02, sec. 4.1.5] [DeL99]. Examples of products that support plug-ins are Apple's QuickTime, Mozilla, Adobe's Photoshop, and the Apache Web server.¹¹³ Many operating systems support extensions with a plug-in concept (e.g., Mac OS extensions and Linux kernel modules) [Ric97]. Eclipse is a popular IDE based on a multi-level plug-in architecture [Lüe03]. Plug-ins typically provide a low overhead of adding a smaller piece of well-defined functionality to the base application (e.g., a new filter or brush in Photoshop, or a tool-bar search-engine in Mozilla). The wiring model of plug-ins is often restricted to a few extension points (e.g., a new entry in the menu or tool-bar). However, some products such as Apache allow plug-ins to extensively modify the base application's behavior.

Plug-ins

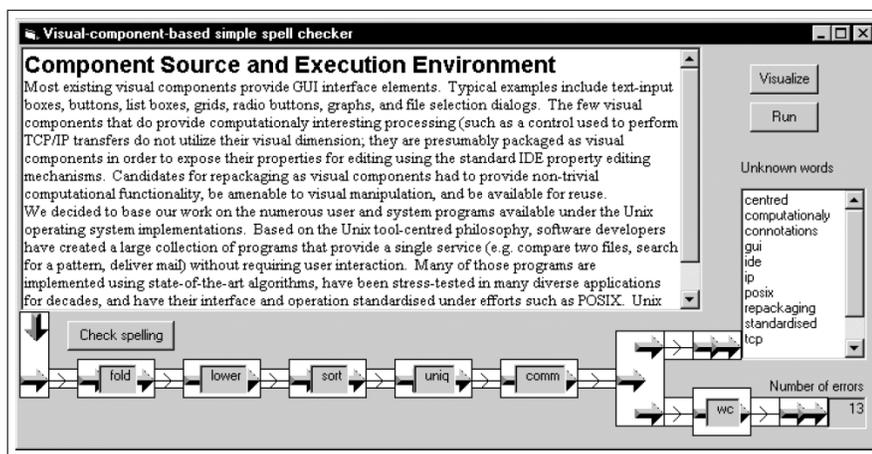


Figure 9: Unix filters wrapped as ActiveX components ([Spi02])

A particular component will support only one standard. However, wrappers¹¹⁴ can be used to integrate components that are based on different wiring standards [Gra95]. For example, IONA's Orbix COMet achieves interoperability between CORBA and COM, JavaSoft's ActiveX Bridge makes it possible to embed JavaBeans into ActiveX containers [Szy98], a dedicated Mozilla plug-in makes it possible to embed ActiveX controls, and so on [PG05] [Mau05] [LW04, p. 24] [Che03]. Spinellis has wrapped Unix filters such as `sort` and `wc` into non-visual COM components and then into visual ActiveX components [Spi02]. The resulting ActiveX components can be used for visual programming in the Microsoft Visual Basic IDE to assemble applications. Figure 9 shows a screenshot of a GUI-based spell checker implemented with Visual Basic GUI elements and ActiveX components. The document in the large text box (top left) is run through several ActiveX-wrapped Unix filters to pro-

Wrappers

¹¹³There are also portals for downloading plug-ins. For example, more than 500 Mozilla plug-ins are available at <http://mycroft.mozdev.org/>.

¹¹⁴One can distinguish between wrappers and bridges as an integration mechanism [BCK98, p. 339ff] [DeL99]. However, these two mechanisms are similar and I use them interchangeably.

duce a list of words not found in the dictionary. This list is shown in another text box (middle right) and also run through another filter to count the number of unknown words (bottom right). All wrapped Unix filters implement the necessary interfaces that enable visual programming in the Visual Basic IDE as well as a custom interface to initialize and execute the filters. The glue code is only two lines. In Figure 9, the Unix filter components and pipes are shown for illustration purposes and can be hidden from the end user.

compound documents: Compound documents are composed of visual (GUI) components. The visual components can range from simple buttons, over desktop calculators, to sophisticated spreadsheets. End users can create new compound documents via interactive (drag-and-drop) manipulation that is similar to the functionality offered by GUI builders. With this approach, new graphical applications can be built by simply connecting preexisting components, which are potentially purchased from different vendors. As a result, the end user partly takes the role of a software developer [Sam97]. Thus, compound documents are a specialized form of off-the-shelf components.

An early example of a commercially successful approach to compound documents is Apple's Hypercard, which had a simple and intuitive composition mechanism—in effect making “everybody a programmer” [Gre87].¹¹⁵ Another example is Visual Basic controls (VBXs), which are glued together via Visual Basic scripting to realize complex graphical forms [Ude94]. VBXs created one of the first component markets [Szy02, p. 213].

Hypercard and VBX

Microsoft's Object Linking and Embedding 2.0 (OLE) generalizes the concepts of VBXs, allowing the hierarchical composition of arbitrary document elements into larger compound documents. An OLE server provides document elements, which are embedded into an OLE container.¹¹⁶ Examples of OLE components range from simple document elements such as pictures and text to Excel sheets and PowerPoint slides; hierarchical composition makes it possible for a PowerPoint slide to be embedded into another PowerPoint slide. A competing compound document architecture that was pushed by Apple, IBM, Sun, and others is OpenDoc [Pie94] [Adl95].

OLE and OpenDoc

There are also approaches to render compound documents in Web browsers. The HotDoc compound document framework (implemented in Smalltalk) makes on-the-fly HTML conversions of documents for Web publishing [Buc00]. Mitchell implements compound documents with Java applets as components [Mit98]. These applets can then be embedded in Web pages or Lotus Notes documents. Figure 10 shows an example of a compound document in Lotus Notes. Standards such as XForms¹¹⁷ and

Browser-based approaches

¹¹⁵However, creating new components was difficult to achieve [Szy02, p. 212].

¹¹⁶Van Ommering notes that a large number of interfaces have to be implemented to realize an OLE container and as a consequence there are comparably few around [vOB02].

¹¹⁷<http://www.w3.org/MarkUp/Forms/>

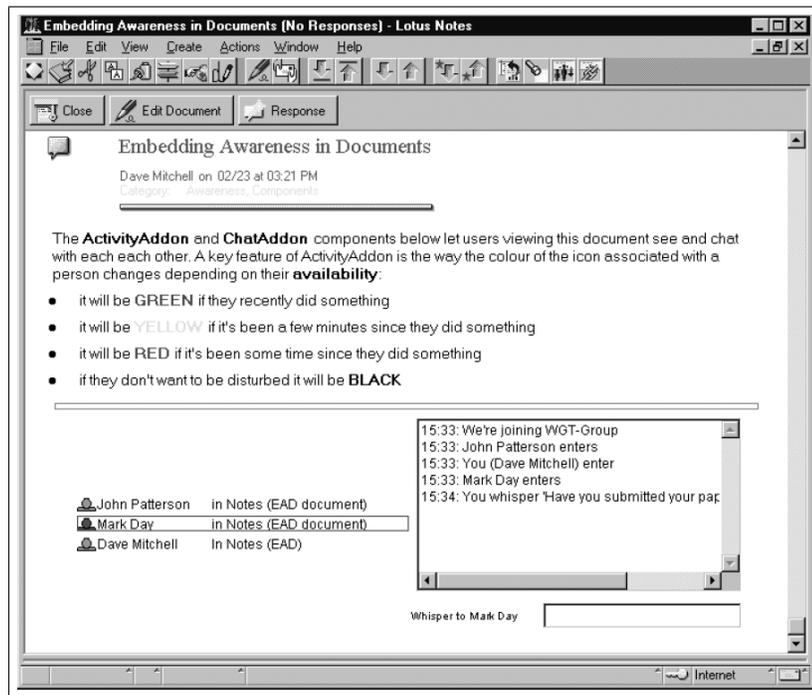


Figure 10: Compound document rendered in Lotus Notes ([Mit98])

JavaServer Faces¹¹⁸ are steps towards native compound documents on the Web.

Similar to off-the-shelf components, compound documents are based on a wiring standard to achieve interoperability. For example, Mitchell's Java applets are wired together with a special InfoBus, which "allows cooperating applets to rendezvous and communicate with each other as data providers and consumers" [Mit98]. There are wrappers to achieve interoperability between different standards. For example, OpenDoc's Component Glue provided a wrapper to integrate OLE components [Sam97, p. 95].

frameworks: Object-oriented frameworks provide a set of abstract and concrete classes with explicit variation points, which can be instantiated (e.g., via subclassing or delegation) to create a complete application. Thus, a framework is a reusable application skeleton that still needs to be fleshed out by an application developer [Joh97, p. 39].

There are many well-known examples of frameworks for the GUI domain: Smalltalk-80 [KP88], InterViews [LVC89]; MacApp; Java's Abstract Windows Toolkit (AWT) and Swing; Eclipse's Standard Widget Toolkit (SWT); Microsoft Foundation Classes (MFC); Windows Presentation Foundation (WPF);¹¹⁹ and so on [FS97]. Frameworks are often used as a technique to support the wiring of compo-

GUI frameworks

¹¹⁸<http://java.sun.com/j2ee/javaserverfaces/>

¹¹⁹http://en.wikipedia.org/wiki/Windows_Presentation_Foundation

nents. For example, MFC provides functionality to simplify the implementation of OLE and COM components. Similarly, OpenDoc was realized with a component framework [Szy98]. For the Web domain, there is the Jakarta Struts¹²⁰ framework to simplify the construction of Web applications.

There are also enterprise application frameworks that target vertical application domains. Examples are SAP's R/3 Business Framework [Dei98] and IBM's San Francisco [Hen98]. The San Francisco framework has various layers. Lower layers provide support infrastructure such as GUI elements, collection data structures, and distributed objects (i.e., transactions, persistence, concurrency, etc.), while the highest layer has reusable core business components for order and warehouse management.

Enterprise
application
frameworks

libraries: Libraries provide a coherent collection of functionalities in the form of procedures or classes. There are many libraries for data structures and algorithms such as the C++ Standard Template Library (STL). Another example is the LEDA library,¹²¹ which also has a large variety of graph data structures (e.g., parameterized graphs, planar maps, node/edge arrays, and node/edge maps). Many applications depend on a surprisingly large number of diverse libraries, which in turn depend on other ones. An example illustrating these dependencies is depicted in Figure 11, which shows the library dependencies of the FreeBSD port of the `xine` multimedia player [SS04].

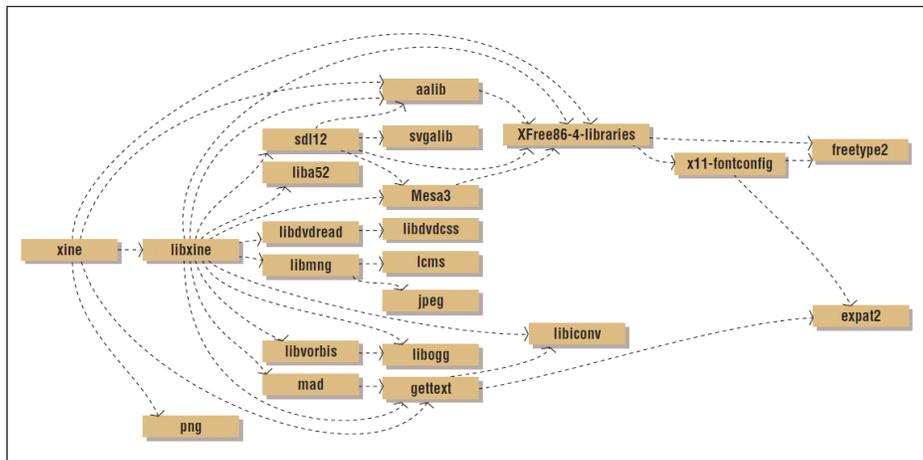


Figure 11: Library dependencies of `xine` ([SS04])

There are often (de facto) standard libraries that are shipped with a programming languages such as the C standard library, the STL,¹²² and the predefined library of Ada95 [Bar96]. ISE's version of Eiffel ships with the EiffelBase library, consisting

Standard libraries

¹²⁰<http://struts.apache.org>

¹²¹<http://www.mpi-sb.mpg.de/LEDA>

¹²²Before the STL became available, many organizations were forced to implement proprietary libraries with similar functionality (e.g., `C++_complib` [CL00b], and the NIH Class Library).

of hundreds of classes covering basic data structures and algorithms, GUI programming, lexical and syntactic analysis, and so on [Mey97, app. A].

The distinction between object-oriented libraries and frameworks is blurry. Typically, libraries are collections of classes that are used as-is, while frameworks require an explicit instantiation step (e.g., via subclassing of framework base classes, or via implementation of framework interfaces). Clients of libraries typically instantiate library classes and make calls; clients of frameworks get called back from framework code.¹²³ In contrast to the EiffelBase library and the Smalltalk-80 collection library, the STL makes no use of inheritance. Instead, C++ templates are used extensively [MKES98]. Microsoft's Active Template Library (ATL) provides similar functionality to MFC, but its implementation is based on templates whereas MFC's implementation is based on inheritance.

Libraries vs.
frameworks

others: A component can be any other reusable element of software with a coherent set of functionalities (e.g., functions and classes, possibly aspects and concerns [CE00, p. 125f]). However, practical experience has shown that it is often difficult to reuse fine-grained assets such as individual classes because of their tight dependencies on other assets [Szy98].

The above types are meant to convey the broad spectrum of software components. However, it is not a taxonomy: a particular component might well fall under several of the introduced component types. All of the above component types have in common that they are a means to package or bundle a set of features or functionalities and that they make (part of) their features or functionalities available for reuse via an explicit interface.

4.1.3 Software Reuse

“Microsoft managers determined they had fourteen different collections of text-processing code in their products.”

– Cusumano and Selby, qtd. in [JGJ97, p. 14]

A concept closely related to software components is software reuse [MMM95]. Software reuse is the process of creating software systems from existing assets rather than from scratch [Kru92].¹²⁴ Reusable assets might be personnel, (formal) specifications, analysis and design models, (reference) architectures, patterns, algorithms, source and binary code, and so on [Pri93] [Mey97, sec. 4.2] .

Definition

Even though producing a software system from scratch can have significant advantages (e.g., it can be optimally tailored to the user's business model and requirements), it is an expensive and risky undertaking. Commercial interest in software reuse is grounded in its potential to increase productivity and to reduce development cost, while improving software quality and shortening time-to-market [Gri98] [PCH93]. A similar case for reuse

Reuse Benefits

¹²³This is also called Hollywood Principle: “Don't call us; we'll call you.”

¹²⁴Some definitions require that the reused code has to be designed to be reused—otherwise it is software salvaging [Edw99].

can be made for academic research, especially if it is a software-intensive activity such as tool-building. Reusable components also facilitate rapid prototyping [Sam97] [Sta84].

For high-level programming languages, examples of reusable source code assets are procedures, modules, (generic) packages, and libraries; object-oriented languages in addition offer classes. Software components can be seen as an example of a particular approach to code reuse. According to Czarnecki and Eisenecker, a goal of using components is “to minimize code duplication and maximize reuse” [CE00].

Code reuse

There are many different forms of reuse ranging from design and code scavenging to very high-level languages (VHLL) [Kru92]. However, one can distinguish between two major techniques to achieve reuse [BP84] [Pri93]: compositional and generative reuse. Compositional reuse assumes existing components, which are used as building blocks to assemble the final system. New systems are built using interface and composition mechanisms. A typical example of a component integration framework is the Unix *pipe* mechanism, which composes new programs from existing ones (also called *filters*) by connecting the textual output of one program to the input of another [BP84].¹²⁵ In this case, the components are complete programs in binary form [Kru92, sec. 5.4].

Compositional versus generative reuse

In contrast, the generative technique automatically generates code for all or parts of a software system based on a higher-level specification [Big98] [CE00, ch. 9]. Examples of generative techniques are application generators (cf. Section 4.1.2). More advanced generational techniques are Neighbors’ Draco and Batory’s GenVoca systems [Big98]. The reusable assets in these techniques are less intuitive and less palpable because they consist of patterns for code generation and transformations.

Whereas the use of a set of standard components to build a customized software solution is a step towards automatic mass-production, most approaches assume a manual assembly of software components to produce the final system (e.g., Jacobson et al.’s Application System Engineering process [JGJ97]). In contrast, generative programming’s *automation assumption* states: “If you can compose components manually, you can also automate this process” [CE00, p. 13]. Thus, the vision of generative programming is the automatic manufacturing of software products out of components. Implementation technologies to realize generative programming are application generators, C++ template metaprogramming, model-driven development [MCF04], and Aspect-Oriented Programming [CE00, ch. 8].

Generative programming

4.1.4 Component Markets

“It is logical to conclude a market for components would develop, as the number of suppliers increase, and the number of consumers building systems increases. Components are potentially reusable assets, but it is not guaranteed that a component will be used simply because it exists.”

– Jon Hopkins [Hop00]

¹²⁵Salus summarizes the philosophy of Unix as follows: (1) write programs that do one thing and do it well; (2) write programs to work together; and (3) write programs that handle text streams, because that is a universal interface [Sal94, p. 53]. In fact, the last point can be seen as an instantiation of a simple wiring standard [Wil04].

Since software components are reusable assets, they can be traded in a market like any other product. Actors in a component market are component suppliers, brokers, systems integrators (i.e., consumers), and potentially third-party evaluators¹²⁶ and certifiers [Bre04]. A discussion how markets are created and function is beyond the scope of this dissertation. However, intuitively, a market is healthy if supply and demand are in balance. There needs to be a critical mass of components (supplied by independent software vendors) with sufficient variety and quality that offer convincing benefits to consumers.

The Internet is well suited as a commerce platform to buy and sell software components. Components can be offered by producers themselves on their own Web site, or by an intermediary or broker. Prominent examples of intermediaries are ComponentSource¹²⁷ and Flashline¹²⁸ [Szy02]. Offered components adhere typically to the following standards: JavaBeans, CORBA, and ActiveX. A study conducted in 2000 concluded that there were relatively few suppliers on the Internet component market at the time [TvH00]; and little has changed in the following years [Szy02, sec. 2.3] [HL02b] [RR03] [US04].

Internet component
market

Perhaps the biggest obstacle for an emerging component market is heterogeneity of both supply and demand [US04]. From the consumer's perspective, the current component market consists of suppliers offering diverse solutions and associated service packages without following any established standards. Thus, consumers have difficulty to evaluate, compare, and select components. On the other hand, consumers have diverse requirements, making it difficult for suppliers to understand and target consumer needs [Hop00].

Market heterogeneity

4.1.5 Product Lines

“It is our belief that one reason why reuse has been so long in arriving is that large-scale reuse depends on having a product line approach to software.”

– Bass et al. [BCK98, p. 332]

Product lines are an approach to software development that combine the ideas of reuse and software components [Gri01]. A product line is “a collection of systems sharing a managed set of features constructed from a set of core software assets” [BCK98, p. 331]. Thus, product lines reuse core assets across the members of a product line. Reusable assets are primarily software components (including code as well as design work), but also processes and tools. The software components of a product line can be custom-made components or off-the-shelf components procured at component markets.

A product line is scoped to satisfy the needs of a particular market (e.g., medical imaging systems or diesel engines [Coh02]).¹²⁹ An important part of the product line is its architecture, which captures the produce line's common and variable parts. An architec-

Scoping and
variability

¹²⁶Examples of companies that assess products of certain market segments are Gartner (<http://www.thegartnergroup.com>) and Ovum (<http://www.ovum.com>) [LM04].

¹²⁷<http://www.componentsource.com>

¹²⁸<http://www.flashline.com>

¹²⁹In contrast to a product line, a *product family* is a collection of systems that can be built from a common set of assets [CE00, p. 36]. Hence, product families are primarily scoped by technical commonalities rather than market strategy. However, both concepts are sometimes treated the same in the literature.

ture can be derived with dedicated methods such as SEI's Framework for Software Product Line Practice¹³⁰ [Nor02] [CN01], Fraunhofer IESE's PuLSE¹³¹ [BFK⁺99], and Lucent's Family-Oriented Abstraction, Specification and Translation (FAST) [Mat04]. A certain product is realized by instantiating the product line's variability points. The bounds of the variability points effectively scope the product line. There are various approaches to identify and represent variabilities (e.g., [MZG03] [AHI⁺05]), which then can be realized with various techniques (e.g., binary replacement, or component customization [SvGB05]).

Software product lines have the potential to reduce costs, decrease time-to-market, and improve the quality of the individual products [KS01]. However, from an organization's perspective, adoption of a product line strategy is not a trivial endeavor [Kru02] [Bos99]. The organizational structure changes into a unit that defines the product line's architecture and manages the core assets, and units that build the actual products using the core assets. As a result, the owners of the core assets and the owners of the individual projects resemble the relationship between a software vendor and its customers [BCK98, p. 334]. Products are now constructed as component-based systems via using and customizing of core assets. As a result, product developers will lobby the core asset owners to evolve core assets to better provide for their needs. Conversely, product developers need to be informed about new versions, extensions and other relevant information to perform optimally. Evolution of components and products may lead to splitting off a product from the product line.

Impact of product
lines

4.1.6 Component-Based Software Engineering

“The possibility of a Software Industrial Revolution, in which programmers stop coding everything from scratch and begin assembling applications from well-stocked catalogs of reusable software components, is an enduring dream that continues to elude our grasp.”

– Brad Cox [Cox89]

The idea of “mass-produced software components” was raised by McIlroy as early as 1968 at the famous NATO conference on software engineering [McI76] in which he argued that the production of software is currently focused on *building* rather than (*re*)using:

Software mass
production

“Software production today appears in the scale of industrialization somewhere below the more backward construction industries. I think its proper place is considerably higher, and would like to investigate the prospects for mass-production techniques in software. . .

When we undertake to write a compiler, we begin by saying ‘What table mechanism shall we build?’. Not ‘What mechanism shall we use?’ ” [McI76] (qtd. in [Mey97, p. 99]).

McIlroy implicitly introduced the notion of software reuse by proposing a component industry (even though at the time his envisioned components were assembly language and

¹³⁰<http://www.sei.cmu.edu/productlines/framework.html>

¹³¹<http://www.iese.fraunhofer.de/PuLSE/>

Fortran subroutines) [Kru92].

Almost 40 years after McIlroy, software engineering is still characterized by hand-crafting rather than assembly-line-like mass-production of software systems. An important prerequisite for mass-production are interchangeable parts, or components, in the manufacturing process [Cox90]. Component-based software engineering (CBSE) [Crn02] [Gra01] [BBB⁺00] [KB98] [BW98] is the process of building systems via the combination and integration of software components.¹³²

There are various potential benefits and risks associated with CBSE in general [US04] [Vit03] [TLH97], and systems based on off-the-shelf components in particular [TM04] [Tra01] [BA99] [HC99] [Voa99] [VGD96]. Benefits for the developer of component-based systems include: immediacy of acquisition, reduced in-house software development risks, lower development costs, higher productivity, more manageable quality, and greater ease of modernization.

On the other hand, examples of potential risks are few (component) standards and established vendors, different skill sets needed compared to traditional software construction, mismatch of buyer's acquisition process and seller's marketing/packaging strategies, inadequate and expensive vendor support, inadequate documentation, frequent upgrades, dependence on component vendor, introduction of additional constraints into the overall system, difficulty to gauge the quality and reliability of a component, version incompatibilities between different releases, and additional staff training for maintenance.

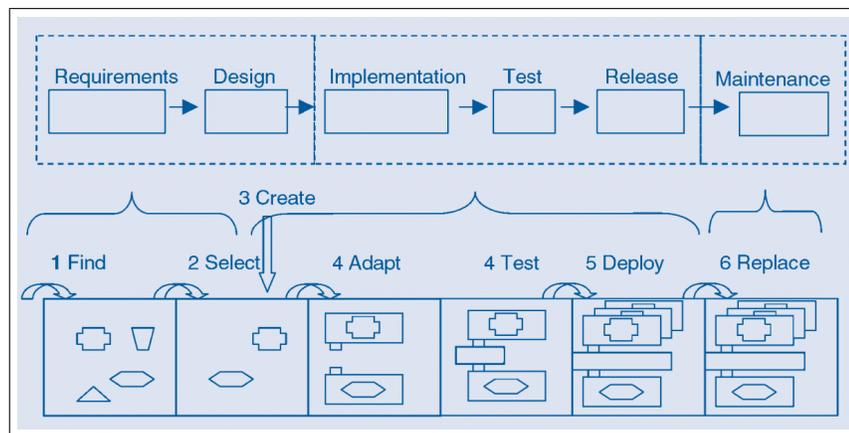


Figure 12: CBSE development cycle compared with the waterfall model ([Crn02])

The development of component-based software differs significantly from the construction of traditional software systems. As a consequence, the activities of the traditional waterfall model (i.e., requirements analysis and specification, design, implementation, test, release, and maintenance) have to be suitably modified and extended. Figure 12 shows how the activities of the waterfall model correspond to the activities of a component-based

¹³²CBSE is also called component-based software development (CBSD) and component-based development (CBD).

Component-based
software engineering

CBSE benefits

CBSE risks

CBSE process

development process. Based on the growing experience with component-based systems, many new CBSE processes have been proposed (e.g., [BW96] [JGJ97] [TLH97] [BOS00] [Hen01] [MSB⁺02] [KH05]). Typical activities of such processes and why these activities are important for CBSE are briefly discussed in the following:

find: In this activity, potentially suitable components are identified that may be used in the final system. For instance, to find suitable candidates one can use Web searches; product literature surveys and reviews; and recommendation from external sources [MSB⁺02]. Also, it is beneficial to create and maintain knowledge of the marketplace through market research, technology watch, and participation in industry and user groups [BOS00]. There are initiatives such as the eCots portal that enables participants to share information on COTS products and producers [MLL⁺03].

select: Out of the candidate components, one is selected that best meets the system's requirements. A selection is based on a thorough evaluation of the component (e.g., in-depth review of manuals and user guides [MSB⁺02]). To reduce the risk of choosing an unsuitable component, a decision should not be based solely on technical documentation, but also employ prototypes and pilots to gain a deeper understanding ("try before you buy") [BOS00]. There are dedicated selection methods that evaluate off-the-shelf products with respect to the system's requirements [PC04] [Gol04]. However, in practice, candidates are often selected based on the developers' familiarity with a component, or its technology [TM04]; or on "slick demos, Web searches, or reading trade journals" [Tra01].

It is unlikely that a candidate can fulfill all requirements completely—Golden puts it as follows:

"No product will completely meet the ideal functionality specification. The requirements gathering phase identifies a wish list of every person's (or organization's) desires. Just as actual individuals always fall short of one's ideal romantic partner, actual software products never deliver ideal functionality" [Gol04, ch. 5].

Often requirements need to be changed to make it possible to use a component.¹³³ Boehm says, "it's not a requirement if you can't afford it." [Boe00]. To facilitate the adaptation of requirements, one can determine and prioritize requirement into negotiable and non-negotiable [BOS00].

adapt: In this activity, the selected components are adapted or customized to meet the system's requirements. Often the selected components cannot readily interact with each other because of architectural mismatch [GAO95]. Typical approaches to component adaptation are tailoring of components, component wrapping, and the writing of glue code [VD97].

¹³³In extreme cases, organizations change their business processes to make their requirements compatible with the ones of a COTS product [Sch04] [NP04].

Component tailoring enhances the functionality of a component in ways that are supported and envisioned by the component vendor (e.g., via plug-ins or call-backs). Wrapping encapsulates a component with an alternative interface (e.g., to make the component's functionality available to CORBA clients [BCK98, p. 340]). Glue code provides the programming logic to make the various components interoperate, for instance, invoking functionality of components, resolving interface incompatibility between components, and handling errors.

create: If no suitable component can be found, it needs to be developed from scratch (i.e., traditional software development).¹³⁴ This can be a significant part of the project effort [MSB⁺02]. CBSE strives to avoid this scenario because of the drawbacks discussed above. The non-COTS development effort can proceed in parallel with the adaptation of COTS components.

test: Testing of the system involves testing of the components and the glue code. Brownsword et al. recommend to test early and continuously to discover problems as quickly as possible [BOS00]. Testing can be done piece-by-piece as each software component is integrated [MSB⁺02].

Testing and debugging of COTS components is complicated by the fact that source code is typically not available [Vid01].¹³⁵ For components that are opaque, "their behavior can be observed but their inner workings are not visible" [HC99]. Techniques can be used to peek into such components (e.g., Unix's `ps`), to expose communication between components (e.g., Unix's `trace` and `ipcs`, or the Smiley tool for Windows [Gol00]), and to measure a components performance (e.g., via execution traces [Put04]).

deploy and maintain: Deployment encompasses initial delivery and installation of the system. Often there is no strict separation between deployment and maintenance [BOS00]. Maintenance events such as upgrading components to newer versions¹³⁶ or replacing of proprietary in-house components with off-the-shelf components [CL00b] can occur before deployment. Consequently, product selection and testing can become part of maintenance.

Compared to traditional maintenance, vendor components change maintenance because these components follow their own upgrade schedules. A vendor component upgrade can add, change, or eliminate features [BCO98]. Not upgrading a component may not be an option if the vendor ceases to support previous versions [VGD96]. The impact of each (potential) upgrade needs to be thoroughly analyzed, requiring

¹³⁴Alternatively, an organization can use an approach such as SEI's Options Analysis for Reengineering (OAR), which identifies existing software assets that may be applicable for reuse, and analyzes the required changes to make the assets conform to the new system's architecture [BOS01] [OHSS02].

¹³⁵There are proposals of built-in test (BIT) components that have a dedicated testing interface [BBB03].

¹³⁶Tracz states that the maximum shelf life of a COTS component is two years [Tra01].

significant time and effort [RBBC03] [RBBC04].¹³⁷ New features in a component may require nontrivial changes in the component’s tailoring, glue code, test scripts, and so on. Reifer et al. believe that maintenance complexity (and costs) increase exponentially with the number of independent COTS components in the system [RBBC03].

4.2 Component Taxonomy

This section introduces a taxonomy to characterize components and component-based systems (CBSs). There is already a number of other component taxonomies that cover diverse characteristics (cf. Section 4.2.7). Rather than striving for an all-inclusive taxonomy, this taxonomy’s goal is it to better characterize the components that I propose for tool building. Thus, the taxonomy concentrates on relevant criteria that have a significant impact on tool building.

To characterize components and CBSs, the following criteria are used: origin, distribution form, customization mechanisms, interoperability mechanisms, packaging, and number of components. The resulting taxonomy is both small and lightweight.

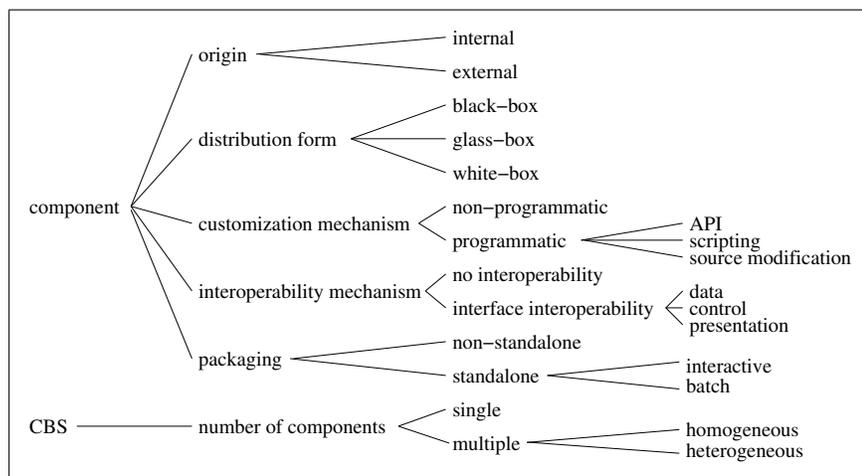


Figure 13: Component taxonomy at a glance

The following sections introduce the taxonomy’s criteria in detail. Figure 13 presents an overview of the complete taxonomy.¹³⁸ The taxonomy can be divided into two main groups: criteria describing software components (“component”), and criteria describing systems that are built from these components (“CBS”).

¹³⁷In the Microsoft world, the incompatibilities between different version of shared libraries are aptly called “DLL Hell” [Hop00] (http://en.wikipedia.org/wiki/DLL_hell).

¹³⁸Taxonomies often use tree views to visualize their organization (e.g., reusable software components [Sam97, fig. 9.1], algorithm visualization [HD02, fig. 5], network protocol visualization [CI02, fig. 1], software visualization [PBS93, fig. 20], cognitive design elements for software exploration [SFM99a, fig. 2], and application domains [GV95, fig. 3]).

4.2.1 Origin

A component's origin or source can be classified as internal vs. external:

internal: The component is developed by the same engineers that build and assemble the final system. In a sense it is a “non-off-the-shelf component” [CL00a] that might be later put into a reuse repository and used for other systems.

external: The component is obtained from an external source (e.g., an in-house reuse repository or a commercial component market), and not developed by the engineers that build and assemble the complete system. Such components are also called nondevelopmental items (NDIs) [CL00a] or off-the-shelf (OTS). NDIs can be further classified by their sources: commercial vendor (COTS),¹³⁹ government-owned (GOTS), military-owned (MOTS) [VGD96], or developed by research institutions (ROTS) [THP03] [Abt02].

In the following, this dissertation mostly uses the term off-the-shelf (OTS) in an effort to avoid potentially misleading qualifications such as commercial OTS or COTS. Many researchers use COTS without defining its meaning and without distinguishing it from OTS—in such cases, COTS and OTS are assumed to be equivalent and consequently this dissertation picks the more general term. In contexts where a distinction is actually important, I use the qualification explicitly for emphasis.

OTS vs. COTS

In contrast to this taxonomy, Carney and Long distinguish the following types of component sources [CL00a]: independent commercial item, custom version of a commercial item, component produced to order under a specific contract, existing component obtained from external sources (e.g., a reuse repository), and components produced in-house. Based on the classification of Carney and Long, Morisio and Torchiano propose the following sources of origin: independent commercial, special version of commercial, externally developed, existing external, and in-house [MT02]. However, for both of these more fine-grained classifications it is not always straightforward to decide on a component's applicable origin.

Carney and Long's classification

4.2.2 Distribution Form

A component can be made available to clients in various forms. This classification focuses on the availability and modifiability of the component's source code:

black-box: Black-box distribution of code means that the component's source code is not available (i.e., closed source). Instead, the component is made available in some binary form. For example, most shrink-wrapped software is available only as stand-alone executable.

¹³⁹The commercial character of a component is not easy to define. For example, Carney et al. ask: “Is something ‘commercial’ if its vendor merely intends to sell it, but has sold none so far? How many sales are needed to qualify something as legitimately commercial? What if the seller does not particularly want to sell it, but is merely willing to sell it?” [CHP00].

Black-box components have certain disadvantages. For example, since source code is not available, “most forms of software analysis that would help you decide if the software is going to perform safely, securely, and reliably are not available” [Voa98].

white-box: This form of distribution is also called open source [SS04]. Thus, interested clients can inspect and modify the component’s sources. In practice, however, clients treat white-box components as black-box or glass-box for most development efforts [TM04].

glass-box: Glass-box (or gray-box) distribution means that clients have access to (parts of) the sources but are not allowed to modify them. This distribution form is of interest if the client wants source code for inspection, performance tuning, debugging, and white-box testing, but is not interested in modification [HC99].¹⁴⁰ According to Möller et al., glass-box distribution is quite common in the embedded systems market [MAFN04].¹⁴¹

From a compositional reuse perspective, black-box and gray-box distribution allows only black-box reuse whereas white-box distribution enables all forms of reuse (i.e., black/white/glass-box) [Pri93].

Many discussions revolve around the question whether components of commercial origin (i.e., COTS) mandate black-box distribution or not. However, COTS and black-box distribution are two independent properties as reflected in this taxonomy: origin (cf. Section 4.2.1) vs. distribution form (cf. Section 4.2.2). Interestingly, more recent definitions of COTS have widened its meaning to include open source software.¹⁴²

COTS and black-box
distribution

4.2.3 Customization Mechanisms

Whereas components might be reused as-is without any modification, in practice some tailoring or customization¹⁴³ (either by the developers or users) takes place. In the context of commercial OTS components, Carney et al. note that “the ‘-OTS’ implies that the software item can be used with little or no modification. But at least some modification, minor or otherwise, is needed for most classes of commercial software”[CHP00].

Support for customization can be divided into non-programmatic and programmatic customization mechanisms:

¹⁴⁰An interesting example of glass-box use is provided by Kvale et al., who first inspect the Java code of open source components with the goal to then customize them at the byte code level via AspectJ aspect weaving [KLC05].

¹⁴¹Indeed, Microsoft has made available the sources of Windows CE under its Shared Source Initiative (<http://www.microsoft.com/presspass/features/2003/apr03/04-09wince.asp>).

¹⁴²For instance, Torchiano and Morisio’s definition: “A COTS product is a commercially available or open source piece of software that other software projects can reuse and integrate into their own products” [TM04].

¹⁴³Merriam-Webster’s definition of *to customize* is as follows: “to build, fit, or alter according to individual specifications” (qtd. in [NP04]).

non-programmatic: Non-programmatic customization is accomplished, for example, by giving command-line switches, by editing parameters in startup and configuration files, or with direct manipulation at the GUI level.

There are many examples of non-programmatic customization. Unix filters and batch-processing tools such as compilers are typically customized via command-line switches. X11 applications can be customized (e.g., fonts, colors, and labels) by changing resource specifications via dedicated tools such as `xrdb` and `editres`. ActiveX controls have properties (typically with default values) that can be modified at design-time or run-time [vOB02, p. 339]. Lastly, GUI-based applications such as Microsoft Office allow user to customize GUI elements such as menus and buttons interactively via check-boxes and drag-and-drop. Rational Rose allows interactive customization of diagram elements [GW99]. Sophisticated non-programmatic customization can exhibit similar characteristics as found in declarative (visual) domain-specific languages [Wil01] [vDKV00] [Sal98].

Note that non-programmatic customization enables, selects, or rearranges predefined component functionality—it does not add or create new functionality.

programmatic: Programmatic customization involves some form of scripting or programming language that allows the modification and extension of the component's behavior. For black-box components, extensions are constrained by the functionality offered by the programmatic customization mechanisms.

Programmatic customization of a component can be accomplished via an application programming interface (API) and/or a scripting language:

API: Most component APIs have to be programmed in C or C++. In this case, the API is described in one or more header files. An example is the FrameMaker Developer's Kit (cf. Section 5.3.2). APIs can also take the form of object-oriented frameworks (e.g., Eclipse) and libraries (e.g., IBM VisualAge). Components that support customization via COM can be programmed in any COM-aware language such as Visual Basic, C++, and C#.

The API of a component can be quite large and complex [VGD96]; for instance, Boehm and Abts claim that Windows 95 has roughly 25,000 entry points [BA99], and Tallis and Balzer report that Microsoft Word has over 1,100 unique commands [TB01]. Component APIs vary in the extent that they permit clients to affect the component's internal state [DeL99, sec. 3.5]. At the one extreme an API may be read-only and expose very limited information about its internal state; at the other extreme, there may be almost no restriction (e.g., allowing a "quit application" operation). The Together Open API has a high-level read-only interface as well as lower-level interfaces that allows state modifications. The APIs for plug-ins often place well-defined restrictions on the internal state that can be manipulated.

scripting: Components can also offer a scripting language to simplify programmatic customization. Sometimes, scripting is offered as an alternative to a traditional API.

A prominent example of a scriptable component is the Emacs text editor, which can be customized by programming in Emacs Lisp [Sta81] [BG88]. Similarly, the AutoCAD system can be customized with AutoLISP [GN92], and UML Studio¹⁴⁴ has a dedicated LISP dialect called PragScript [JWZ02]. Microsoft products typically support Visual Basic scripting. Adobe GoLive can be customized with HTML files containing embedded JavaScript (cf. Section 6.6). Scripting is also used to simplify game development. UnrealScript is a dedicated scripting language for the distributed Unreal Tournament online game [MDW05].

source code modification: In case of a white-box component, its source code can be directly modified in order to achieve the required customization. Typically, source code is only modified if a customization cannot be realized via scripting or API programming.

Morisio and Torchiano explicitly distinguish between required modification (i.e., necessary customization mechanisms to build a certain component-based system), and possible modification (i.e., supported customization mechanisms by the component) [MT02]. This distinction can be of importance. For example, whereas white-box components offer source code modification, component-based systems development rarely makes use of this option (because of future maintenance problems).

Carney and Long provide a classification to which degree a component's code can or must be changed [CL00a]. They distinguish: (1) very little or no modification, (2) simple parameterization, (3) necessary tailoring or customization, (4) internal revisions to accommodate special platform requirements, and (5) extensive functional recoding and reworking. The first two modifications are non-programmatic. The third involves programmatic customization via API or scripting. The last two are modifications not anticipated by the original component vendor and require modification of the source code.

Carney and Long's
classification

Another classification introduces three types [CHP00]: installation-dependent components (which require the user to take some actions before the software can operate, for instance, in the form of setting environment variables or providing license information), tailoring-dependent products (which require a considerable amount of initialization data such as the definition of schema information or business rules), and modified products (which occurs "when a customer asks a software vendor for (or, conceivably, makes himself) some ad hoc, to-order alteration of a commercial product"). With the last type, the component's modification is a change in functionality or behavior not originally intended by its vendor.

Carney et al.'s
classification

¹⁴⁴<http://www.pragsoft.com>

4.2.4 Interoperability Mechanisms

Wegner states that “interoperability is the ability of two or more software components to cooperate despite differences in language, interface, and execution platform” [Weg96]. Interoperability among components can be achieved, for instance, by passing control, sending messages, or sharing data. Note that customization (cf. Section 4.2.3) addresses functionality that is added to the component itself, whereas interoperability refers to the use of the component’s functionality by other components (i.e., its clients).

The following classification of component interoperability is based on research in integration of software engineering environments [BP92] [TN92] [Was90]:

no interoperability: In this case, the component does not support any explicit, programmable interoperability mechanism. In this case, the only form of interoperability that the component provides is via its user interface. This can be the case with legacy terminal-based business applications. Even though the user interface is meant to be used interactively by end users, such components can be made interoperable via ad hoc wrapping techniques. For example, batch applications and textual user interfaces can be wrapped with pseudo ttys (on Unix) or dedicated (screen scraper) tools [Sam97] [VD97] such as Expect.¹⁴⁵ There are also approaches that intercept and synthesize GUI events [GBP02] [BH96].

interface interoperability: Interface interoperability means that the components offers some kind of programmable interface to enable interoperability with other components. The following interface types can be distinguished:

data: Data interfaces provide a simple API to access component-specific data. Sametinger distinguishes between textual, specific file (i.e., binary), and database input/output [Sam97, sec. 9.1]. Examples of data formats that are supported by components are Word documents (textual or binary), SVG files (XML-based), and RSF graphs (textual).

Unix’s pipe-and-filter architecture uses textual input/output to achieve data interoperability. Another example of data interoperability is provided by early IDEs such as PCTE and Centaur that employed a central data repository. The repository connects the various tools that are part of the IDE by allowing them to store their intermediate results for use by other tools [Rei96] [Rei90].

control: Control interoperability involves an interface that allows a component to pass a message to another component, or to access state that is shared among components (e.g., via global variables or shared memory). This functionality is provided by wiring standards such as CORBA and COM (as well as other message-oriented and object-oriented middleware). The ToolBus of the ASF+SDF Meta-Environment (cf. Section 4.1.2) is another example of a

¹⁴⁵<http://expect.nist.gov>

control-based interoperability infrastructure. Similarly, the Common Desktop Environment (CDE) offers a message brokering system called ToolTalk [Kra98]. Succi et al. use JavaSpaces tuple spaces to exchange messages between tools [SPLY01].

There are many examples of IDEs that are primarily based on control interoperability such as HP's Softbench, Sun's ToolTalk, and Reiss' Field and Desert environments [Bro93] [Rei02]. With these IDEs, a message server allows tools to register their interest in certain events that other tools might send to the server [HOT00] [Rei90]. Tools can report events to the server which are then distributed to the registered tools.

presentation: Presentation interoperability refers to a seamless interoperation at the user-interface level. Components are tightly integrated and have a common look-and-feel. An Example of components that support presentation interoperability are compound documents such as OLE and OpenDoc (cf. Section 4.1.2). In order to achieve such tight integration some kind of wiring standard is necessary.

IDEs such as Eclipse, IBM VisualAge, and Microsoft Visual Studio have extensible architectures that makes it possible to add components (i.e., plug-ins) seamlessly.

The three interoperability types introduced above are usually inclusive. Presentation interoperability typically has to support control and data interoperability in order to achieve seamless integration of components. Components with control interoperability also support some kind of data interoperability.

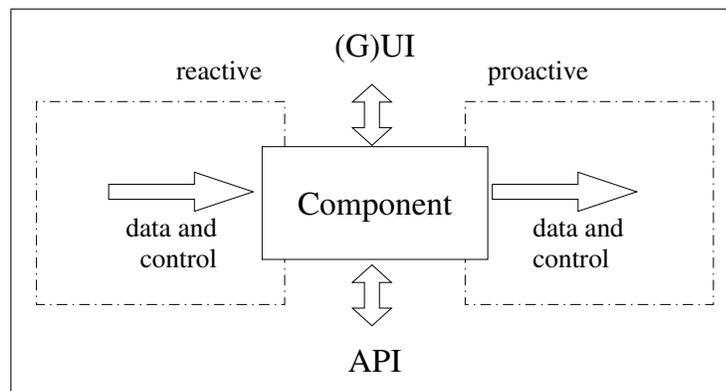


Figure 14: Proactive vs. reactive parts of a component ([EJB04])

Egyed et al. make a useful distinction of components, classifying them as reactive or proactive [EJB04]. Reactive components react to user input and other outside stimuli (e.g., made via the user interface or through API calls). Input can be data (e.g., obtained by

Proactive vs. reactive
components

reading from a file), or control messages (e.g., a mouse event). A reactive component is passive in the sense it does not initiate interactions with other components (and hence need not have knowledge of its surrounding components); it waits for service requests [EB01]. In contrast, a proactive component notifies other components of changes in its state (e.g., via event notification). An example of such a state change could be the selection of an element in the GUI or the user's activation of a button or menu item. A reactive component can use both data and control to notify other components of its state change. Figure 14 illustrates the difference between reactive and proactive. Components often have both reactive and proactive characteristics. However, OTS products and IDEs often have deficiencies in their realization of proactiveness, lacking, for instance, sophisticated event notification; this is the case with Microsoft Office [GB99] and Rational Rose [EB01].

Similar to my classification, Davis et al. distinguish between data and control interactions among components [DGP02] [DGP⁺01]. They survey and summarize a number of characteristics for both data and control. Examples of characteristics for data are

Data and control
characteristics

- data topology (hierarchical, star, linear, or arbitrary) [SC97]
- supported data transfer (explicit, implicit, or shared)
- data storage method (repository or local)
- data scope (restricted or unrestricted) [KCBA97]
- data mode (passed, shared, multicast, or broadcast) [SC97]

For control, examples of characteristics are

- control topology (hierarchical, star, linear, or arbitrary) [SC97]
- control structure (single-threaded, multi-threaded, or decentralized) [YBB99]
- control scope (restricted or unrestricted) [KCBA97]
- synchronicity (lockstep, asynchronous, or synchronous) [SC97] [YBB99]

Software architectures use both components and connectors to describe software systems. Connectors are responsible for the transfer of control or data among components. Hence, they can be used for component integration [GLGB05]. Mehta et al. introduce a connector classification framework whose characteristics overlap with the ones introduced above [MMP00]. Part of the framework are connector types such as procedure call, events, data access, streams, et cetera. A procedure call can be explicitly or implicitly invoked and can be synchronous or asynchronous; events can be distinguished by their notification scheme (e.g., polled or publish/subscribe); data access can be thread-specific, process-specific or global, and persistent or transient; streams can be binary or N-ary (i.e., multiple senders and/or receivers), synchronous or asynchronous, and stateful or stateless.

Connectors

4.2.5 Packaging

Components can be packaged in different ways. Packaging is the form in which the component is used, not the form it is distributed in [MT02]. For example, a white-box component is distributed with source code, but can be packaged via compilation into a stand-alone executable or a library. The packaging of a component is characterized as follows:

standalone component: Such a component is a standalone program, application, or tool that can be directly executed. The component is standalone in the sense that it can be used without prior integration or customization. Examples of this type of component are OTS products, IDEs, and domain-specific tools (cf. Section 4.1.2). Standalone components can range from huge (e.g., Microsoft Office) to small (e.g., Unix's `echo`¹⁴⁶).

The taxonomy further distinguishes standalone components whether they are interactive or batch systems [DZS97]:

interactive: Interactive (or conversational¹⁴⁷) components have a graphical or textual user interface so that a human can participate in the computation. In fact, often the human drives the computation, the system reacting to the user's input (e.g., from the keyboard or mouse). Microsoft Office is an example of an interactive component.

batch: Batch components make a complete program run without human intervention. Input is specified before program start (e.g., in the form of input files or streams). The behavior of the system is typically specified with command-line arguments and configuration files (i.e., non-programmatic customization). Unix filters are examples of batch components.

non-standalone component: A non-standalone component has to be integrated and/or customized before it can be executed. A typical example are linkable components such as object modules or (class) libraries, which must be linked statically or dynamically with other components to obtain an executable [YBB99]. Other examples of this component type are components from application generators (e.g., a scanner generated by `lex`), OTS components (e.g., ActiveX controls),¹⁴⁸ compound documents, (object-oriented) frameworks, classes, and functions (cf. Section 4.1.2).

The packaging of a component is of importance for interoperability and reuse. For example, Unix provides sorting functionality both with the `sort` filter (which is a standalone

¹⁴⁶The NetBSD version of `echo` consists of about 30 lines of code (excluding license and version identifiers) [Spi03, fig. 2.1].

¹⁴⁷The *IEEE Standard Glossary of Software Engineering Terminology* defines conversational as “pertaining to an interactive system or mode of operation in which the interaction between the user and the system resembles a human dialog” [Ins90].

¹⁴⁸Maurer says of components such as ActiveX controls, “although binary components are essentially independent programs, they cannot be loaded and run like an ordinary program. To be useful they must be incorporated into a larger application” [Mau05].

component), and the `qsort` C library call (which is a non-standalone component). RCS, a tool for version control, is implemented as a set of Unix filters (`rCS`, `co`, `ci`, etc.), but now also offers a C/C++ and Java API.¹⁴⁹

Dusink and van Katwijk distinguish between active and passive components [DvK87] [Sam97, sec. 9.2.3]. Active components run on their own and use operating system services as a means of interoperation (e.g., shared memory or message passing). Passive components are functions, classes, and modules, which are included in a software system by linking or directly including their source code. There seems to be a correspondence between Dusink and van Katwijk’s active and passive components, and this taxonomy’s standalone and non-standalone components, respectively.

Active and passive
components

4.2.6 Number of Components

In contrast to the previous criteria (cf. Sections 4.2.1–4.2.5), which describe components, this criterion characterizes a component-based system. A component-based system can be distinguished by the number of components that constitute the system:

single: These systems use a single component on which they heavily depend. Single-component systems often rely on a powerful standalone component (e.g., Microsoft Office) or IDE (e.g., Eclipse). Further examples of single-component systems are given in Section 5.2.

Carney distinguishes single-component systems into turnkey systems and intermediate systems [Car97]. Turnkey systems¹⁵⁰ use a single OTS component such as Microsoft Office or Mozilla, on which they heavily depend and which is used out-of-the-box without programmatic customization. Intermediate systems are also built on a single OTS component, but also “have a number of [programmatically] customized elements specific to the given application.” In this taxonomy customization is addressed by a separate criterion (cf. Section 4.2.3).

multiple: These systems are (primarily) composed of multiple components, possibly from different vendors and having different characteristics.

Often multiple-component systems have two to four core components which realize a significant amount of the system’s functionality. But there are also systems that are composed of more components. For example, Morisio et al. mention a system comprising 13 OTS components [MSB⁺02], and NASA’s Hubble Space Telescope command and control system uses even more than 30 COTS/GOTS components [PR02]. Traditional information management systems typically consist of three layers: database, application logic, and user interface. Each layer can be realized as a component.

¹⁴⁹http://www.aicas.com/rce_en.html

¹⁵⁰Presumably the term *turnkey system* wants to evoke the notion of a system that is as readily up-and-running as an automobile that just requires to turn the key in the ignition (and possibly changing the positions of the seat and mirrors) before one can drive away.

Multi-component systems can be further distinguished as being composed of homogeneous or heterogeneous components:

homogeneous: A system is homogeneous if its components adhere to the same wiring standard. For example, all Microsoft Office products support COM and OLE, and all Eclipse plug-ins use the same extensibility mechanism to integrate into the platform.

Galileo (cf. Section 5.3.3) is an example of a tool that is based on three homogeneous components: Microsoft Word, Excel, and Internet Explorer. The SHriMP reverse engineering tool is an example of a homogeneous system based on JavaBean components [BSM02] [Bes02].

If a component-based software-engineering tool uses popular standards such as JavaBeans or COM, it can incorporate off-the-shelf components that have been developed independently by third parties. However, few researchers have so far taken advantage of procuring items externally at component markets (cf. Section 4.1.4). An exception is the SPOOL reverse engineering environment (implemented with Java 1.1 and Swing 1.0.3), which visualizes HTML code with a commercial JavaBean component, the ICEBrowser [KSRP99].

Instead of relying on an established wiring standard, a software system can implement its own. For instance, the Serendipity-II software engineering environment uses a component-based approach for tool-building that defines its own component framework, called JViews [GMH00]. JViews, which is based on JavaBeans, extends JavaBeans with abstractions for managing multiple views, persistency management, event broadcasting, et cetera. Whereas the definition of a proprietary component framework such as JViews has the advantage that it can provide dedicated abstractions for the problem domain, it precludes the use of third-party components.

heterogeneous: A system is heterogeneous if it mixes components based on different wiring standards, interoperability standards, or architectures. Such systems are often composed of OTS products from different independent vendors (e.g., Rational Rose and Matlab/Stateflow [EB01]), possibly also integrating proprietary, system-specific components [EMP05].

I give a few examples of heterogeneous multiple-component systems that are described in the literature. Garlan et al. describe the building of a software design environment, Aesop, consisting of the following components: (1) a GUI framework (InterViews), (2) an event-based tool-integration mechanism (HP Softbench), (3) an RPC mechanism, and (4) an object-oriented database [GAO95]. Staringer provides another multi-component tool-building experience, integrating (1) a GUI builder (NeXTStep Interface Builder), (2) a spreadsheet application (Lotus Improv), (3) Mathematica, and (4) a relational database (Sybase). Bass et al. report on integrating multiple components

such as Web browsers (Netscape Navigator), video-conferencing applications (Communique), and graphic manipulation software via a CORBA message bus [BCK98, ch. 18]. ANUBIS, a multi-user librarian system is implemented using a middleware product (IBM Websphere MQ), an open source database (MySQL), and Amazon.com Web Services 2.0 as online data source [FGBS04].

Multi-component systems can have a significant amount of *glue code*,¹⁵¹ whose purpose is it to bind the (often heterogeneous) components together—in Carney’s words, “sometimes cleanly and sometimes crudely” [Car97]. Glue code realizes functionality such as transferring of control flow to a component; bridging of incompatibilities between components; and handling of errors and exceptions [VD97]. Whereas components use typed high-level programming languages [Ous98], glue code is often written with a scripting language such as Perl, Tcl, Visual Basic, or Unix shells (e.g., sh) [Vin02] [Ous98] [Hal99] [VD97] [Pri93]. For example, ActiveX components are often written in C++, but glued together with Visual Basic [Mau05].

Glue code

For multiple-component systems an important consideration is which component holds the main thread of control or else implements the control loop [GAO95]. The component that implements the control loop is the system’s “driver” and in charge of delegating control to other components. Components are often difficult to make interoperate if more than one components in the system assumes that it is the driver. A standalone component has its own thread of control by definition. Some non-standalone components such as libraries are passive in the sense that they do not have their own control thread—control is passed to them with an explicit invocation from the client. However, other non-standalone components such as GUI frameworks have their own control loop.

Main control thread

Similar to my classification, Wallnau distinguishes OTS-based systems into OTS-intensive systems (which integrate many OTS products) and OTS-solution systems (which consist of one substantial OTS product customized to provide a “turnkey” solution) [Wal98].

Wallnau’s classification

Boehm et al. classify component-based system according to the dominating development activity. Based on their experiences, they identify four major project types [BPY⁺03b]: (1) assessment intensive projects (which focus on finding a feasible set of components having capabilities that require little or no programmatic customization); (2) tailoring intensive projects (whose main efforts lie in programmatically customizing one or only a few components to realize most of the system’s capabilities); (3) glue-code intensive projects (which require a significant amount of glue code design and implementation involving multiple components); and (4) non-OTS intensive projects (in which most of the system’s capabilities are provided by traditional custom development). The latter project type only marginally makes use of components and thus is outside of the scope of this dissertation. The definitions of the other three types draw from criteria already introduced in

Boehm et al.’s classification

¹⁵¹Glue code is sometimes called glueware, glue logic, or binding code.

my taxonomy, namely the number of components and customization mechanisms.

4.2.7 Other Taxonomies

There are a number of taxonomies and frameworks to characterize and classify components. In the following, I give an overview.

In his book, Sametinger gives a brief survey of component classifications [Sam97, sec. 9.2]. The discussed classifications reflect the diverse notion of component. Some classifications are targeted at white-box components such as Ada packages (Booch [Boo87]), or are based on classifying programming-language features (Wegner [Weg89]). Other classifications use broader dichotomies such as fine-grained vs. coarse-grained (Kain [Kai96]), specification vs. implementation (Kain), and active vs. passive (Dusink and van Katwijk [DvK87]).

Component
classifications

Attribute	Possible values
user interface	command-line interface, graphical interface
data interface	textual I/O, specific file I/O, database I/O
program interface	textual composition, functional composition, modular composition, object-oriented composition, subsystem composition, object model composition, specific platform composition, open platform composition
component platform	hardware, operating system, programming system, libraries/frameworks, programming language

Table 3: Summary of Sametinger’s reusable software components taxonomy ([Sam97, fig. 9.1])

Sametinger has developed his own taxonomy, distinguishing user interface, data interface, program interface, and component platform (cf. Table 3). The user interface of a component can be a command-line or graphical user interface. The data interface characterizes a component’s input/output (I/O), which is distinguished as textual I/O, specific file I/O, and database I/O. A component’s program interface characterizes how functionality is reused. Lastly, a component’s platform characterizes the environment that is necessary for the component to operate.

Sametinger’s
component
taxonomy

For OTS components, Morisio and Torchiano provide a comprehensive characterization framework based on evaluation of previous work (e.g., [Car97] [YBB99] [CL00a] [MT02]). Table 4 provides a summary of the characterization framework.

OTS classifications

Sassi et al. propose another characterization framework for OTS components in the context of an OTS-based development environment [SJG03]. The framework groups characteristics into: general (e.g., cost, date of first release, and change frequency), structural (e.g., name and number of services), behavioral (e.g., pre/post-conditions and state-transition diagrams), architectural (e.g., component type and architectural style), quality of service

Category	Attribute	Possible values
source	origin	in-house, existing external, externally developed, special version of commercial, independent commercial
	cost and property	acquisition, license, free
customization	required modification	minimal, parameterization, customization, internal revision, extensive rework
	possible modification	none or minimal, parameterization, customization, programming, source code
	interface	none, documentation, API, object-oriented interface, contract with protocol
bundle	packaging	source code, static library, dynamic library, binary component, stand-alone program
	delivered	non delivered, partly, totally
	size	small, medium, large, huge
role	functionality	horizontal, vertical
	architectural level	OS, middleware, support, core, user interface

Table 4: Summary of Morisio and Tarchiano's OTS components characterization framework (adapted from [MT02])

(e.g., non-functional properties and possible modification), technical (e.g., conformance to standards), and usage (e.g., similar components and use cases).

Torchiano et al. present a list of OTS characterization attributes based on the proposal from students made in a fifth year course [TJ03] [TJSW02]. Examples of attributes are product maturity, market share, reliability, hardware requirements, product support, documentation, modifiability, change frequency, license type, and cost of use.

4.2.8 Discussion

There are many existing taxonomies. Why did I propose a new one? As will become apparent in the next chapter, the goal of my taxonomy is to characterize components specifically for tool-building. Other taxonomies typically have a broader goal such as to aid in the evaluation of any kind of component in any application domain. However, my taxonomy has been influenced by and reflects previous work of these other taxonomies, and thus is well-grounded.

Goal of taxonomy

My taxonomy combines a number of interesting properties. It is small because I have focused the taxonomy on a relatively small number of criteria, which are highly relevant for tool building. Furthermore, each criterion is rather coarse-grained, having at most three alternatives. As a result, the taxonomy omits a number of less relevant criteria such as market share [TJSW02], execution platform [Sam97], and component size [MT02]. However, such component characteristics can have an impact on the functional or non-functional requirements of component-based systems. For example, an OTS product's market share and execution platform can have an impact on the adoptability of a tool based on the OTS product.

Properties of
taxonomy

It typically is a straightforward task to assign the taxonomy's criteria to a particular component or component-based system. In other words, the taxonomy uses *orientation level* criteria, which "paint an overall picture of the component" and whose "values can be often gleaned from developer documentation" [DGP02]. Hence, classifying a component with my taxonomy is a lightweight activity that should not pose any difficulty.

A taxonomy should be extensible so as "to permit new discoveries to be cataloged and more detailed study in specific areas" [PBS93, sec. 6]. My taxonomy is extensible by defining new criteria or by further refining existing ones. For researchers who wish to extend the taxonomy, I have identified criteria with potential for a more fine-grained characterization.

Whereas this taxonomy has been created primarily for the purpose to characterize components for component-based building of reverse engineering tools, I believe that its properties make it also applicable for other classification purposes that go beyond this dissertation's context. For example, it enables a structured comparison of components that can be used as input for a component selection activity (cf. Section 4.1.6). Furthermore, the taxonomy can identify characteristics that impact the system's requirements and design, as well as the development process. The taxonomy can help also to reason about quality attributes of certain types of components and component-based systems. For example, using the criteria defined in my taxonomy one might look at testability of components and conclude that the follow kinds of distribution forms are increasingly difficult to test: white-box, glass-box, and black-box. Similarly, one might conclude that the following types of systems are increasingly difficult to maintain: single-component, homogeneous multiple-component, and heterogeneous multiple-component.

Other uses of
taxonomy

4.3 Summary

Software components play an important role in this dissertation because they are the building blocks that I strive to use for assembling reverse engineering tools. This chapter introduced software components and related topics such as software reuse, component markets, product lines, and component-based software engineering. To provide a better understanding of what a software component is, this chapter also introduced definitions, and provided extensive examples covering the entire spectrum.

The second part of this chapter consists of a taxonomy that targets component-based

tool building. The taxonomy can be used to characterize components and component-based systems—to put it differently, it lays out the *design space*. In the next chapter, the taxonomy will be used as an instrument to precisely define the points in the design space that this dissertation investigates for building of reverse-engineering tools.

5 Building Reverse Engineering Tools with Components

“Most applications devote less than 10% of their code to the overt function of the system; the other 90% goes into system or administrative code: input and output; user interfaces, text editing, basic graphics, and standard dialogs; communication; data validation and audit trails; basic definitions for the domain such as mathematical or statistical libraries; and so on. It would be very desirable to compose the 90% from standard parts.”

– Mary Shaw [Sha95]

This chapter—as its title suggests—brings together the two major pillars of this dissertation:

(i) reverse engineering (RE): The chosen **domain** to investigate tool-building is software reverse engineering (cf. Chapter 2). The reverse engineering domain provides constraints on tool-building via functional and non-functional requirements (cf. Chapter 3). RE pillar

In this dissertation, the foci are on two tool component types, namely (graph) visualizers (cf. Section 2.3.4) and fact extractors (cf. Section 2.3.2). They are addressed in the tool component catalog in Sections 5.2.1 and 5.2.2, respectively.

(ii) component-based software engineering (CBSE): The chosen **approach** to follow for tool-building uses software components (cf. Chapter 4) as building blocks to assemble software systems. CBSE pillar

CBSE (cf. Chapter 4.1.6) puts constraints on tool-building. For example, CBSE requires a different process and skill-set compared to traditional software engineering approaches, which build every asset from scratch.

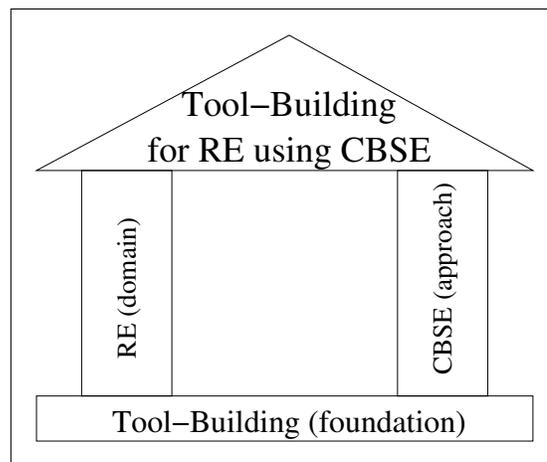


Figure 15: Major themes of this dissertation and how they relate to each other

The underlying **foundation** for both pillars is tool-building, namely (i) tool building in the domain of reverse engineering, and (ii) tool building using the approach of CBSE. The dissertation then unites both pillars as depicted in Figure 15. Hence, a more precise title of the dissertation would be *Tool-Building in the Reverse Engineering Domain using an Approach of Component-Based Software Engineering*.

Tool-building as
foundation

Looking at the two pillars and the underlying foundation, the following observation can be made: On the one hand, there are many publications in the literature about reverse engineering tools and even some tool-building experiences in the reverse engineering domain. On the other hand, there are also many publications about components, reuse, and CBSE. However, there is little work that attempts to close the gap between these two areas.

Closing the gap

The remainder of this chapter addresses this gap by summarizing relevant components and experiences for component-based tool-building. However, before I can start to discuss relevant experiences, it is necessary to understand first what *relevant* actually means. The next section uses my component taxonomy (cf. Section 4.2) to identify the characteristics of the components that this dissertation targets. Section 5.2 then gives a summary of tools that have been built using such components. The catalog shows that component-based tool-building is feasible and already practiced. Section 5.3 provides three representative tool-building experiences by describing the tools' functionalities, implementation, and evaluation. Each tool addresses a different domain and has been implemented by a different researcher or research group. My perspective on these tools is given at the end of each tool description. In Section 5.4, based on the catalog and the three sample tool-building experiences, I discuss the current state of component-based tool-building.

5.1 Characteristics of Targeted Components

“An especially promising trend is the use of mass-market packages as the platforms on which richer and more customized products are built. . . . The want ads in computer magazines offer hundreds of Hypercard stacks and customized templates for Excel, dozens of special functions in Pascal for MiniCad or functions in AutoLisp for AutoCad.”

– Frederick P. Brooks [Bro95, p. 285]

The design space for components and component-based systems is large, having many dimensions. I have introduced the component taxonomy (cf. Section 4.2) to focus and structure this design space. The introduced criteria (e.g., component origin) are the chosen dimensions in the design space. The criteria's characteristics (e.g., internal vs. external origin) define the discrete¹⁵² values within the design space.

Design space

This section uses the taxonomy's criteria to narrow down the design space to the components that I want to further explore as candidates for component-based tool-building—I refer to this restricted design space as the *target design space*, which is defined as follows:

Target design space

¹⁵²These discrete values are often an idealization of a continuum without clearly defined boundaries. For example, component-based and custom software systems are often treated as a dichotomy, but concrete systems actually occupy places on a continuum with the dichotomy marking the two extremes [HTT97].

origin (cf. Section 4.2.1): The targeted tool-building components are **external**. This means that they have been developed independently without having a particular (reverse engineering) tool already in mind. Thus, their use for tool-building can be seen as unanticipated component reuse.

External components can have many sources. Typical sources in the context of reverse-engineering tool-building are OTS products such as Microsoft Office and FrameMaker, and ROTS tools such as Rigi and AT&T graphviz. Since the component's origin is not internal, tool-builders are more dependent on the component vendor, and have limited influence on the component's functional and quality attributes.¹⁵³ On the other hand, external components are immediately available for reuse, and are further enhanced and maintained by the vendor. In fact, tool-building with external components exhibits many of the risks and benefits that are associated with CBSE in general (cf. Section 4.1.6).

Using OTS components can lead to vendor lock-in,¹⁵⁴ which Brown et al. discuss in their AntiPatterns book [BMMM98]. Vendor lock-in occurs if “a software project adopts a product technology and becomes completely dependent upon the vendor's implementation.” As a result, the vendor's upgrades drive the software project's maintenance cycle. Interestingly, the authors state that “the Vendor-Lock-In AntiPattern is acceptable when a single vendor's code makes up the majority of code needed in an application.”

Vendor lock-in

distribution form (cf. Section 4.2.2): My tool-building approach treats components as either **black-box or glass-box**. This means that components are used without modifying their source code. This is even the case when the source is actually available.

If the source code of the component is available, it can be treated as black-box, glass-box, or white-box. If the component can be easily integrated and operates as expected there is no need to spend time and effort in investigating its source code. Popov et al. emphasize that

White-box as
black-box

“we consider together commercial-off-the-shelf [i.e., black-box] and non-commercial off-the-shelf [i.e., white-box] . . . components: the difference is not significant in our discussion. Even when the source code is available, it may be impossible to make use of it—its size and complexity (and often poor documentation) may deny the system integrator the advantages usually taken for granted when the source code is available” [PSK⁺04, fn. 1].

¹⁵³Conversely, it can be also a problem if the client has too much influence on the vendor. Baker relates the following experience made at Boeing: “For many companies, doing business with Boeing can be like a deer doing business with a tour bus at night. They become fixated on Boeing's needs and forget about their marketplace. We worked very hard with our vendors on this point. We insisted that they not do anything to meet our needs that would not be part of their standard product offering. It also had to offer value to other customers for them to do so” [Bak02]. It is unlikely that academic researchers will find themselves in this position.

¹⁵⁴<http://www.antipatterns.com/vendorlockin.htm>

For example, even though the source code of Rigi is available on request, few tool-builders ask for it. Presumably, they are able to use the component as-is or customize it via its Tcl scripting interface, thus having no need to inspect or modify Rigi's internals, which are coded in C/C++. Another example of a white-box component that is used by many researchers as-is is AT&T graphviz.

Regarding open source software, Wang and Wang caution that “while OSS is generally considered highly customizable and extensible—as the source code is publicly available—you must take into account the complexity of the effort to make such modifications at the source level” [WW01].

It can be beneficial to treat a white-box component as glass-box, for instance to ease integration, customization, and debugging. The Eclipse framework, implemented in Java, is a white-box component. Inspecting the source code is a crucial prerequisite for understanding and extending Eclipse.¹⁵⁵ However, developers who are using the framework never modify the framework itself. For one thing, maintaining the changes would be difficult because of the constant evolution of Eclipse.

White-box as
glass-box

To summarize, a white-box component's sources need not be modified, and there are good reasons for treating it as black-box or glass-box.

customization mechanisms (cf. Section 4.2.3): My tool-building approach assumes that components are customized via either **API programming or scripting**. Grafting of reverse engineering functionality on top of an existing component requires some programming effort and cannot be accomplished solely with non-programmatic customization.

Morris et al. identify five patterns of successful OTS-based systems; one of these patterns, *Avoid COTS Modification*, cautions that

“any tailoring or modification to a product that causes deviation from the ‘out-of-the-box’ capability and configuration potentially increases maintenance effort across the life of the system. Project management should consider the type of modification or tailoring required, the level of skill and engineering needed to maintain the system (initially and for each product upgrade), and the likelihood that supplied tailoring mechanisms will change over time in deciding what form of modification or tailoring to allow” [MAB03].

Customization should be accomplished without modifying the component's source. Modifying the source indicates that the customization is not anticipated by the component vendor. In the following, *modification* of a component refers to a change in

Customization vs.
modification

¹⁵⁵In practice, Eclipse programmers look extensively through code to find Eclipse-specific programming idioms and to scavenge pieces of code. This is also apparent in one of the plug-in development “rules” proposed by Gamma and Beck [GB04, p. 40]: “Monkey See/Monkey Do Rule: Always start by copying the structure of a similar plug-in.”

functionality or behavior not originally intended by its vendor [CHP00], whereas *customization* of a component refers to the use of a component’s customization mechanisms, which constitutes an intended use by definition. Modification of a component often has negative consequences for evolution. Carney et al. provide a number of recommendations for the maintenance of OTS systems [CHP00]. They caution that modification “is widely held to be a poor practice and is the likely occasion for maintenance issues to arise.” Whereas both customization and modification can have a negative impact for maintenance, (larger-scale) modification can have far more negative consequences. Hence, modification should be avoided, whereas customization is a necessary prerequisite for my tool-building approach.

Besides programmatic customization via API or scripting, tool-building can also take advantage of non-programmatic customization. However, in practice this only constitutes a small part of the development effort. Also, it is often desirable or necessary to use programmatic customization even if the desired effect could be accomplished via non-programmatic customization as well. For example even though Microsoft Word’s GUI can be customized interactively (e.g., modification of the toolbar), programmatic customization has been used to guarantee that the user is presented with a certain toolbar after startup.

interoperability mechanisms (cf. Section 4.2.4): The targeted tool-building components have to provide a means for programmatic **interface interoperability**, regardless whether they support **data, control, and/or presentation** interoperation. Interoperability is less of an issue if the component-based system consists of a single component only.

Generally, presentation interoperation is most desirable from the user’s point-of-view because it can achieve a seamless access of all reverse-engineering functionality provided by a tool. However, this form of integration can be difficult or impractical to accomplish. Instead, data integration can be used, which often achieves an integration between components that is satisfactory to the tool’s users. For example, to display (non-interactive) information such as passive documents, graphs, and reports, in a dedicated viewer such as Word, Excel, Emacs, or dot, simple file-based data interoperability without (programmatic) customization is often sufficient. Rigi realizes data interoperability between its extractor components and the graph visualizer with its file-based RSF exchange format (cf. Section 2.3.5).

As already discussed, components can have a wide range of interoperability mechanisms and wiring standards. Among the problems that OTS-based development efforts have to address is the fact that “COTS products make implicit assumptions about the way the product will interact with other products” [TAB03]. Thus, it is unlikely that two independently developed components will interoperate seamlessly out-of-the-box. In the literature, *architectural mismatch*¹⁵⁶ denotes problems

Architectural
mismatch

¹⁵⁶Some authors refer to architectural mismatch as interoperability conflict [DFGK03] or packaging mismatch

in integrating component-based systems. It occurs because the involved components make incompatible assumptions, typically at the architectural level [GAO95]. As a result, integrated components will not properly compile, link, or execute [BCK98, sec. 15.3]. Architectural mismatch is less of a problem for data interoperation compared to control and presentation interoperation.

In contrast to the popular opinion that architectural mismatches are the primary cause of integration problems [DeL01][EMG00] [YBB99], Torchiano and Morisio report that for component-based systems that have adopted a well-established generic architecture and technology (e.g., the Linux-Apache-MySQL-PHP architecture), components' lack of standard compliance can constitute a bigger integration problem [TM04, Thesis T2]:

Standard
non-compliance

“Problems arise when the architecture’s components don’t fully or correctly support the standards or when different components support different and incompatible versions of a standard. In one project, two COTS products supported slightly different versions of CORBA, making it impossible for them to interact.”

packaging (cf. Section 4.2.5): The packaging of the targeted component is **standalone**.

A standalone component can be either **interactive or batch**. Most coarse-grained components, products, and tools are standalone.

A standalone component is typically easy to deploy, requiring no explicit compilation or linkage. Often a simple interactive installation procedure suffices. For example, most popular open source products on Linux are conveniently packaged as a binary RPM. RPMs can be easily found using dedicated services such as Rufus.¹⁵⁷

Traditionally, compilers are batch components. Consequently, most reverse engineering extractors are also batch components. In fact, many extractors are based on a compiler (front-end), thus inheriting its packaging. Examples of batch extractors are CPPX [DMH01], gccXfront [PM02] (both based on GNU GCC), and the FORTRESS Java/C++ extractor (based on the EDG front-end) [GPB04]. The Rigi C++ extractor is based on IBM’s VisualAge C++ IDE, but runs the IDE *headless*.¹⁵⁸ in batch mode from the command line [Mar99]. Generally, experience shows that converting batch systems to support interactivity is difficult to achieve [DZS97]. However, some IDE-based extractors take advantage of the incremental nature of the underlying compiler (e.g., SHriMP’s Eclipse-based Java extractor and visualizer, called Creole [LMSW03]).

In contrast, reverse engineering visualizations are often interactive and consequently based on interactive components. Examples of utilized interactive products and tools are Microsoft Office, Adobe FrameMaker, Web browsers, and AT&T’s *dotty*,

[DeL01].

¹⁵⁷<http://www.rpfind.net/>

¹⁵⁸<http://en.wikipedia.org/wiki/Headless>

a programmable graph editor. There are also some non-interactive visualizations, many based on AT&T's `dot` graph layouter.

Interactive components are often active (i.e., they have their own thread of control or else control loop). Müller et al. observe that

“most reverse engineering tools attempt to create a complete integrated environment in which the reverse engineering tool assumes it has overall control” [MJS⁺00].

number of components (cf. Section 4.2.6): With my approach, a reverse engineering tool that has been constructed as a component-based system should consist of one primary component and hence is **close to a single-component** system. In other words, my approach follows Abt's rule of thumb: “Maximize the amount of functionality in your system provided by OTS components but using as few OTS components as possible” [Abt02].

A goal of the tool-building approach is to minimize the number of components and the amount of glue code, and to concentrate programmatic customizations on a single component.¹⁵⁹ I refer to the components that have been chosen for customization as the *host components*. Section 5.2 gives examples of suitable host components for the building of reverse engineering tools.

Host components

Typical examples of single-component reverse-engineering tools are based on IDEs. In this case, an IDE acts as the host component. For instance, Creole is realized as an Eclipse plug-in, using Eclipse's Java API to realize both visualizer and extractor functionality.

Single-component tools

Since a reverse engineering tool has to provide visualizer functionality as well as extractor functionality, it is not always practical or desirable to realize both with a single host component. In this case, visualizer and extractor are based on two different host components. A lightweight and effective approach is to use an exchange format for interoperability. This approach is used, for instance, by REVisio (cf. Section 6.4) and RENotes (cf. Section 6.5). Alternatively, a more sophisticated interoperability mechanism based on control interoperation can be used. This approach has been explored, for instance, with REGoLive (cf. Section 6.6). Often, a reverse engineering tool uses an extractor component as-is (i.e., with no or non-programmatic customization), focusing its effort on programmatic customization of the visualization component. Thus, from a customization perspective, these tools are comparable to a single-component system. This is the case, for instance, for REVisio and RENotes.

Two-component tools

From the perspective of tool development and maintenance, it seems desirable to

Integration complexity

¹⁵⁹These constraints are similar to *tailoring intensive* projects, which are characterized by Boehm et al. as follows: “A tailoring intensive project is where the desired capabilities & priorities are covered by a single (or very few) COTS package(s). The primary effort is on adapting a COTS framework whose existing general capabilities can be customized (i.e., tailored) in a feasible way to satisfy the majority of a system's capabilities or operational scenarios” [BPY⁺03b].

minimize the number of host components because it decreases complexity. Hissam and Carney point out that

“a major complication for any system is the extent of its diversity. As the number of components to be integrated with each other grows, and as the number of combinations increases, then so does the likelihood that failures will occur in one of three possible places:

- in any individual component,
- in any interaction between pairs of components, and
- in the entire system itself” [HC99, sec. 2.3].

If multiple components are used, complexity can be decreased by using a simple, transparent interoperability mechanism. Complexity is also decreased if the components are homogeneous. Such systems often require little or no glue code. In these cases, the system has traits that are similar to a single-component system. Conversely, multiple-component systems based on heterogeneous components typically have to cope with challenging integration problems (caused by architectural and standards mismatches), resulting in nontrivial glue code and wrappers.

Another source of problems is the use of glue code and wrappers to integrate (heterogeneous) components. It is almost a truism that “COTS integration almost never goes as smoothly as planned” [BEA01]. McKinney states that a “model many of us have tried without much success to implement cost-effectively is the integration of disparate COTS packages with glue code” [McK99]. Glue code is often developed ad hoc and turns out to be brittle [BEA01], potentially exacerbating maintenance problems [COR98]. Furthermore, development of glue code can be expensive [BEA01]. One of Basili and Boehm’s hypotheses about COTS-based systems says “the effort per line of glue code averages about three times the effort per line of developed-applications code” [BB01, Hypothesis 6].¹⁶⁰ In contrast, the development effort of a single-component system mainly deals with adding of functionality via programmatic customization, which is often less problematic [TM04] [ABC00].

Glue code

Figure 16 provides a graphical representation of the introduced target design space. The taxonomy’s criteria that fall within the target design space are highlighted with boxed text. It is important to identify the target design space of the components used for tool-building because it clearly identifies the boundaries of the reported experiences and the proposed tool-building approach. Component-based systems that fall outside of the target design space might be not applicable for my proposed tool-building approach.

Besides the identified target design space, one can consider other characteristics as well. For instance, it might be desirable to construct a tool by choosing components with larger size. Sullivan et al. rationalize this as follows:

Component size

¹⁶⁰The hypothesis is grounded in the COCOTS cost model [ABC00].

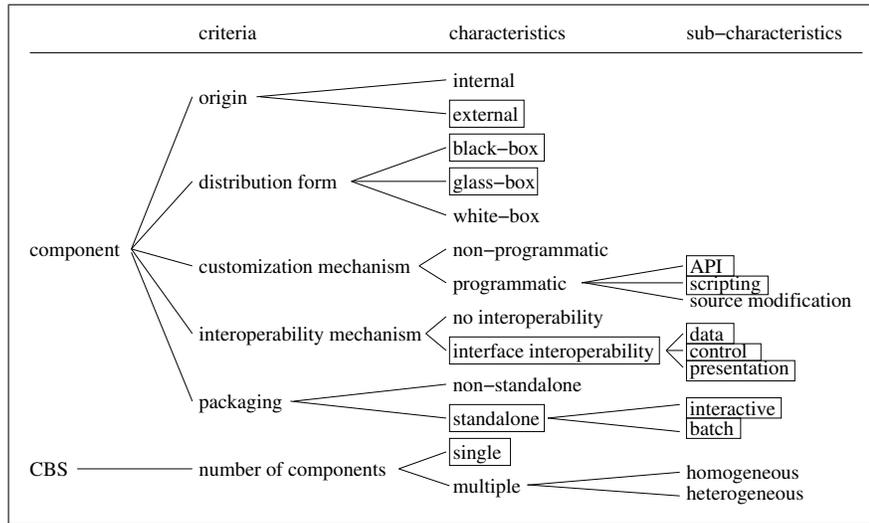


Figure 16: Tool-building target design space

“It can be argued that reuse in which engineers attempt to develop systems by reusing small building blocks does not attack the essence of the problem. . . . Achieving truly significant benefits from component-based reuse would appear to require the reuse of massive components so as to enable large systems (for example, one million lines) to be constructed by straightforward integration of just a few components. With this goal in mind, it is clear that components that average 100 lines in length are too small by about three orders of magnitude” [SKC⁺96].

Striving for a single host component as advocated by my tool-building approach suggest that this component probably has a significant size (e.g., a COTS product such as Excel). However, multiple-component systems might also employ components of comparably small size. For instance, Rigi relies on an external command-line utility to compute spring layouts; the utility’s executable is about 19KB.

5.2 Catalog of Targeted Components

This section briefly describes several systems—primarily from reverse engineering, but also from software engineering and other domains—that have been built using host components that are within the target design space.

In the following, I categorize the host components by the tool component types (cf. Section 2.3) leveraging the host component: visualizers (cf. Section 5.2.1) are discussed first, followed by extractors (cf. Section 5.2.2). Typical realizations of these two tool component types have different characteristics. These characteristics are summarized in Table 5.

Visualizations are often interactive and GUI-based. Jahnke’s classification (Section 2.3)

Tool component
types

Extractors	Visualizers
batch	interactive
command-line	(G)UI
human-excluded	human-aware/centered

Table 5: Characteristics of visualizers vs. extractors

characterizes them as either human-aware or human-centered. In contrast, extractors are often batch style and command-line oriented, classifying them as human-excluded. Extractors and visualizers are often coupled via data integration: The extractor produces facts, which are stored in a repository and then retrieved by the visualizer. A repository can range from a simple text file to a sophisticated relational or object-oriented database. Depending on their realization, reverse engineering analyses often exhibit characteristics that are extractor- or visualizer-inherited. Visualization-based analyses are tightly coupled with the visualizer component. They are invoked interactively by the user via the visualizer's GUI. In contrast, extractor-based analyses are batch-oriented because they are run as part of the fact extraction activity.

5.2.1 Visualizer Host Components

Table 6 provides a summary of the host components that are suitable to realize visualization functionality, along with tool-building examples. The components are classified according to their component types (cf. Section 4.1.2). Systems that are discussed in more detail in Section 5.3 are shown in **bold**. Tools that I have been involved in implementing are presented in *italics* and are discussed in Chapter 6.

OTS products of the Microsoft Office suite have been extensively used as host components for tool-building. The Visual Design Editor (VDE) is a domain-specific graph editor implemented with Visual Basic on top of PowerPoint (cf. Section 5.3.1) [Bal03] [BG00a] [Wil01] [GB99]. The experiences with VDE lead to the development of the Document Integrity Manager, a wrapper around Microsoft Word, which captures document changes and generates corresponding events (e.g., selection of text, or toggling of tool bar buttons for font styles) [TB01]. SemanticWord¹⁶¹ is an environment for authoring annotated text documents in Word [Tal03]. Word is customized with toolbars to support the insertion of semantic annotations, which are visually rendered with ActiveX controls.

Microsoft Office

There are also tools that leverage Visio with non-programmatic customizations. For example, the Restaurant Modeling Studio (RMS) is a process simulator that uses Visio for graphical modeling via customized stencils [BK02]. Similarly, the microSynergy editor customizes Visio stencils and masters to realize a graphical development environment for the construction of embedded microcontroller software [KJ03]. Huang et al. have developed a software redocumentation tool, which uses Visio to render static graph-based information such as call graphs [Hua04] [HHT01]. In this case, no customization was

¹⁶¹<http://mr.teknowledge.com/daml/SemanticWord/SemanticWord.htm>

Types	Host component	Tool-building examples
OTS products	Microsoft Office	<ul style="list-style-type: none"> • Visual Design Editor (cf. Section 5.3.1) • Document Integrity Manager [TB01] • SemanticWord [Tal03] • Restaurant Modeling Studio [BK02] • microSynergy [KJ03] • Huang et al. [Hua04] [HHT01] • Nimeta [RY02] • Galileo (cf. Section 5.3.3) • AFJCMC [Che00] [COR98] • Snap-together visualization [NS00] [Nor00b] • <i>REOffice</i> [MWW03] [Yan03] (cf. Section 6.2) • <i>REVisio</i> [ZCK⁺03] (cf. Section 6.4)
	Adobe FrameMaker	<ul style="list-style-type: none"> • SLEUTH [PFK96] • others [ATW00] [Til95] [BH96] • Desert (cf. Section 5.3.2)
	Lotus Notes	<ul style="list-style-type: none"> • IOS [LT97] • others (e.g., [LY01] [JQB⁺99] [CB99] [LIGA96]) • <i>RENotes</i> [MKK⁺03] [Ma04] (cf. Section 6.5)
	AutoCAD	<ul style="list-style-type: none"> • VisFactory [Sly98] • others [BDDD89] [Sly93] [Par99]
	Web browsers	<ul style="list-style-type: none"> • REPortal [MSC⁺01] • Software Bookshelf [FHK⁺97] and Portable Bookshelf • TypeExplorer [vDM00] [vDK99]
IDEs	Eclipse (and WSAD)	<ul style="list-style-type: none"> • SHriMP [WMSL04] [LMSW03] [RLS⁺03] [RLSB01] • many others (e.g., [BMS05] [BFBS04] [BGSS03])
	Rational Rose	<ul style="list-style-type: none"> • Rose/Architect [EK99] • UML/Analyzer [Egy00] [Egy02] • Noe and Hartrum [NH00] • ARAT [WHG⁺03] • Tichelaar and Demeyer [TD99] • others [Ber04] [Bre99] [SEU04] [CT01] [Riv04] [SPLY01]
	TogetherSoft Together	<ul style="list-style-type: none"> • JaVis [Meh02] • GoVisual [GJK⁺03] • Elucidative Programming [Ves03]
	IBM VisualAge C++	<ul style="list-style-type: none"> • Ephedra migration tool [Mar02] [MM01b]
tools	Emacs and XEmacs	<ul style="list-style-type: none"> • ASF+SDF Meta-Environment [vdBHdJ⁺01] • ISVis [JR97] • Cnest and Cscope [KC92] • Elucidator [Nør00a] • others [TAFM97] [VBM05] [Gra87] [Wii92]
	AT&T graphviz	<ul style="list-style-type: none"> • Reflexion model viewer [MNS01] [MN97] • ReWeb [RT01b] [RT01a] [RT00] • Malloy and Power [MP05] • Balmas [Bal04] [Bal01] • Reveal [MCG⁺02] • SoftVision [TMR02a] [TMR02b] • Ciao/CIA [CFKW95] [CNR90] • SPOOL [KSRP99] • others [Hua04] [GPG04] [HHT01] [vdBV05] [KCK99] [JCH01] [RC05] [BH99] [CK98] [AFL⁺97]
	SVG	<ul style="list-style-type: none"> • BOX [NEFZ00] • gViz [DS05] [BDG⁺04] • BioViz [LKSP02] • <i>SVG graph editor</i> [KWM02] (cf. Section 6.3)

Table 6: Examples of visualizer host components for tool-building

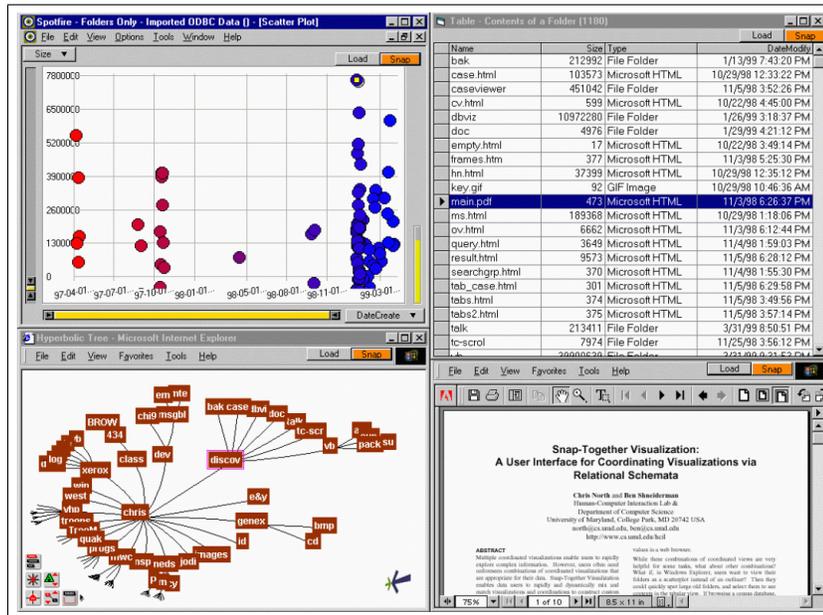


Figure 17: A snap-together visualization ([Nor00b])

necessary. The Nimeta reverse engineering tool uses Visio's functionality without customization to create architectural UML diagrams and to export the diagrams to HTML [RY02]. Nimeta produces Visio-conform XML files containing the shape, position, and size of UML elements. Visio can then directly read in and render these files.

There are also tools that combine several Office applications for visualization. The integrations use COM as a wiring standard. Galileo is a fault tree analysis tool to model the reliability of a system (cf. Section 5.3.3) [CS03] [CS00b] [CS00a] [CS99] [CS98] [SCZC97]. Engineers can manipulate graphical or textual representations of fault trees. Galileo's implementation leverages Word (for textual information), Visio (for graphical renderings of the fault trees), and Internet Explorer (for hypertext-enabled documentation). The AFJCM tool is a configuration management environment built for the Air Force [COR98] [Che00]. Information from an Access database is presented for analysis in Excel; Outlook is used for event logging; on-line information is accessed via Internet Explorer. The components are orchestrated via a stand-alone GUI application written in Visual Basic.¹⁶²

Snap-together visualization is a lightweight approach to coordinate multiple visualizations using COM-enabled components such as Spotfire, a commercial data analysis package for visualizing scatter plots, pie charts, and bar charts; and Internet Explorer to visualize images, Java applets, HTML, PDF, and Word documents [NS00] [Nor00b]. To visualize PDFs and Word documents, Adobe Acrobat and Word are embedded in Internet Explorer, respectively. Figure 17 shows a visualization with four components: A Spotfire scatter plot

¹⁶²AFJCM consists of 4418 LOC of glue code, 119 procedure, and 104 controls [Che00, sec. 7.1.2].

(top left), a custom-coded table view (top right), a hyperbolic tree view rendered in a Java applet embedded in Internet Explorer (bottom left), and a PDF file embedded in Adobe Acrobat, which in turn is embedded in Internet Explorer (bottom right). Each component has been augmented with a “Snap” button (colored in orange at the top right corner of each component), which is used by users to interactively define coordinations between components. To support coordinations, each component has to implement a simple API consisting of three calls.¹⁶³

I have participated in the implementation of two tools that customize Microsoft Office products: REOffice (cf. Section 6.2) and REVisio (cf. Section 6.4) visualize reverse engineering information in PowerPoint/Excel and Visio, respectively.

FrameMaker is a word processing and desktop publishing product from Adobe. It is used by the SLEUTH system for the authoring and viewing of system documentation [PFK96]. Among other features, SLEUTH leverages FrameMaker’s hypertext, cross-referencing, and navigation functionalities. An example of programmatic customization is SLEUTH’s Wide Area Information Server (WAIS) search engine. Other examples of tools that leverage FrameMaker’s capabilities are the Chimera hypermedia system [ATW00], PCTE [Til95, sec. 1.4.5], the WinWin requirements negotiation system [BH96], and Reiss’ Desert software development environment (cf. Section 5.3.2) [Rei99] [Rei96] [Rei95b].

Adobe FrameMaker

IBM Lotus Notes is a popular groupware and email solution employed by many diverse organizations [Kar99]. It is used extensively in research and industry projects (e.g., [Der03] [LY01] [JQB⁺99] [CB99] [LIGA96] [BL96]), and been customized to realized large-scale systems. For example, Singapore’s Housing and Development Board has realized an Integrated Office System (IOS) which is used by 3,000 staff members with Lotus Notes [LT97]. Also, Lotus Notes is used within IBM to provide various services to its staff, ranging from software development tools to registering of fitness club members [Mal01]. I have been involved in research to augment Lotus Notes with reverse engineering capabilities; the resulting tool is called RENotes (cf. Section 6.5).

Lotus Notes

AutoCAD is an example of an OTS product that is not targeted at the general end user. It can be customized with API-programming in C, or scripting in AutoLISP (a Lisp dialect with custom extensions) [GN92].¹⁶⁴ There is evidence in the literature of a number of commercial tools realized via programmatic customization of AutoLISP (e.g., GSI [BDDD89], FactoryFLOW [Sly93], and work by Park [Par99]). Since 1997 versions of AutoCAD offer, besides AutoLisp, a C++ API called ObjectARX,¹⁶⁵ which allows to access AutoCAD’s data structures and graphics system to define native commands, and to get notifications on specific events. VisFactory is an example of a tool customized with ObjectARX [Sly98].

AutoCAD

Web browsers differ from the other OTS products described so far because they represent a whole class of products: Internet Explorer, Mozilla, Opera, Konqueror, and so on [GG05]. Usually OTS products cannot be easily exchanged for one another (with the possible exception of some commercial products that have open source counterparts (e.g.,

Web browsers

¹⁶³Supporting this API for Spotfire required approximately 10 LOC in Visual Basic [Nor00b, p. 79].

¹⁶⁴Other CAD systems provide similar customization mechanism [GN92] [Bar91].

¹⁶⁵<http://usa.autodesk.com/adsk/servlet/index?id=773204&siteID=123112>

Microsoft Office and OpenOffice)). Web browsers are much more substitutable because they adhere to a number of accepted Web standards, most importantly HTML. There are myriad applications that use Web browsers to visualize information. The following examples are drawn from reverse engineering.

REportal is the attempt to make the functionality of a reverse engineering tool accessible via a Web interface [MSC⁺01]. For instance, forms allow the user to query source code for certain artifacts, which are then reported in tables. Source code can be browsed online or graphically visualized.

The Software Bookshelf is a reverse engineering and documentation environment. Once the bookshelf has been populated, reverse engineers navigate through the contents using a Web browser. The bookshelf contains hyperlinked information that is both textual (e.g., description of components and indexes) and graphical (e.g., architectural diagrams rendered with a Java applet). The original version of Software Bookshelf was used to document a 12-year-old PL/I legacy system with 300,000 LOC. More recent versions of Software Bookshelf—called Portable Bookshelf (PBS)¹⁶⁶—have been used for several reverse-engineering research projects [HH02] [HH00][BHB99] [Bre98]. There are many other tools that offer visualizations based on hypertext. For example, TypeExplorer [vDM00] [vDK99] is a tool for browsing Cobol legacy systems. It offers various (graphical) views (e.g., lists of variables, usage relationships in programs, and type graphs), which are extensively hyperlinked.

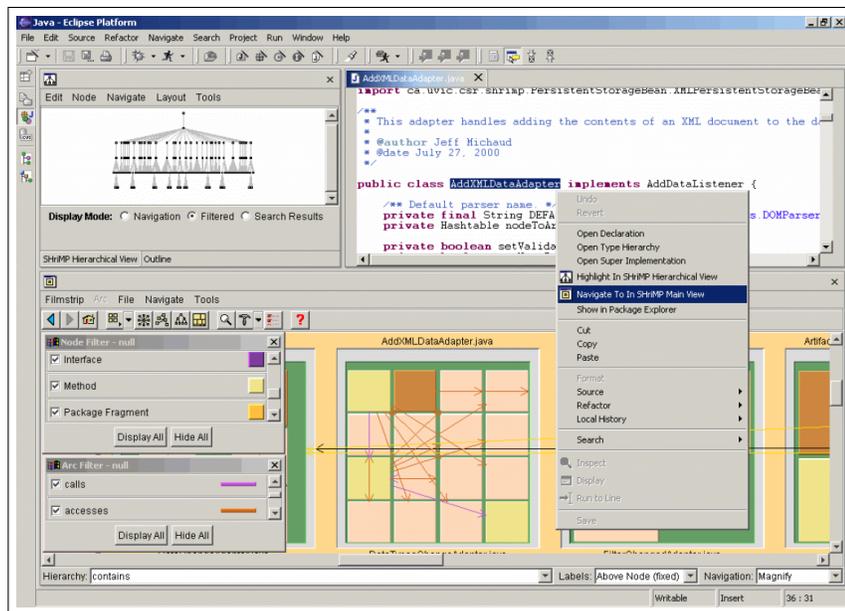


Figure 18: SHriMP Eclipse plug-in ([LMSW03])

Eclipse is an open source framework targeted at building IDEs. It is implemented in Eclipse

¹⁶⁶<http://swag.uwaterloo.ca/pbs/>

Java and can be extensively customized. The SHriMP visualization tool (implemented in Java and Swing/AWT) has been integrated into the Eclipse Java IDE [LMSW03]. The resulting tool, Creole, visualizes Java code with two new Eclipse views (cf. Figure 18): the Main View (bottom) shows a zoomable, nested view of the Java system under investigation, and the Hierarchical View (top left) shows the nesting relationships as a tree. Besides new views, Creole implements new actions; for instance, a new pop-up menu entry “Navigate to in SHriMP Main View,” which focuses the view on the selected Java element (class `AddXMLDataAdapter` in Figure 18). In previous work, SHriMP was also integrated into IBM Websphere Application Developer (WSAD), which is built on top of Eclipse [RLS⁺03] [RLSB01]. In this implementation, SHriMP was used to visualize hierarchical flow diagrams. Xia is another SHriMP-based plug-in that visualizes information of CVS repositories [WMSL04].

Many researchers have started to develop Eclipse-based tools, especially in the software engineering domain (e.g. for model checking [BHJM04], trace analysis [HLF04], assisting in code reuse [MXBK05], and Web site development [CST05]). The published papers in the *eclipse Technology Exchange* (eTX) and related workshops show the broad range of the available tools [BMS05] [BdM04b] [BFBS04] [BGSS03].

Rational Rose is a graphical software modeling tool that supports UML. It has several customization mechanisms. Non-programmatic customization is supported (e.g., customization of main and shortcut menus¹⁶⁷) as well as programmatic customization [Rat01]. The latter can be accomplished via scripting in Rose Script or a COM-enabled API. The Rose Script Editor is part of Rose and provides access to the scripting environment so that users can interactively edit and execute scripts. The syntax of Rose Script is similar to Visual Basic. Part of the internal state of the Rose application is exposed with the Rose Extensibility Interface (REI). It provides access to Rose diagrams and model elements.

Rational Rose

Egyed and Kruchten have developed an add-on to Rose called Rose/Architect, which analyses the current Rose model (i.e., UML class and object diagrams) and then proposes an abstraction of this model [EK99]. This tool was further extended into another tool called UML/Analyzer [Egy00] [Egy02]. Noe and Hartrum provide another customization example by extending Rose with formal Z specifications [NH00]. Property text fields and UML annotations are used to attach specifications to diagram elements such as classes and events. Rose Scripts then export the enhanced UML model to a textual Z- \LaTeX file. The file export can be invoked via a custom menu entry. Similarly, the Architecture-level Risk Assessment Tool (ARAT) uses Rose Script to transform information from visual UML models into a text file for subsequent analysis [WHG⁺03]. Tichelaar and Demeyer have realized an import facility with Rose Script [TD99]. Model information in a CDIF file is converted to a script with the corresponding calls to the REI. Execution of the generated script effectively imports the CDIF model into Rose. Berenbach has used Rational Rose to create a UML profile to associate use cases with requirements information.¹⁶⁸ He states, “I have found

¹⁶⁷The content of Rose menus is defined declaratively in the `Rose.mnu` file.

¹⁶⁸Private email communication with the author.

it relatively easy to extend commercial CASE tools to include symbols and relationships” [Ber04]. The APICES prototyping environment integrates Rational Rose with the SNiFF+ source code exploration tool [Bre99]. Lakshminarayana et al. use scripting to extract class information from UML diagrams in Rose [SEU04, p. 148]. This information is then used to compute (semantic) metrics to assess the software design. The UMLTest tool analyzes Rose RealTime state diagrams and automatically derives test cases from them [CT01]. Riva’s Nimeta architecture reconstruction tool uses a number of visualizer components, including Visio and Rational Rose [Riv04, sec. 8.5.3]. Nimeta uses a Python script to export its architectural models for visualization in Rational Rose, which is used because “it is familiar to the architects that are the end-users of the reconstructed views.” Lastly, the Holmes domain analysis tool uses Rational Rose for use-case and class diagram editing [SPLY01].

TogetherSoft Together is a UML CASE tool, and an IDE for Java and C++.¹⁶⁹ It has modeling capabilities comparable to Rational Rose. The Together Open API can be programmed in Java and consists of three different kinds of interfaces.¹⁷⁰ A low-level interface allows to manipulate Java sources at the byte code level. A high-level interface allows read-only access of Together’s state (e.g., model information). The mid-level interfaces allows modification of the model and customization of Together. The JaVis tools uses Together to visualize trace data for concurrent Java programs as UML sequence diagrams [Meh02]. The trace data is read from a file and the corresponding sequence diagram is constructed via calls to Together’s API. There is also a step by step display of the trace, which can be navigated forwards and backwards.

TogetherSoft
Together

The GoVisual plug-in enhances Together with sophisticated layout algorithms for UML class diagrams [GJK⁺03]. GoVisual adds a new top-level menu, toolbar menu, and dialogs to control the layout of the diagrams. (There is also a GoVisual plug-in for Microsoft Visio.) Vesterdam has implemented an elucidative programming environment¹⁷¹ on top of Together [Ves03].¹⁷² Elucidative programming separates source code and documentation into separate entities, but ties them together with bidirectional hyperlinks. New views are provided to support this kind of programming; for instance, the editor tab allows to create links via marking, the message tab reports problems such as invalid links, and the catalog tab shows a file explorer providing an overview of the physical catalog organization of the documentation. Navigation is provided from links in the editor tab to the corresponding source code. Navigation from the source code to the documentation is provided by a separate view that shows all relevant links depending on the current cursor position.

IBM VisualAge C++ (which is also know by its code name Montana) is an extensible C++ IDE with an incremental C++ compiler [Kar98] [Nac97].¹⁷³ It is a black-box compo-

IBM VisualAge C++

¹⁶⁹TogetherSoft has been acquired by Borland and the tool is now called the Borland Together ControlCenter.

¹⁷⁰<http://bdn.borland.com/article/0,1410,29957,0.html>

¹⁷¹<http://dopu.cs.auc.dk/>

¹⁷²Elucidate programming is inspired by Knuth’s literate programming [Knu84], but does not require a reorganization of the source code [VN02].

¹⁷³IBM has decided to cease development of VisualAge; it has been superseded by Eclipse [BdM04a].

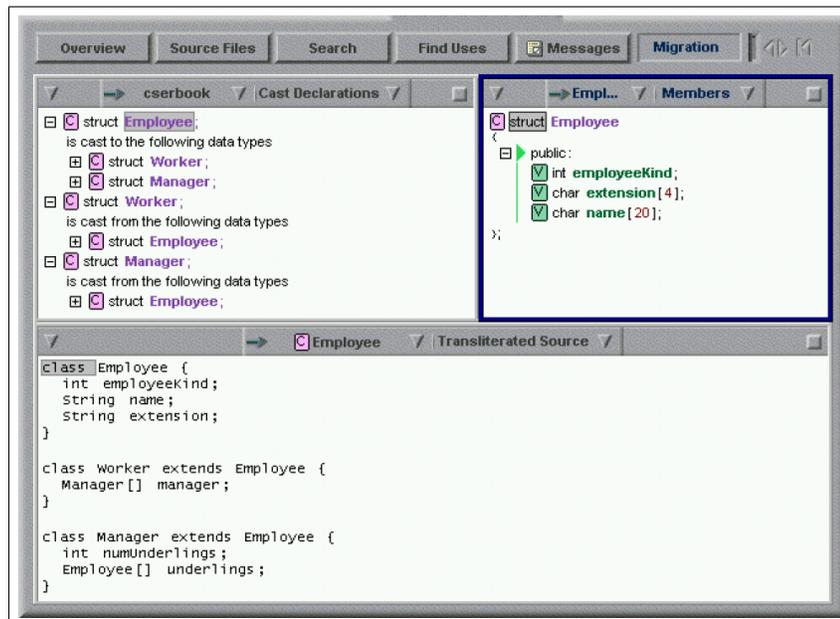


Figure 19: Ephedra migration tool ([Mar02])

ment with a sophisticated C++ API, which allows compiler-related extensions such as style checkers and stub generators. Extensions are described declaratively in special text files and are loaded dynamically [SKBS97]. VisualAge is a closed environment in the sense that its API is not officially published. However, the API has been used by different groups within IBM and affiliated researchers to add functionality to VisualAge (e.g., [Ham97]). Martin has built a Visual Age C++ extension that aids reverse engineers to migrate C code to Java (cf. Figure 19). This extension, part of the Ephedra tool,¹⁷⁴ identifies certain code patterns and proposes suitable code migrations to Java. Specifically, data structures and type casts involving these data structures are analyzed for a possible (conceptual) inheritance relationship of the data structures. Ephedra adds a page (“Migration”) with three new views (cf. Figure 19). One view (top left) shows the program’s data structures (e.g., structs) and casts to and from the data structures. Upon selection of a data structure, the top right view shows the data structure’s corresponding C language implementation. The bottom view shows the proposed Java class hierarchy based on the analysis of the program’s type casts.

The GNU Emacs and XEmacs text editors¹⁷⁵ are popular customization targets for text-based software engineering tools. Emacs has a small kernel written in C; most of the editor’s functionality is implemented in Emacs Lisp. Consequently, programmatic customiza-

Emacs and XEmacs

¹⁷⁴<http://ovid.tigris.org/Ephedra/>

¹⁷⁵GNU Emacs and XEmacs are the two most popular versions of the Emacs text editor. XEmacs is the result of a code fork in 1991 from GNU Emacs. XEmacs’ strength is more extensive GUI support compared to GNU Emacs (<http://en.wikipedia.org/wiki/XEmacs>).

tions can be realized seamlessly with Emacs Lisp programming. Emacs has a number of different customization mechanisms. For example, some functionalities offer callbacks (or *hooks*) that an extension can implement. Since Version 20, GNU Emacs has included a customization facility to set options with an interactive interface. Changing an option triggers Emacs Lisp code on behalf of the end user. Specific customizations (e.g., major modes for editing) can be mostly realized with a declarative specification in Emacs Lisp. Many IDEs have been developed on top of Emacs; in fact, XEmacs (then called Lucid Emacs) was created to realize Lucid's Energize C/C++ development environment.

The ASF+SDF Meta-Environment integrates XEmacs via its tool bus [vdBHdJ⁺01]. The ISVis software visualization tool uses Emacs to show source code and other views to the programmer [JR97]. Tonella et al. leverage Emacs to visualize slices of C programs [TAFM97]. Xrefactory is a program understanding and refactoring tool that has been integrated into Emacs [VBM05]. Pressing a hot-key display a context-dependent menu of refactoring options. Not all customizations are targeted at programming languages; there is also a microprocessor development environment based on Emacs [Gra87]. The Emacs HyperText System (EHTS) is a multiuser system for collaboration [Wii92]. It extends Emacs with a new minor mode and commands for hypertext navigation.

Emacs has been used as a vehicle to explore different programming paradigms. Kortright and Cordes have developed the Cnest and Cscope tools integrated in Emacs to support literate programming [KC92]. The editor part of an environment to explore elucidative programming, called Elucidator, uses Emacs to show a split screen: one view presents the source code, the other view shows the code's documentation [Nør00a].¹⁷⁶ It seems that Emacs' popularity for customization has waned among researchers in recent years, mainly in favor of GUI-based IDEs such as Eclipse.

AT&T graphviz is a package for graph drawing and editing.¹⁷⁷ Its most popular tools are `dot` and `dotty`. A related tool is Grappa,¹⁷⁸ which has ported a subset of graphviz's functionality to Java [BML97].¹⁷⁹ Many reverse engineering tools use `dot` to render static graphs. Telea et al. use several graph layouts of the graphviz package to lay out graphs for their SoftVision software visualization tool [TMR02a] [TMR02b]. The SPOOL reverse engineering environment employs the same approach to visualize properties of C++ code (e.g., class hierarchies) [KSRP99]. The Reveal tool uses `dot` to create class diagrams from C++ code [MCG⁺02]. The class diagrams are exported to Postscript with `dot` and rendered with the `gv` Postscript viewer. There are many more visualization examples (e.g., Huang et al. [Hua04] [HHT01], ASF+SDF Meta-Environment [vdBV05], Chava [KCK99], clustering for data mining [JCH01], FPGA testing [RC05], and Verilog designs [BH99]).

AT&T graphviz

Part of the tool support for Murphy's reflection models are viewers implemented with `dotty`¹⁸⁰ [MNS01] [MN97]. For example, one viewer visualizes the high-level archi-

¹⁷⁶Vesterdam has realized a more recent tool for elucidative programming with Together; see above.

¹⁷⁷The graphviz package consists of more than 300,000 LOC [dj04].

¹⁷⁸<http://www.research.att.com/~john/Grappa/>

¹⁷⁹Grappa is implemented with 10,000 LOC of Java.

¹⁸⁰The `dotty` graph editor can be customized with a dedicated scripting language, called `lefty`, which is

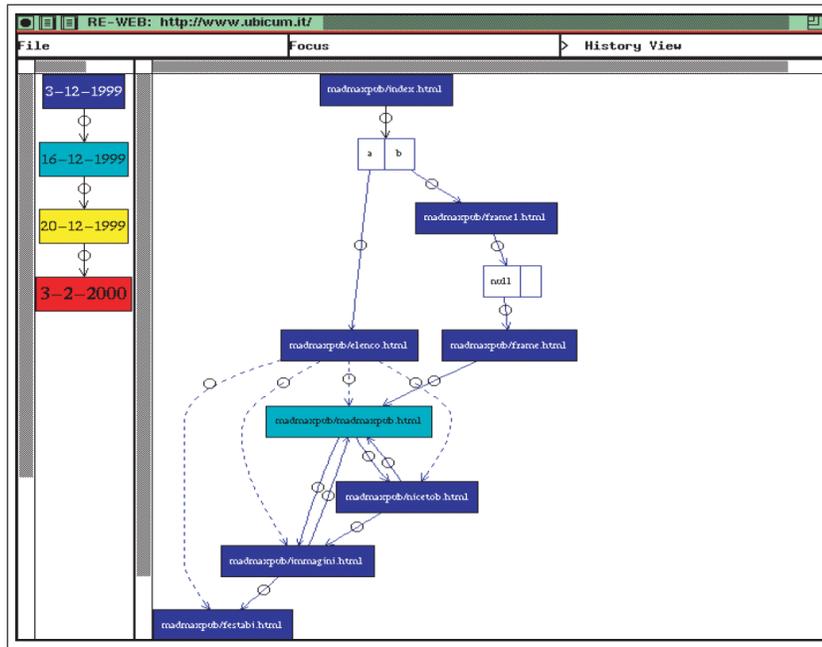


Figure 20: ReWeb tool ([RT01b])

texture of the software system. The viewers are interactive; for instance, clicking on an arc opens a window that provides more detailed information about the arc and its nodes. The ReWeb Web site understanding tool visualizes the architecture and structure of a Web site [RT01b] [RT01a] [RT00]. Among the available views is a history view that shows the changes made to a site over time (cf. Figure 20), and a system view that shows the organization of pages into directories. The viewers support navigation and querying, including zooming, searching, and displaying of HTML code. Similarly, the WARE tool, which reverse engineers Web applications, uses `dotty` (besides `Rigi` and `VCG`) to visualize architectural diagrams as well as clusterings [DLFP⁺02b] [DLFP⁺02a]. Malloy and Power create class diagrams and call graphs from C++ programs [MP05]. These graphs are rendered with `pyDot`, which allows to visualize `dot` graphs in a Python canvas. The graph is interactive because it is rendered as Python canvas objects. For instance, selecting a node in the call graph launches a pop-up window that displays the node's immediate predecessors and successors. Similarly, `PROVIS` (part of `CANTO`) provides various interactive representations of C++ code, including system level, module, and task views, as well as call graphs, post-dominator trees, ASTs, control dependency graphs, and storage shape graphs [AFL⁺97]. Balmas uses `Tcldot` (part of the `graphviz` distribution) to load and render C dependency graphs represented as `dot` files in a `Tcl/Tk` canvas [Bal04] [Bal01]. The graph editor, implemented in `Tcl/Tk`, allows interactive manipulation such as grouping, selecting, and annotating of nodes. The `Ciao/CIA` tool analyzes C/C++ code [CFKW95]

similar to `awk` and `C`.

[CNR90].¹⁸¹ An extractor populates a relational repository, which can be queried and visualized with textual and graphical viewers. For graphical views (e.g., call graphs and include dependencies), both *dot* and *dotty* are used. From the *dotty* viewer “users can invoke queries and operators directly from each node in the graph. A pop-up menu is attached to each node listing only the legal operations that can be applied to the corresponding entity” [CFKW95].

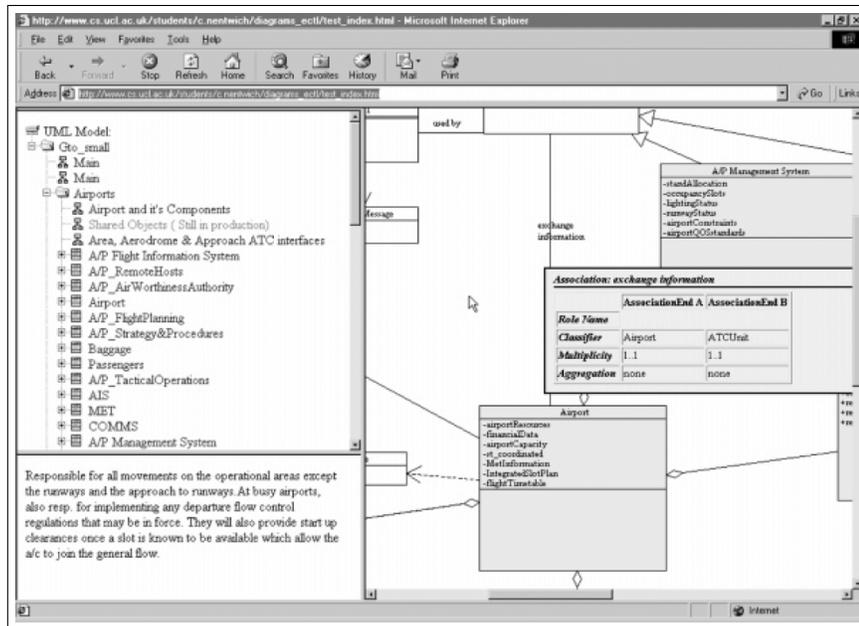


Figure 21: BOX tool ([NEFZ00])

Web-based reverse engineering tools have used mostly static images to provide graphical information. However, reverse engineers often desire interactive manipulation of information. This can be achieved with applets. For instance, the LSEdit graph editor, written in Java, can be run as an applet [SHD05]. Similarly, AT&T’s Enterprise Navigator uses Grappa to visualize graphs interactively in an applet [BCH⁺01]. Another approach is the W3C’s Scalable Vector Graphics¹⁸² (SVG) standard. SVG is a XML-based graphics format that can be made interactive via JavaScript programming (cf. Section 6.3). SVG is becoming more and more popular as a graphics format. The *dot* tool now also generates (static) SVG as an output format, and the ZGRViewer¹⁸³ renders *dot* graphs in SVG. Meng and Wong use Singleton Lab’s MonarchGraph library¹⁸⁴ to generate story diagrams (a combination of UML activity and object collaboration diagrams) in SVG [MW04a]. The BOX

¹⁸¹There is also an instantiation of the tool for Web sites, called Web Ciao (<http://www.research.att.com/~chen/webciao/>) [CK97].

¹⁸²<http://www.w3.org/Graphics/SVG/>

¹⁸³<http://zvtm.sourceforge.net/zgrviewer.html>

¹⁸⁴<http://www.singleton-labs.com/mgraph.html>

tool exports UML diagrams from Rational Rose as Vector Markup Language (VML)¹⁸⁵ and renders them in Internet Explorer [NEFZ00]. Figure 21 shows a screenshot of BOX's Web interface, which is implemented with Dynamic HTML and JavaScript to provide interactive visualization. For instance, detailed information about a UML modeling element can be obtained by moving the mouse over it. There are a number of interactive viewers and editors that have been developed with SVG (e.g., the BioViz genome viewer [LKSP02] and the gViz dataflow networks editor [DS05] [BDG⁺04]). To visualize reverse engineering graphs, I have developed a generic graph editor in SVG [KWM02] (cf. Section 6.3).

Besides the components discussed in some detail above, there are many others. For instance, researchers have written Visual Studio .NET plug-ins to augment the IDE with program comprehension functionality for searching [PMDS05], and to make the IDE interoperable with reverse engineering tools such as Rigi [MWS04]. Egyed and Wile have customized Matlab/Stateflow to realize a statechart simulator [EW01]. If reverse engineering data is provided as XML, it can be visualized with generic XML tools such as Corel's XMetal and Altova's XML Spy. Huang et al. encode program documentation in XML, visualizing it with XMetal and Internet Explorer [Hua04] [HHT01]. Sperberg describes the benefits of generic XML tools as follows [Spe05]:

Others

“XML is interesting to people who wish to exploit their data, because it provides enough commonality among documents to make possible the construction of generic tools instead of tools specialized for single notations. For those who want to spend their time working with their data, rather than building new tools, such generic tools are a great step forward.”

Discussion The collected catalog shows the diversity of host components, but also exposes that many tool builders have chosen to customize Microsoft Office in recent years. Recognizing the popularity of Microsoft Office among tool users and builders, I have also implemented a number of tools with Microsoft Office. The discussed tool-building examples vary widely in the effort that has been expended when leveraging a host component. Some component-based tools represent a significant programming effort by programmatically customizing a component. Others tools do not customize a component at all. In such cases the component is often used to render a document generated by the tool such as an HTML page rendered by a Web browser. AT&T graphviz is an example of a host component that has been programmatically customized via `dotty`, as well as used as-is as a graph viewer via `dot`.

Catalog

While my focus is on leveraging components for reverse engineering tools, it should be pointed out that there is a close relationship to the building of domain-specific visual languages (DSVLs) [KWJM02]. DSVLs describe a problem domain with a graphical paradigm. As a result, a DSVL has to provide an environment to render and manipulate

DSVLs

¹⁸⁵VML, introduced by Microsoft, was one of the proposals considered by the W3C for the SVG standard. As a result, SVG is similar to VML. Microsoft has implemented a native VML renderer in Internet Explorer 5.

graphical elements. GUI-based components with appropriate base functionalities such as Visio and PowerPoint can be a suitable host component for a DSL [KWJM02] [Wil01].

Tool builders seem to neglect that reverse engineering and maintenance is a document-centric process. In order to understand a software system, diverse sets of documents have to be consulted and manipulated—for instance, UML diagram, natural language user documentation, configuration files, and source code; possibly audio and video sequences. For effective management of these documents, the reverse engineer requires some kind of version control and configuration management functionality. Furthermore, larger-scale maintenance activities are typically performed by teams (possibly geographically distributed), which makes tool support for collaborative work a desirable feature.

Document-centricity

Few researchers have started to create reverse engineering tools that support this crucial combination of features. Instead, tools are mostly stand-alone solutions, focusing on the manipulation of a single document type in isolation. As a result, to accomplish a single reverse engineering activity, a complex tool chain is necessary. For example, to update a UML diagram might involve separate tools (1) to edit the UML file, (2) to evaluate it, (3) to (re-)generate code from it, (4) to archive it, and (5) to communicate the changes to team members. The above scenario is still simplified because it does not take into account that changing a document such as a UML diagram might require interactions with various stakeholders; and these interactions trigger in turn the manipulation of supporting documents.

The building of more document-centric tools is desirable to increase the attractiveness—and ultimately adoption—of reverse engineering tools. This poses new challenges to researchers because such tools need to be highly configurable, and provide a diverse set of functionalities—many in areas that fall outside of the researcher’s expertise and core research agenda.

One approach to ease the construction of document-centric tools is the use of OTS products. Office tools such as Microsoft Office and Lotus Notes already have support for document management and collaboration. In fact, reverse engineers already use these products in their tools chain to accomplish reverse engineering activities. Researchers can leverage the existing functionality of suitable OTS products to create tools that are more document centric, and that better reflect the complex activities of users. I believe that my proposed tool-building approach can accommodate the creation of document-centric reverse engineering tools.

5.2.2 Extractor Host Components

This section provides examples of fact extractor components that have been employed in the building of reverse engineering tools. Table 7 provides a summary of the host components that are suitable to build fact extractors for reverse engineering. Note that there are components (e.g., Eclipse and IBM VisualAge) that have been already discussed in the previous section (cf. Table 6) because they can be utilized for both visualization and extraction; this is typically the case for IDEs. Researchers have a keen interest in reusing

Types	Host component	Tool-building examples
IDEs	Eclipse (and WSAD)	<ul style="list-style-type: none"> • SHriMP [LMSW03] • JavaDB (http://www.cs.mcgill.ca/~martin/javadb/) • Xia [WMSL04] • WSAD Web site extractor [KM06] (cf. Section 6.7)
	IBM VisualAge C++	<ul style="list-style-type: none"> • Hind et al. [HBCC99] • Rigi C++ extractor [Mar99] • ISME [MK00]
	Adobe GoLive	<ul style="list-style-type: none"> • REGoLive [GKM05b] [GKM05a] [GKM05c] [Gui05] (cf. Section 6.6)
tools	GNU GCC	<ul style="list-style-type: none"> • CPPX [DMH01] • Myer [Yav04] • Yu et al. [YDMA05] • Power and Malloy [HMP03] [PM02] • TUNalyzer [GPG04] • XOgastan [APMV03] • g⁴re [KMP05a] [KMP05b] • GCC.XML (http://www.gccxml.org)
	SNiFF+	<ul style="list-style-type: none"> • sniff2cdif [TD99] • Moose [NDG05] • CrocoCosmos [LS02] • Dali [KC99] [CK98] [KC98] • Woods et al. [WCK99] • Huang [Hua04, p. 83]
	Source Navigator	<ul style="list-style-type: none"> • TKSee/SN (C++) • Moise and Wong (C/C++) [MW03] • Pinzger et al. (Visual Basic) [POG03]
	Reasoning Refine	<ul style="list-style-type: none"> • RevEngE (PL/AS) [MSW⁺94] [Won99] • Buss and Henshaw (PL/AS) [BH91] [BH92] • Newcomb et al. (Cobol) [NM93] [MNB⁺94] • ART (C) [FATM99] [AFL⁺97] [TAFM97] • Woods et al. [WCK99] • others [MGK⁺93] [RPR93] [WBM95] [YHC97] [NH00]
	EDG	<ul style="list-style-type: none"> • FORTRESS [GPB04] • Xrefactory [VBM05] • GENOA [Dev99a, p. 198]

Table 7: Examples of extractor host components for tool-building

existing extractors because many languages are difficult to parse. Also, parsing of source is for most researchers a tedious prerequisite that needs to be done before the actual research can take place.

Part of the Eclipse Java IDE is an incremental compiler that can be used for fact extraction. The compiler provides a Java API that allows access to its AST; there is also the Java Development Tools (JDT) model, which provides a simplified, high-level view of the AST. The Creole tool, which integrates SHriMP with Eclipse, makes use of the Java API [LMSW03]:

Eclipse (and WSAD)

“The JDT provides a well-documented API which gives us access to information about the software artifacts in a project. Software elements are organized into an easily understood hierarchy. . . . The API also provides a search engine that allowed us to find the relationships between software artifacts. For example, the engine finds all methods that reference a specific field.”

The JavaDB¹⁸⁶ fact extractor uses mostly the JDT model to obtain facts from Java code. It is used by other Eclipse plug-in such as FEAT,¹⁸⁷ SHriMP,¹⁸⁸ and Hannemann et al.’s refactoring tool [HK05]. Eclipse has extensions in the form of plug-ins; some of these can be used as a source for fact extraction. The Xia tool uses Eclipse’s CVS plug-in to extract and visualize version information of a software system [WMSL04]. IBM Websphere Application Developer (WSAD) provides plug-ins that make it possible to extract information from Web sites. I have built a Web site extractor for J2EE projects using WSAD (cf. Section 6.7).

IBM VisualAge C++ offers a C++ API to interface with it; however since the API is not publicly available, there are few documented uses. The API provides access to the compiler’s internal data structures, referred to as CodeStore.¹⁸⁹ Hind et al., from the IBM Thomas J. Watson Research Center, leverage VisualAge to obtain an AST that they use as a basis for for alias analysis [HBCC99]. The Rigi C++ extractor is based on VisualAge [Mar99]. It uses the API to generate facts about the compiled C++ source in RSF as a side-effect of the compilation process. Similarly, the C++ parser of the Integrated Software Maintenance Environment (ISME) uses VisualAge to generate information in the C++ Markup Language (CppML) [MK00].

IBM VisualAge C++

Adobe GoLive is an integrated environment to produce Web sites. Since GoLive provides programmatic access to assets such as HTML and XML files via a JavaScript API, information for Web site reverse engineering can be extracted. I have used GoLive to realize such a tool, called REGoLive (cf. Section 6.6).

Adobe GoLive

The GNU Compiler Collection (GCC) project of the Free Software Foundation is one of the most widespread set of compilers in use today [EGH⁺05]. GCC can parse a va-

GNU GCC

¹⁸⁶<http://www.cs.mcgill.ca/~martin/javadb/>

¹⁸⁷<http://www.cs.ubc.ca/labs/spl/projects/feat/>

¹⁸⁸<http://www.thechiselgroup.org/creole>

¹⁸⁹Since extensions are written in C++ and have write access to the internals, they can crash the compiler. VisualAge assumes that writers of extensions “know what they are doing” [SKBS97].

riety of languages, among them C, C++, Java, Fortran, and Ada. The CPPX extractor¹⁹⁰ modifies GCC's C++ front end to generate output in TA, VCG, and GXL [DMH01]. To obtain information about the processed source, CPPX accesses GCC's internal representation (which is encoded as C structures). The Myer project¹⁹¹ patches GCC to provide precise location information for C program units [Yav04]. The author of Myer states that "little patch code is needed: The gcc patch accounts for only 15% of Myer's source code, and almost half of that is 'original lines' to be replaced with 'patch lines.'"

To reduce the build time of large software systems, Yu et al. have developed algorithms to detect redundant declarations and false dependencies (caused by superfluous inclusion of header files) in C/C++ code [YDMA05]. GCC has been modified to track these dependencies.¹⁹² Instead of directly modifying gcc, Power and Malloy have extended the bison parser generator to emit a parse tree of the processed source in XML format [PM02].¹⁹³ There is also a graphical tool, gccXfront, that visualizes a simplified version of the parse tree [HMP03]. The simplified version is obtained by transforming the XML output with XSLT. Gschwind et al. use GCC to analyze templates in C++ code with the TUAnalyzer [GPG04]. They simply apply compiler switches that produce textual tu files of GCC's internal representation.¹⁹⁴ These text files are then translated to RSF format. XOgastan uses the same approach to obtain a parse tree for C++ source code [APMV03]. A Perl script translates the tu files into GXL. The GXL output is then analyzed to produce information (e.g., call graphs and uses of variables) in HTML, XML, dot format, and GIF images. The GCC XML Node Introspector¹⁹⁵ and g⁴re [KMP05a] [KMP05b] are also based on GCC's tu files. Since tu files can get large, g⁴re provides tools to prune them.

SNiFF+ is a source code browsing and comprehension tool [Bis92]. It consists of various browsers that show symbol, class, inheritance, cross reference, and include information. To obtain this information, SNiFF+ uses a fuzzy parsing technology, supporting various languages, among them C, C++, Java, Ada, and Tcl. SNiFF+'s Symbol Table API provides programmatic access to the extracted information. Tichelaar and Demeyer's sniff2cdif tool uses SNiFF+'s C API to produce UML diagrams of SNiFF+ projects [TD99]. The authors note, "especially convenient is the fact that SNiFF+ provides queries to extract information about which method calls which method and which attribute is accessed by which method." An early incarnation of the Moose reverse engineering tool has also used SNiFF+ to parse C/C++ code [NDG05]. CrocoCosmos [LS02] uses the SNiFF+ API to extract information from C++ and Java, which is then stored in a relational database. The Dali architecture reconstruction tool uses SNiFF+ to extract facts from C and C++

¹⁹⁰<http://swag.uwaterloo.ca/~cppx>

¹⁹¹<http://savannah.nongnu.org/projects/myer>

¹⁹²Yu et al. have modified the GCC type-checker that calls the `build_external_ref` function. They also added two new options, `-fdump-program-unit` and `-fdump-headers`.

¹⁹³The C, C++ and Java front ends of GCC all used to be based on bison. However, GCC is now using a different parser based on recursive decent, which makes this approach no longer applicable [KMP05a].

¹⁹⁴For instance, `-fdump-translation-unit` outputs a textual tree representation for each translation unit. GCC's `-fdump-*` options are typically used for debugging of the compiler and are available since Version 2.9.

¹⁹⁵<http://www.gccxml.org>

[KC99] [CK98] [KC98]. Woods et al. use SNIFF+ to obtain information for call graphs and include-file dependencies of C code [WCK99]. Huang uses SNIFF+'s Java API (part of Version 4) to extract facts for program redocumentation [Hua04, p. 83].

Source Navigator¹⁹⁶ is a source code analysis tool similar to SNIFF+. First a commercial product from Red Hat, it is now open source. Source Navigator supports C, C++, Cobol, Fortran, and Tcl. The parser populates a database that can be queried with a procedural API. TkSee/SN¹⁹⁷ is a C++ fact extractor based on Source Navigator, developed by Tim Lethbridge's group at the University of Ottawa. It generates output in GXL format. Moise and Wong have written a fact extractor for C/C++ based on Source Navigator [MW03] [MW05]. They retrieve information from Source Navigator's repository, taking advantage of its fuzzy parsing, but also use "local scans into the source files, especially in parts where Source Navigator is weak, such as local variables or templates" [MW03]. They generate RSF (adhering to their own C/C++ domain model, CPPDM) that is subsequently visualized by Rigi. Pinzger et al. use Source Navigator to extract information from Visual Basic for program comprehension of COM+ components [POG03].

Source Navigator

Software Refinery is a commercial reengineering tool environment [ABC⁺94]. It enables source-to-source transformations of languages such as COBOL and C with dedicated instantiations of the tool environment with Refine/COBOL and Refine/C, respectively. Refine provides customizable LALR(1) parsers¹⁹⁸ and pretty-printers, an object-oriented database that stores ASTs, and a domain-specific language for analyzing and transforming the database contents. There is also a GUI toolkit that provides support for automatic tree layouts, hyperlinking among multiple views, and customizable diagrams, table, menus, and so on. The RevEngE tool integrates Rigi and Refine to realize an integrated toolset for program understanding [MSW⁺94] [Won99]. RevEngE uses Refine to parse PL/AS code from the SQL/DS system. The parsed code is represented as an AST and analyzed to produce clusterings. The clusterings are visualized as table views with Refine's GUI toolkit. In another project, Buss and Henshaw also use Refine to analyze SQL/DS [BH91] [BH92]. Since PL/AS is not supported by Refine, they had to write their own grammar and domain model.

Reasoning Refine

The Newcomb tool uses Refine/COBOL to automatically restructure COBOL programs. To parse the code, Refine's COBOL parser had to be modified [NM93] [MNB⁺94]. ART is an architecture reverse engineering environment that "is built on top of the Refine/C and exploits its capabilities to build and analyze ASTs" [FATM99]. ART augments Refine's AST with control and data flow information (using pattern matching over Refine's AST) to recover the architecture of C code [AFL⁺97] [TAFM97]. Woods et al. use Refine/C to obtain fine-grained information of C code, which they use to augment the coarse-grained information obtained from SNIFF+ [WCK99]. There are many more examples of reverse engineering tools that are based on Refine [MGK⁺93] [RPR93] [WBM95] [YHC97] [NH00].

¹⁹⁶<http://sources.redhat.com/sourcenav>

¹⁹⁷<http://www.site.uottawa.ca:4333/dmm/>

¹⁹⁸However, the parsers now probably use a GLR-based parsing technique [vdBSV98].

The C++ front end of the Edison Design Group¹⁹⁹ is capable of a full syntax and semantic analysis of the source code. It is a very mature product²⁰⁰ used by various compiler vendors; a free license can be obtained for non-commercial use. EDG can handle various dialects (e.g., GNU C/C++, Microsoft Visual C++, and cfront), controlled by command-line options. EDG is written in C and has an API to hook up back ends, but can also generate cross-reference information. EDG

The GENOA source code analysis tool defines its own specification language to interface with compiler front ends. There is an implementation of GENOA that attaches to EDG [Dev99a, p. 198]. Xrefactory is a source code understanding and refactoring plug-in that supports C, C++, and Java. For C++, Xrefactory relies on the EDG front end [VBM05]. The FORTRESS reverse engineering tool uses EDG parsers to obtain ASTs for C++ and Java [GPB04]. Given the features of EDG, it is surprising that relatively few academic tools take advantage of it.

There is a broad spectrum of other components to extract facts from diverse sources. Others Imagix 4D²⁰¹ is a commercial program comprehension tool. Its C/C++ extractors has been used (e.g., Dali [KC99] [O'B01] [CK98], and Pinzger et al. [POG03]) and evaluated (e.g., [BG98] [MNGL98]) by various researchers. An interesting example of an OTS component that can be utilized for fact extraction is Microsoft Visio. It offers functionality to crawl Web sites (to visualize its link structure) and to parse Visual C++ code (to generate UML diagrams). However, these extractors have a limited customizability.

Many extractors are based on parser generators such as `yacc` (e.g., the Bauhaus and Rigi C parsers as well as PAn [CC00]), `BtYacc`²⁰² (e.g., CobolTransformer [vdBSV98] and the Keystone C++ parser²⁰³), and `JavaCC` (e.g., SPOOL's UML/CDIF parser [KSRP99] and ISME's Java parser [MK00]). Devanbu's GENOA source code analysis framework [Dev99a] is used by researchers to parse C++ (e.g., SPOOL [KSRP99] [SSK00]) and Java (e.g., [KP03]). Bauhaus' Ada extractor uses GNAT's ASIS implementation for fact extraction [Nei04]. The MultiLex system cascades five `flex` analyzers to implement a hierarchical lexical analyzer for C [CC00].

Scripting languages with powerful string processing and matching capabilities are convenient for simple fact extraction tasks. Riva uses both Perl and Python's regular expression facilities to extract specific architectural concepts [Riv00a] [Riv04]. Scripting languages such as Perl are also used to transform or merge fact bases (e.g., [KP03] [O'B01]). Finnigan et al. have developed a simple extractor with Emacs macros, which can handle "procedure definitions and calls, and variable declarations and references" [FHK⁺97, p. 577]. The CLICS clone detection tool uses the `ctags` utility "for extracting indices of language objects found in the source code" [KG05]. To analyze the architecture of COM+ components,

¹⁹⁹<http://www.edg.com/cpp.html>

²⁰⁰The front end "consists of about 426,000 lines of source code, of which about 30% are comments," and has extensive documentation of about 560 pages.

²⁰¹<http://www.imagix.com>

²⁰²<http://www.siber.org/btyacc/>

²⁰³<http://www.cs.clemson.edu/~malloy/projects/keystone/keystone.html>

Pinzger et al. rely on Imagix 4D and Source Navigator, but also use `grep` to match certain COM+ idioms [POG03].

Discussion For a visualizer host component, the capabilities of its user interface plays an important role. In contrast, this is only a minor concern for extractor host components, which are non-interactive. Instead, the extractor’s parsing strategy is important, because it determines properties such as the extractor’s accuracy, speed, and robustness. An extractor that provides accurate facts is a necessary prerequisite for certain analyses (e.g., slicing and alias analysis), but it tends to fail during parsing for various reasons. On the other hand, a robust extractor can skip over problems encountered during the extraction, but the resulting facts often contain false positives and false negatives (that are also difficult to track down). Thus, selecting of an extractor host component requires an understanding of its employed extraction technique with its associated merits and drawbacks. Compilers and compiler front-ends are a popular foundation for accurate fact extractors; however, they typically use parsing techniques that are not necessarily the best choice for a reverse engineering tool because their grammar lacks modularity and compositionality [vdBSV98]. Also, using a compiler can introduce unnecessary overhead. For instance, to obtain facts from the Rigi C++ extractor, the VisualAge compiler has to perform a full compile, including optimizations and code generation.²⁰⁴ Certain language features can pose additional requirements for fact extractors. The C and C++ preprocessor makes it necessary to decide whether facts should be extracted before or after the preprocessing stage. This needs to be kept in mind when selecting an extractor host component. For instance, Martin made the following experience when implementing the Rigi C++ fact extractor: “The information available in the [VisualAge] CodeStore refers to the program after macro expansion. It is very hard to find out where macros are defined or used” [Mar99]. Since different extractor host components have different properties, researchers have combined the facts produced by multiple extractors to obtain complementary information (e.g., [POG03] [WCK99]). This approach results in more complete fact bases, but can complicate maintenance.

5.3 Sample Tool-Building Experiences

The previous section provided a catalog of host components and briefly identified tools that have been built leveraging these components. The catalog does not go into detail on how these tools have been built—indeed, most research does not provide sufficient detail on this.

In contrast, this section describes three tool-building experiences in detail:

- Visual Design Editor (cf. Section 5.3.1)
- Desert (cf. Section 5.3.2)

²⁰⁴This can be avoided if the compiler offers suitable command line options to omit compilation phases. For example, GCC has an option `-fsyntax-only` [YDMA05].

Taxonomy criteria	VDE	Desert	Galileo
origin	external	external	external
distribution form	black-box	black-box	black-box
customization mechanism	scripting	API	scripting
interoperability mechanism	data	control	presentation
packaging	interactive	interactive	interactive
number of components	single	single	homogeneous

Table 8: Characteristics of the discussed tool-building experiences

- Galileo (cf. Section 5.3.3)

For each of these tools, I describe its features and how it was implemented. Thus, these descriptions serve as sample case studies of component-based tool construction. Table 8 provides a brief summary of the characteristics of the tools and their leveraged components using my component taxonomy (cf. Section 4.2). The tools cover both graph-based and textual editing in the domains of semantic markup of documents (Visual Design Editor), software construction (Desert), and fault-tree analysis (Galileo).

5.3.1 Visual Design Editor

The Visual Design Editor²⁰⁵ (VDE) is a domain-specific graph editor implemented on top of PowerPoint [GB99] [BG00a]. VDE is a fully interactive graph editor. Users can create new nodes, connect nodes with arcs, and assign properties to nodes and arcs. Node types are rendered with different shapes and colors to distinguish them from each other. Similarly, arc types use different line styles and colors. Arcs can be directed (rendered with one arrow) or undirected (rendered with two arrows).

Features

The Briefing Associate [TBG02] [TBG01] is a descendent of VDE. It targets authoring of information for the Semantic Web, which enhances documents with semantic markup. The Briefing Associate uses DARPA Agent Markup Language²⁰⁶ (DAML) to encode the semantics of a document. In this case, the nodes and arc types that are used to construct a semantic document with PowerPoint represent corresponding DAML entities.

VDE originally was designed for a specific domain (i.e., to provide notations and analyses for architectural styles in Acme) [Wil01]. It now is customizable and can support different domains. Figure 22 show a screenshot of VDE that allows users to design graphs to represent satellite communications systems. There are node types to represent satellites (Sat, S1, and S2 in Figure 22), terminals (T1 and T2), switches (SW1 and SW2), processors (P), and users (U1, U2, and U3), which are each rendered differently. Properties of arcs and nodes can be set with a tabbed dialog, which is called up selecting “Edit Attribute Values” in the context menu. Each tab corresponds to an attribute, which is manipulated

²⁰⁵<http://mr.teknowledge.com/daml/>

²⁰⁶<http://www.daml.org>

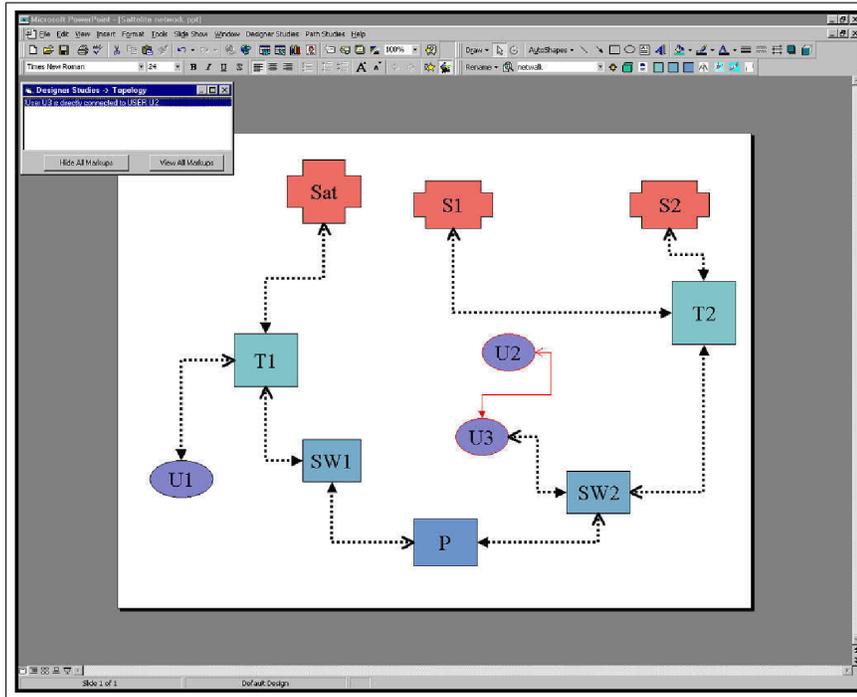


Figure 22: Visual Design Editor for satellite systems domain ([BG00a])

via text fields, radio buttons, or check-boxes, depending on the attribute's nature. Figure 23 shows the dialog to manipulate attributes of satellites. Besides the domain-specific graphical entities provided by VDE, a graph can contain any graphical entity provided by PowerPoint. This makes it possible to augment the graph, for instance, with textual and graphical annotations.

A particular domain is expressed as a graph and can be also edited within PowerPoint. In fact, the domain editor is just a particular instantiation of the VDE editor. Figure 24 shows the specification of the satellite communications domain. The domain describes the nodes and arc types (whose graphical representation is specified by associating a suitable PowerPoint autoshape with it), properties of node and arc types, name and parameters of analyses, et cetera. The domain graph is used to generate menu entries for the analyses and toolbar buttons for node and arc creation. This approach makes it easy to create an editor for a new domain.

For a domain, specific analyses can be written and made accessible to the user in PowerPoint via an entry in the “Designer Studies” or “Path Studies” top-level menus (cf. Figure 22). For instance, there is a topology analysis that checks for domain violations. These violations are reported in a separate pop-up window. Figure 22 shows a report containing a single item (upper left window in PowerPoint's canvas), which points out that “User U3 is directly connected to User U2.” This is a violation of the domain's rules. The report also highlights the violation within the graph itself; in this case, the arc that connects the

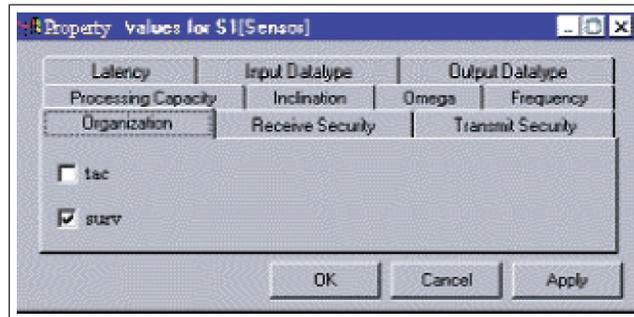


Figure 23: VDE dialog to edit satellite attributes ([BG00a])

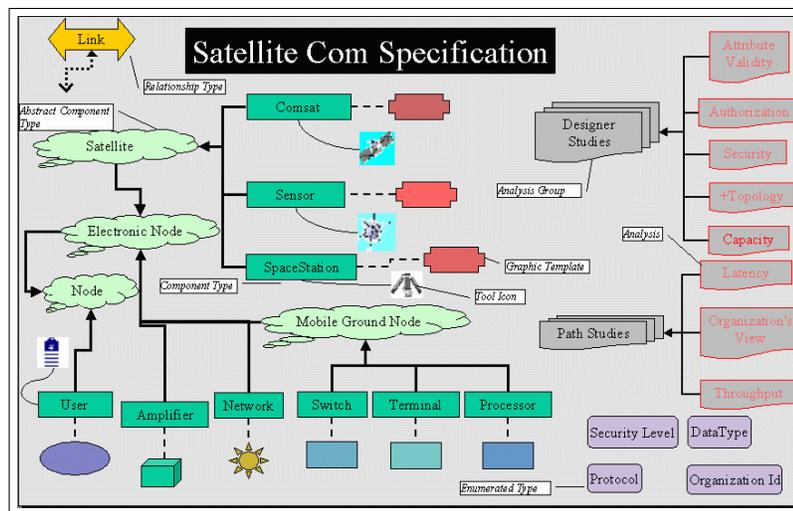


Figure 24: VDE domain specification ([BG00a])

two users is drawn in red. Latency analysis is an example of an analysis found under “Path Studies.” It computes the latency between two satellites, which have to be selected before the analysis is run. Selecting a particular latency highlights its path. The example in Figure 25 highlights the path “FortCollins9” → “Oakland2” → “ATTs0088” → “Alice1” → “Djkarta4.”

From the implementation’s viewpoint, VDE customizes PowerPoint with new pull-down menus, icons, and dialogs. These customizations add functionality to PowerPoint while leveraging PowerPoint’s existing functionality. Goldman and Balzer state [GB99]:

Implementation

“PowerPoint offers a highly functional GUI for interactively designing presentation graphics. Virtually every part of that GUI is useful, *without modification*, as part of our design editor.”

Among the reused PowerPoint functionality is scrolling, zooming, loading/saving, cut/copy/paste, and printing. Often only these standard functionalities can be reused. How-

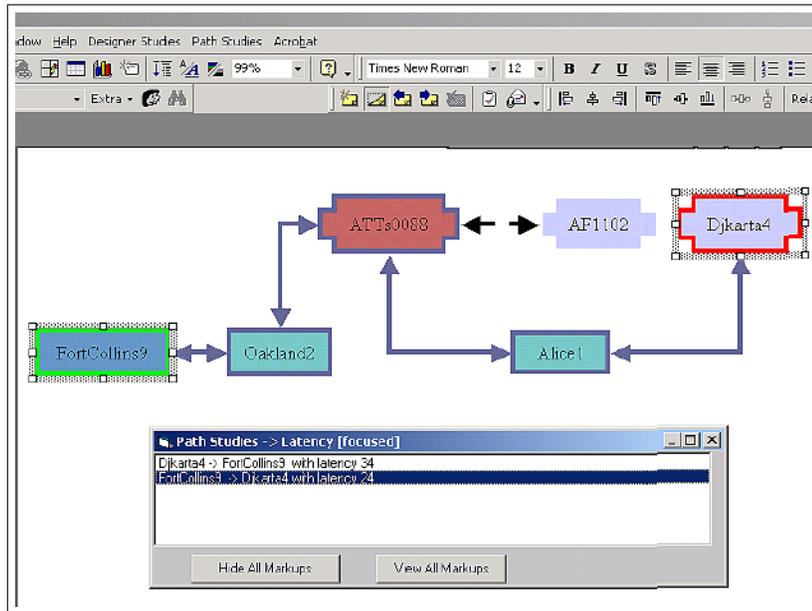


Figure 25: VDE latency analysis ([Wil01])

ever, VDE achieves a higher level of reuse because PowerPoint internally uses a graph model that can be directly leveraged by the design editor. PowerPoint’s graphical objects are VDE graph nodes, and PowerPoint’s connectors (i.e., lines that attach to other objects) are VDE arcs. Thus, VDE can also reuse operations on objects and connectors (such as deletion, selection, grouping, and aligning).

Similar benefits are realized for the Briefing Associate by using PowerPoint as a visualizer host component:

“PowerPoint gives us a higher-level platform for building a briefing tool than generic middleware such as COM/CORBA and GUI widget libraries. It offers an extensive graphical base for representing a briefing’s visual content and support for retaining nongraphical data, such as DAML markup persistently within its documents. PowerPoint also offers an extensive WYSIWYG user interface for viewing and editing a briefing’s visual content. To accommodate DAML-aware briefings, the interface requires only an extension, not a redesign or reimplementaion” [TBG02].

VDE offers domain-specific functionality, but users can also fall back to PowerPoint’s drawing functionality for annotations. The authors of VDE state that “we have found no reason to *remove* any of PowerPoint’s standard GUI.” To implement domain-specific functionality, VDE uses string/value pairs that can be attached to any (graphic) object in PowerPoint. These tags are preserved when a file is saved. Thus, the standard loading/saving functionality could be directly reused to provide persistence. The Briefing Associate ex-

tends PowerPoint's copy and paste operations, providing two different semantics [TBG02]. "Paste as new instance" creates a new graphical object, copying the domain-specific properties of the original object. In contrast, "paste as proxy" just places a reference to the domain-specific properties of the original object. This lets users represent a single object in multiple slides consistently.

Analyzers are implemented as separate processes and communicate with PowerPoint via DCOM. Each analysis is associated with a particular domain. VDE reports (incremental update) information of PowerPoint's state to the analyzers. Alternatively, an analyses can query VDE's state. Analysis results can be reported back with a number of markups, specifying, for instance, graphical objects that should be highlighted, annotated, or hidden.

VDE is implemented in Visual Basic as a COM add-in for PowerPoint. VDE acts as a COM server, receiving change events as the user modifies PowerPoint's internal state via GUI interactions; it also acts as a COM client, enabling it to manipulate PowerPoint's state via its API. Analyzers are also programmed in Visual Basic. Visual Basic is a good choice to rapidly implement communication between VDE and its analyzers because it is a COM-aware language and "trivializes the implementation of DCOM clients and servers" [GB99]. Even though Visual Basic is an interpreted scripting language, VDE's authors report that "the performance of the Visual Basic code has been acceptable to us to date" [BG00a].

To realize VDE as an extension of PowerPoint, VDE needs to be aware of state changes initiated by users interacting with PowerPoint's GUI. Examples of state-changing events are the manipulation of graphical object, or the activation of a toolbar button or pull-down menu. Unfortunately, PowerPoint's API does not expose these events.²⁰⁷ As a work-around, VDE replaces PowerPoint's native GUI object such as menu items and buttons with equivalent ones that provide event notification. VDE also monitor's PowerPoint's message queue that contains information about mouse and keyboard events, and computes deltas of PowerPoint's internal state before and after events. This allows VDE to synthesize four high-level pseudo-events (i.e., object creation/destruction and making/breaking of a connection between an object and a connector) [TB01]. The pseudo-events are sufficient to capture all topological changes made by a user editing a graph.

On the downside, PowerPoint does not always provide the flexibility and customizability that one would like to have. For example, not all GUI events can be associated with event notifications:

"An extreme example is the 'Undo' command. Although we may have control both before and after PowerPoint executes that action, we have no effective means, short of a complete comparison of before and after states, to determine the relevant state changes. The best we can do is simply remove such tools from the GUI, which is trivial" [GB99].

Upgrading to a new PowerPoint version poses the risk that the implemented extensions no longer work. This risk depends on the extension's mechanism. If the extension uses a

²⁰⁷In fact, many OTS products (e.g., Rational Rose, Microsoft Word, and Mathwork Matlab) have deficiencies in this respect [JE04] [TB01].

documented (or advertised) API there is less risk. For instance, VDE's authors say that "all our code that relies on the advertised (D)COM object model should require no change, because numerous other third party PowerPoint extensions rely on the same model" [GB99]. On the other hand, extension that rely on implementation-specific behavior might break. For example, VDE accesses events at a low-level message queue to track changes of PowerPoint's internal state. This approach is fragile "because our rules for interpreting the significance of messages in the message queue are based only on observation of the current version" and as a result "there is reason to expect they might have to be revised in potentially non-trivial ways" [GB99].

This case study shows that it is feasible to graft significant functionality on top of a COTS product in order to realize domain-specific functionality. Importantly, a significant amount of the features already available as part of the COTS product are reused as-is. Additional functionality was realized exclusively with Visual Basic, suggesting that component customization via scripting is an effective customization mechanism. The case study also shows that the use of DCOM as a wiring standard enables interoperability.

Discussion

The reported experiences with PowerPoint are of specific interest to researchers who want to customize Microsoft Office products. Since these products are based on similar technology and offer similar customization mechanisms, the experiences made for PowerPoint are potentially applicable to other members of Microsoft Office.

5.3.2 Desert

Reiss' Desert is a software engineering environment composed of several tools covering editing, content management, searching, and visualization [Rei99] [Rei96]. The tools of the environment communicate with a message-passing framework. This approach is known as control integration (cf. Section 4.2.4); it has been pioneered by Reiss in his FIELD programming environment [Rei90]. Information about source code is extracted with so-called scanners. Scanners are fault tolerant and do not perform a full parse. For instance, the scanners for C and C++ ignore the syntax of statements and expressions. Scanners are "designed to be as simple and fast as possible" and are typically implemented in less than 500 LOC [Rei99]. Extracted facts are stored in relational databases, which can be queried with SQL.

Features

Desert's goal is to increase programmer productivity. This is achieved by providing a common editor for all phases of software development that is capable of handling a wide variety of programming tasks. Furthermore, the editor presents the source code such that it is as readable as possible. Desert's editor has been implemented by customizing FrameMaker, thus "showing how a modern word processor can be extended to handle the tasks involved with software development" [Rei99]. A screenshot of the editor is shown in Figure 26.

Within Desert, the common editor based on FrameMaker is used to provide and integrate new facilities. For instance, Desert provides facilities for annotations (e.g., to mark a debugging breakpoint), hyperlinking, literate programming, et cetera. One of these facilities, called FOOD, extends FrameMaker with an object-oriented analysis based on nouns

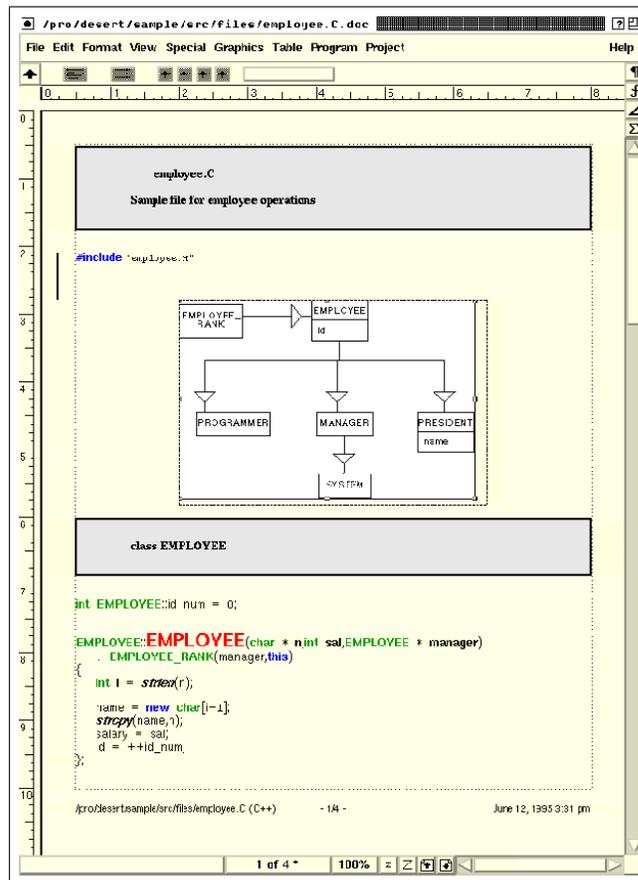


Figure 26: Desert editor ([Rei95b])

and verbs found in a natural-language specification. Nouns that occur frequently are candidates for classes; verbs acting on these nouns are candidates for methods. Figure 27 shows a screenshot of FOOD. The specification (top) is analyzed for nouns (lightly underlined) and verbs (heavily underlined). Based on the analysis, a table is constructed (bottom) that contains the proposed classes (left column) and their methods (middle column). Users can add additional information (right column), and generate OMT diagrams.

Among the features of FrameMaker that are leveraged by Desert's editor are:

hyperlinking: FrameMaker provides support to insert hyperlinks to targets in the same or another FrameMaker document. Desert uses this feature, for instance, to insert links from variable references to their definitions. FrameMaker also supports HTML hyperlinks. This allows Desert to create links for URLs given in JavaDoc comments.

text formatting: Since FrameMaker is a word processor and desktop publishing system, it has formatting features that go beyond a simple text editor.²⁰⁸ Desert takes advantage

²⁰⁸The boundary between text editors and word processors is blurry. For instance, XEmacs is considered a text

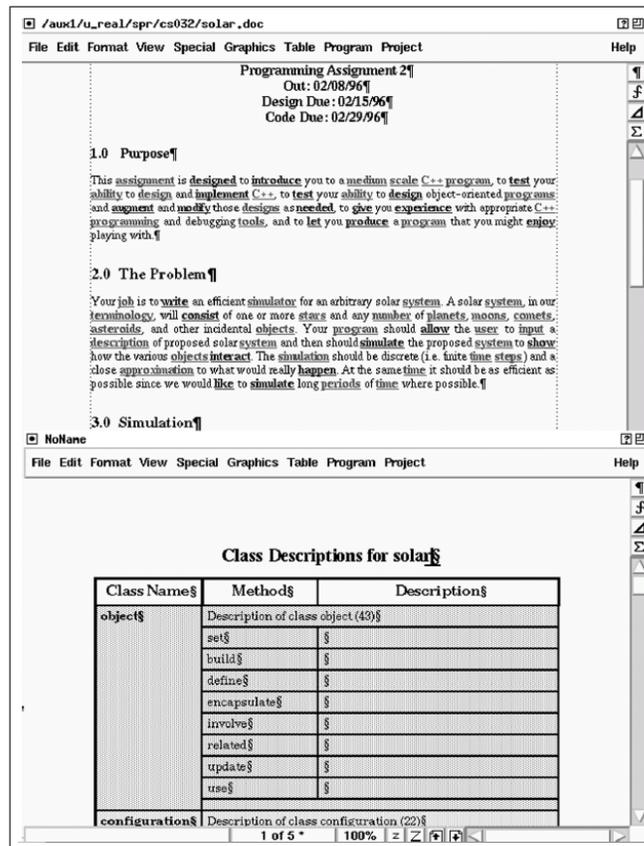


Figure 27: Desert's FOOD tool ([Rei99])

of FrameMaker's text formatting features to render source code with different fonts, font sizes, styles, and colors (cf. Figure 26, bottom).

graphics support: FrameMaker supports so-called insets, which can be used to render images in Postscript or X11 image format. Insets allow Desert to augment source code with graphical artifacts within the same document. For instance, Figure 26 shows an OMT diagram that provides a graphical representation of the inheritance relationships of the EMPLOYEE class.

Reiss concludes that

“FrameMaker provides many of the baseline features we needed: it displays both pictures and text; it can display high-quality program views; and many programmers are familiar with it, using it for documentation or specifications” [Rei99].

Desert accomplishes customization of FrameMaker with the FrameMaker Developer Implementation editor, but has capabilities to render text in different (variable-sized) fonts, styles, and colors.

Kit²⁰⁹ (FDK). The FDK includes a set of C libraries and API header files that allow to write extensions in C and C++. The API provides services for initialization requests, event notifications, editing commands, and messages. Desert mostly relies on FrameMaker's event notification, which invokes a callback function after each command when control is being returned to the user. However, this callback does "not provide any information about what the command did or what was changed" [Rei95b].

To realize automatic, on-the-fly indentation and formatting of source code, the scanners have to process the input as it is entered. Lines are assigned a different FrameMaker paragraph format depending on their contents (source code, preprocessor directive, comment, and blank line). Block comments are rendered with shaded backgrounds of different colors (e.g., gray for information, and light red for warnings). To render a comment with a shaded background, it has to be embedded into a one-row, one-column table. Obtaining the text to scan is not straightforward because of limitations in FrameMaker's API. Changes made to text are detected by analyzing the cursor position before and after a user command. If a change cannot be reconstructed, the whole text is rescanned. This is the case, for example, after a global find/replace command. FrameMaker usually saves files in a proprietary binary format. These files cannot be processed by other tools that are part of Desert. To realize an open environment, filters are responsible for converting FrameMaker files to other formats and vice versa.

Even though Desert is in the stage of a prototype tool, it is designed to handle large systems with over a million LOC. Desert has been used on its own source code, which is about 250,000 lines of C++ code. The FRED program-editing interface (which is the largest of Desert's components) is about 30,000 LOC. Even though Desert's implementation "is heavily dependent on FrameMaker, the concepts and techniques we use can be applied to other editor frameworks" [Rei95b]. More specifically, Reiss also says that "most of the extensions that we provide for FrameMaker could have been done for any other word processor with a suitable API and capabilities. (Microsoft Word, for example, could be used on a Windows platform.)" [Rei99]. Thus FrameMaker's customization can be seen as a proof-of-concept that an editor such as desired by programming environments can be realized based on an OTS product.

Reiss discusses the limitations of FrameMaker with respect to the implementation of Desert:

"Insets in FrameMaker are somewhat primitive when compared to Microsoft's ActiveX and similar technology. FrameMaker's cross-reference links are also somewhat restricted, being difficult to activate and define. Comments in the program display had to use simple tables to achieve the desired background. This made the design and implementation of the program-editing front end more complicated than necessary. Annotations were also difficult to integrate into the document-oriented philosophy of FrameMaker. Finally, because FrameMaker essentially controlled the display, some of our user interface op-

²⁰⁹<http://partners.adobe.com/public/developer/framemaker/devcenter.html>

tions are not optimal. For example, the dialog box for symbol completion would probably be better using a pop-up menu or cue, as in Visual C++” [Rei99].

There are also performance problems. Whereas interactive editing of files is sufficiently fast, loading/saving of files as well as reformatting of a whole document can cause delays. Still, the implementation has “gotten to the point where the editor is usable” [Rei99].

Desert is probably the first tool that extensively customized the GUI of a WYSIWYG editor to realize a software engineering tool. Reiss went through the effort to describe Desert’s implementation in detail, providing valuable experiences, and allowing other researchers to get productive quickly when customizing FrameMaker for similar purposes. However, to my knowledge Desert has not been used to build other software engineering tools.

Discussion

The customization of Emacs to provide software engineering functionality is much more popular among researchers. An explanation may be that Emacs is already extensively used for programming, whereas FrameMaker is primarily used to generate high-quality documentation. Tool developers may find it difficult to recognize a potential host component if it is not already used within their domain. Furthermore, many researchers have already experiences in customizing Emacs using its scripting language. Since most of Emacs’ functionality is realized with scripts, there is a large body of existing code that provides customization samples. In contrast, FrameMaker is customized with a compiled language, which prohibits rapid prototyping of functionality.

The editors of modern IDEs such as Eclipse now provide functionality that rivals and exceeds the capabilities of FrameMaker. These IDEs are attractive for tool developers because they can be often programmatically customized, and provide dedicated functionality for programming (e.g., symbol completion) that is not already available for general-purpose editors.

5.3.3 Galileo

The last case study of component-based tool-building that I discuss is Galileo. Galileo²¹⁰ is a fault tree analysis tool that is used to model and assess the reliability of complex systems (e.g., software controlling missiles or nuclear power plants) [CS00b] [CS00a] [SCZC97] [SK96]. A fault tree is a rooted, directed graph that estimates system failure probabilities. Its leaves represent basic events of failures, giving a probability of occurrence; internal nodes represent gates that propagate the events according to certain rules. The top-level gate corresponds to a system failure.

Features

Galileo supports a graphical and a textual notation of the fault tree; both are equally expressive. In the graphical notations, the fault tree is visually rendered as a hierarchical tree structure. Gates and events are represented with different symbols that are connected with directed arcs (cf. Figure 28, top right). The textual representation is a declarative

²¹⁰<http://www.cs.virginia.edu/~ftree/>

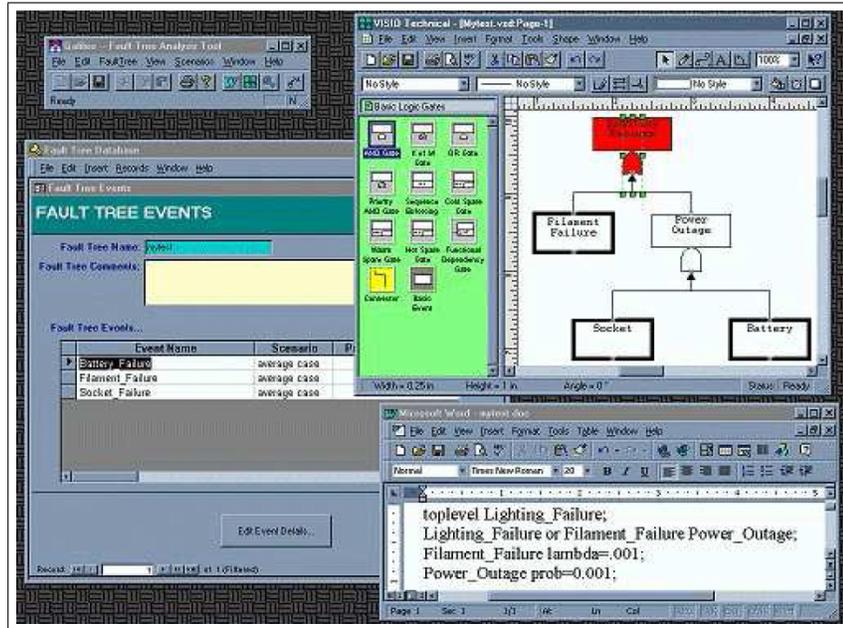


Figure 28: Early version of Galileo tool ([CS98])

domain-specific language (cf. Figure 28, bottom right). To evaluate the fault tree, Galileo relies on an external, Unix-based tool, called DIFTree.

Galileo leverages Microsoft Word, Visio, and Internet Explorer. The latter is used to provide hypertext-based documentation. The graphical and textual representation of the fault trees is presented to the user with Visio and Word, respectively.

An earlier version of Galileo did also use Microsoft Access 97 to realize persistence and concurrent access to multiple fault trees as well as to generate reports [SCZC97] [SKC⁺96] [SK96]. The use of Access was later dropped because it did not support a tighter integration implemented in a later version of Galileo [CS00b]. Galileo's early version is realized as a set of independent OTS components that are coordinated by a top-level component. Each component has its own GUI window. Figure 28 shows a screenshot of the components that make up Galileo: Access for data management (bottom left), Word for text editing (bottom right), Visio for graphical editing (top right), and the top-level component for coordination (top left). The coordinator manages starting/terminating of the OTS products and allows the user to manage consistency among OTS products via updating and transferring of data.²¹¹

Galileo now presents the user a single top-level GUI window that contains the OTS products. Figure 29 shows a screenshot of this approach with Visio (top left), Word (bottom left), and Internet Explorer (right). Galileo's top-level tool-bar incorporates tool-bar elements from Visio, Word, or Explorer, depending on which one is currently active (Visio in Figure 29). A user editing the active view of a fault tree (either in Visio or Word), has

²¹¹This approach to tool interoperability is similar to Star (cf. Section 3.2.2).

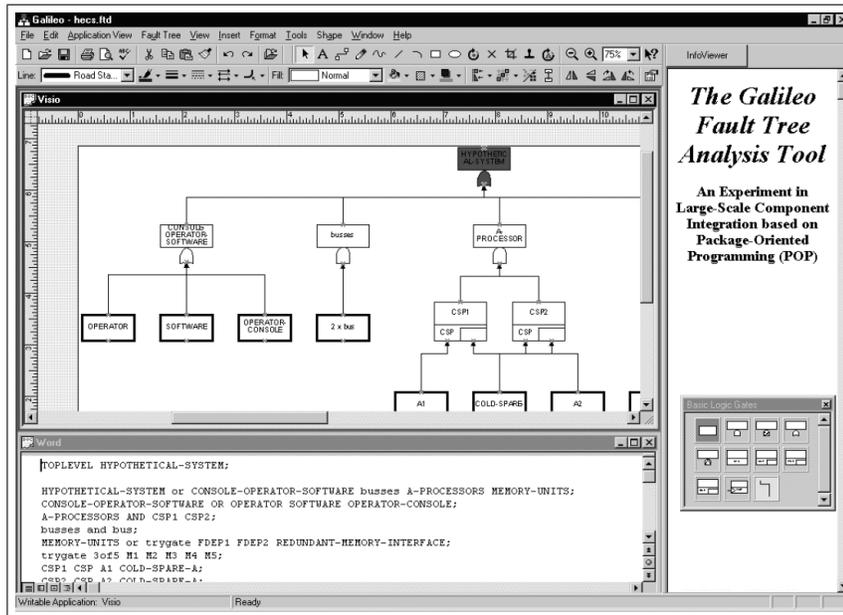


Figure 29: Galileo tool ([CS00b])

to explicitly issue a command to synchronize the data with the inactive view. For instance, after the user has finished editing the graphical representation in Visio, the textual view can be updated (and optionally activated) with a menu item (cf. Figure 30). The inactive view is visually highlighted with a red rectangle drawn around its window; this warns users to not edit the inactive view (which is still modifiable).



Figure 30: Switching of active views in Galileo ([SDC99])

Using Visio, the user creates graphical fault trees by dragging shapes from a custom stencil to the drawing area. Connections between shapes are made via Visio connectors. Early implementations of Galileo relied on Visio's built-in functionality to create fault trees. However, users complained that this was tedious because several low-level editing steps were necessary to achieve a high-level task such as adding a new gate to the tree. Galileo now implements custom operations to simplify such tasks (e.g., one can select a gate and then, by pushing a button, add and connect a new gate to it). Galileo also customizes menus of its components to prevent the user from terminating them independently. Furthermore, Word is customized to disable grammar checking on fault tree specifications.

There is also a later incarnation of Galileo, called Nova, which incorporates Excel as an additional component [CS03]. Excel is used to edit properties of fault-tree events. Nova is implemented with more advanced customizations of the OTS products compared to Galileo. For instance, Nova provides automatic syntax highlight of keywords in Word. Custom operations for fault tree editing are implemented with the components' native scripting support. Fault tree shapes in Visio have custom behavior—"connectors automatically 'hop' over each other, shapes move automatically to prevent overlap, and text boxes expand to accommodate long label names." Nova adds menu items and toolbar buttons for domain-specific functionality such as selecting a subtree, and provides visual feedback to indicate whether a connector is properly attached to a gate. Nova also removes or modifies inappropriate Visio functionality.

Galileo is a case study for an approach to CBSE called package-oriented programming (POP) [CS00b]. POP is based on "the use of multiple, architecturally compatible, mass-market packages as large components" [CS00b]. The use of multiple components is motivated by the observation that many tools require baseline functionality in multiple independent domains such as text editing (e.g., provided by Word) and graph editing (e.g., provided by Visio). This approach is in contrast to the Visual Design Editor and Desert, who focus on a single component (i.e., PowerPoint and FrameMaker, respectively). However, POP uses components that are architecturally compatible to simplify integration and to minimize architectural mismatch. For instance, both Word and Visio support Microsoft's COM and Active Document Architecture.

POP

An explicit goal for Galileo is the objective to create an industrially competitive system. Galileo's authors elaborate on this goal as follows [CS00b]:

"We adopted the 'industrially competitive' goal not because we want to sell a product, or because our research is short-term, but because unless such an objective is accepted it is too easy to miss problems that will be encountered in practice. Adopting a 'product' goal, rather than building a toy system, is thus important to avoiding 'false positives' in an assessment of the potential of the POP model."

Galileo has been continuously improved based on user feedback from industry. Galileo is available for free on the Web and hundreds of industrial users have downloaded it. The authors state that they "have interacted with industrial users to gain feedback on shortcomings, leading to requirements for subsequent iterations" [CS00a]. For instance, one industrial user pointed out the need for hierarchical drawing on multiple pages. As another example, an early version of Galileo was developed based on an evaluation from an engineer at Lockheed Martin. This evaluation exposed missing requirements in terms of usability and functionality. Detailed feedback was also obtained from NASA's Langley Research Center, which has expressed interest to develop a version of Galileo that is suitable for industrial use. 16 NASA engineers evaluated Galileo with a survey that asked about tool requirements and users' perceptions of its features [CS03]. The respondents identified "an easy-to-use user interface" as the most important quality attribute (above speed). Galileo's

User feedback

editing capabilities (i.e., the combination of native functionality and domain-specific customizations) were judged as being “the same or better than other tools.” Except for one respondent, Galileo’s usability was perceived as being the same or better compared to other tools. Especially, most respondents indicated that “the Word- and Visio-based views were easy to use.” Generally, the results from the survey are encouraging, indicating that leveraging of existing components can product competitive tools. Galileo’s authors say the following:

“End user evaluation is a key arbiter of success in the use of POP for the construction of tools. Despite the ‘beta’ status of Galileo, the surveys indicate that the Galileo tool meets or exceeds the expectations of engineers. In most respects the tool was judged comparable with commercial tools” [CS03].

The early version of Galileo implements most functionality in its top-level coordinator, which is written with Visual Basic and Visual C++ (cf. Figure 28) [SK96] [SCZC97]. The coordinator interacts with the OTS products (Visio, Word, and Access) that it controls through an abstraction interface that hides and insulates the OTS products’ APIs from the coordinator. The coordinator manages the associations between different views of the fault tree (e.g., it associates Visio shapes with corresponding Access database records) and ensures consistency of the views (e.g., adding of a new Visio shape is reflected in Access by adding a corresponding database record) [SK96]. The coordinator indirectly manipulates the components through Microsoft’s OLE, acting as an OLE Automation client. The coordinator has a menu item that users have to activate to synchronize views. View synchronization is accomplished by querying components for their current state and comparing them with the other component’s states. Implicit synchronization of the views turned out to be infeasible. While OLE supports registering for call-backs with components to detect state changes, Access 3.0 did not yet provide this capability. Visio 3.0’s event notification turned out to be incomplete, because it lacks a notification for shape deletions. Also, Visual Basic 3.0 lacks functionality to realize call-backs. Another problem was the lack of unique shape identifiers in Visio. As a workaround, own identifiers are maintained in shape objects, which provide a number of slots to store arbitrary data. Despite these difficulties, the Galileo’s authors feel that “a useful tool was built quickly and at low cost” [SK96].

Implementation

The recent version of Galileo uses the Microsoft Active Document standard to integrate Word, Visio, and Internet Explorer into a single window (cf. Figure 29) [CS00b]. This Galileo version ceased to use Access because at that time Access did not support Active Documents; instead of Access, the file system is used to store fault trees. On the other hand, Internet Explorer did support the standard, enabling its integration into Galileo. Galileo has several undesirable features caused by limited component customizability. For instance, the inactive fault tree view can be modified because neither Word nor Visio provide a “ready only” mode. While it is possible to prevent GUI events from reaching components, this also disables desirable functionality such as document navigation.²¹² Galileo’s authors

²¹²Marchukov and Sullivan discuss an enhancement to Galileo that selectively filters GUI events at the operating-

also encountered unexpected component behavior (i.e., bugs). For example, programmatic manipulation of multiple pages in Visio did not work for Active Documents. A prototype implementation of multi-page fault trees was not able to uncover this problem because it was built as a standalone application. Since there was no apparent workaround, Galileo's authors lobbied the component vendor to fix the problem:

“During our discussions with Visio Corporation, it became clear that to get the functionality we needed we had to provide a rationale in business terms. After we described our funding from NASA Headquarters under our agreement with NASA Langley Research Center, they agreed to fix the problem in the next release. We continued to develop our tool to use the page manipulation functions, in the expectation that a suitable version would become available, which it did” [CS00b].

The implementation of Galileo did uncover undocumented limitations in Visio. For instance, Visio 4.1 and 5.0 malfunctions if a client holds more than 700 references to shape objects. In this case a workaround was possible that looks up references on demand rather than keeping them stored in the client. However, the on-demand lookup causes additional runtime overhead.

In a time frame of four years of developing Galileo, the leveraged components have evolved significantly. Galileo has upgraded to new versions steadily to take advantage of new features and bug fixes. Whereas it takes additional programming effort to take advantage of new capabilities, upgrading to new components rarely caused problems—“new components, in our experience, have offered an instant, low-cost upgrade path” [CS99]. However, not all upgrades are painless; upgrading to Visio 2000 failed because the new API is presumably not backwards compatible [CS00b].

Nova realizes more sophisticated customizations of components compared to Galileo [CS03]. For instance, a custom operation (accessible as a menu item and toolbar button) allows the user to change the type of fault tree shapes. This operation first checks that a shape is selected and then displays a dialog box to change the shape's type. Changing the type involves deletion of the old shape, creation of a new one, and appropriate rewiring of the old shape's connections. Such customizations are implemented in Visual Basic with 3,100 LOC for customizing Word and 8,800 LOC for customizing Visio. Interestingly, Nova still relies on explicit view synchronization (introduced early in Galileo), presumably because the components' event notification is still lacking in functionality or convenience.

Galileo is a case study in tool construction that integrates two COTS products to realize graph-based and textual editing of information. Since two disparate components provide views of common data, a synchronization mechanisms had to be realized. This complicates tool construction compared to solutions that customize a single component only.

Galileo's implementation uses interfaces that abstract away from the actual COTS components. This introduces a level of indirection that makes it easier to switch versions of the system level to prevent modification of the inactive view [MS99]. However, this approach “is difficult and non-intuitive” [CS00b].

Discussion

same component, or to switch to a different component. Whereas the construction of such an interface increases the development effort, it mitigates the problem of vendor lock-in [BMMM98].

The obtained user feedback suggests that Galileo is an industrially competitive tool. Leveraging of familiar COTS products can have both negative (e.g., performance) and positive (e.g., usability) consequences. The implementors of Galileo realize that they are pursuing an approach to tool-building that has different trade-offs compared to implementing a tool from scratch. In contrast to other researchers, they discuss the impacts of their tool-building approach on developers and tool users in more detail.

5.4 Discussion

The catalog of applicable host components that can be leveraged by tool-builders to construct visualizers and extractors for the software engineering and reverse engineering domain shows the great potential of a component-based tool-building approach. The fact that there are a number of tool-building examples for each of the identified host components shows that researchers have recognized the potential of leveraging components and have started to exploit it. For instance, Tallis et al. explicitly emphasize that

Feasibility of
leveraging
components

“the Briefing Associate, like the Design Editor, is implemented as an extension of Microsoft PowerPoint. We regard this choice not as an implementation detail, but as central to this research” [TBG01].

Unfortunately, few researchers bother to report their tool-building experiences. It is often difficult or impossible to understand how researchers have built a tool. The fact that a component has been leveraged is often touched on only in a single sentence. The following examples are from papers that devote only a few meagre sentences to address the tool’s implementation:

Lack of published
experiences

- “The updates in the documents are detected by a Notes script and written to a log” [JQB⁺99].
- “We are trying to build up the system with sole Lotus Notes products such as Notes Formulas, LotusScript and Lotus Components to ensure the ease of development and maintenance” [CB99].
- “To evaluate their criteria, the authors have developed a proof-of-concept test data generation tool, called UMLTest, which is integrated with the Rational Rose tool” [CT01].
- “A prototype version of the IBM VisualAge C++ compiler is used as the front end. The abstract syntax tree constructed by the front end is transformed into a PCG and a CFG for each function, which serve as input to the alias analyses” [HBCC99].

- “Using a commercial tool Rose Real Time as a front end for our tool we transform the visual UML model to a textual format data. We use Rose Script to transform UML diagrams to textual data” [WHG⁺03].
- “Even if Xrefactory is using a professional C++ front-end provided by EDG it is still very difficult to take care of all special cases defined in the ANSI standard” [VBM05].
- “We use EDG front end compilers to parse source code and produce ASTs” [GPB04].

One reason for this lack of detail might be the perception that such experiences are not part of the publishable research results. I believe, however, that published experiences are valuable, greatly benefit other researchers, and constitute a first step towards a better and more formal understanding of tool-building issues.

In the context of this dissertation, the lack of published experiences is frustrating. It also shows that tracking down evidence of component-based tool-building is tedious and difficult to accomplish. However, there are a few examples of researchers who have explored component-based tool-building in more depth and have published their experiences in some detail (e.g., cf. Section 5.3).

Unfortunately, almost all of the published tool-building examples lack a conscious exploration of their approach in the sense that there is no reflection or discussion of the impact that component-based tool-building has on the development effort and the tool users. The authors of the Dali architecture reconstruction tool note that

Tool-building is ad
hoc

“the use of commercial tools has provided a great deal of leverage during the development of Dali: these tools are typically robust, well documented, and well tested” [CK98].

O’Brien makes a recommendation based on his experiences with two fact extractors—unfortunately without further elaborating on how he reached his conclusion:

“Both Imagix-4D and SNIFF+ tools were applied to extract views from the source code. After analyzing the output from both tools, we concluded that the Imagix-4D was more useful for extraction purposes” [O’B01].

Whereas such reported experiences provide valuable information and leads for other researchers to follow up on, they lack sufficient detail. Consequently, tool-building continues to be approached in an ad hoc manner, lacking general solutions. There are a few exceptions; for instance, Egyed et al. have proposed approaches and architectures to integrate OTS products [EMG00] [EB01] [Bal03] [EJB04] [JE04]; Balzer et al. have developed a technology for mediating shared library calls to customize OTS components [TB01] [BG00b] [BG99]; and Hepner et al. have identified recurring component interoperability problems and solutions, encoding them as conflict patterns [HGK⁺06]. The emergence

of general techniques and approaches to achieve CBSE using OTS products indicates the beginning of maturation for this field.²¹³

The identified tool-building examples suggest three prominent platforms: Web browsers, Microsoft Office (including Visio), and Eclipse. Web browsers offer a ubiquitous computing platform that is highly familiar to users. Technologies such as HTML, JavaScript, SVG, and applets allow the rapid development of sophisticated Web applications that are portable across browsers. Microsoft Office offers a diverse set of popular OTS-product functionality for data persistence and querying, spreadsheet computations, charts, text formatting, and graph editing. Functionality of individual OTS products can be customized with a convenient scripting environment; and OTS products can be made interoperable via OLE and COM. Eclipse is a plug-in based IDE implemented in Java that is highly customizable. It provides APIs for fact extraction (e.g., Java and CVS) as well as seamless presentation integration. The research community has embraced Eclipse enthusiastically.²¹⁴

Popular components

5.5 Summary

This chapter introduced the characteristics of the host components that provide the building blocks for component-based tool-building. Examples of host components that meet the introduced characteristics were then discussed along with tool-building examples. The number of existing tools shows that it is both possible and practical to build tools on top of these host components.

Unfortunately, most researchers do not discuss in sufficient detail how they accomplished building their tools, and do not relate their experiences and lessons learned—leveraging of host components is seen as a means to an end, which is not worthwhile describing. However, fortunately there are a few examples of researchers who describe their tool-building approach in more detail. In the next chapter, I follow the example given by these researchers, providing a detailed account of my own tool-building experiences.

²¹³A similar maturation can be observed for grammar-dependent software such as fact extractors. Fact extraction is still ad hoc, but researchers have started to recognize the limitations of this and call for a *grammarware engineering* discipline that “is subjected to best practices, tool support and rigorous methods” [KLV05].

²¹⁴This is evident from recent conferences as well as dedicated Eclipse workshops [BMS05] [BdM04b] [BFBS04] [BGSS03]. For instance, ICSM 2005 had four papers in the Program Comprehension track; of these, one leveraged Columbus [FBLL05], one LSEdit [KG05], and two Eclipse [EHMS05] [SES05].

6 Own Tool-Building Experiences

The previous chapter discussed many examples of tools that have been built leveraging components. This chapter describes the reverse engineering tools that I have been involved in building. My case studies serve several purposes. They augment the existing body of knowledge with additional tool-building experiences. In contrast to most published experiences of other researchers, my case studies provide a more detailed treatment of implementation issues and reflect on the impact of using a component-based tool-building approach. Furthermore, the case studies increase the credibility of my recommendations and lessons learned (cf. Chapter 7).

The tools of the case studies have been constructed as part of the Rigi group’s ACRE projects over a period of roughly five years (2001–2006). ACRE explores tool-building approaches to make software engineering tools more adoption-friendly (cf. Section 3.2.5). Its main assumption is that in order for new tools to be adopted effectively, they must be compatible with both existing users and existing tools. Research tools are lacking in this respect because they are often built from scratch with limited resources and expertise, resulting in idiosyncratic, stand-alone tools. ACRE proposes to increase adoptability by leveraging OTS components and products that are popular with targeted users.

Tool	Host component	Implementors	Publications
REOffice	Excel, PowerPoint	Yang, Weber	[WYM01] [Yan03]
SVG graph editor	SVG	Kienle	[KWM02]
REVisio	Microsoft Visio	Zhu, Chen	[ZCK+03] [Che06]
RENotes	Lotus Notes	Ma	[MKK+03] [Ma04]
REGoLive	Adobe GoLive	Gui	[GKM05b] [GKM05a] [GKM05c] [Gui05]
WSAD extractor	IBM WSAD	Kienle	[KM06]

Table 9: Summary of own tool-building experiences

Table 9 provides a brief summary of my tools, characterizing the leveraged host component, the primary implementors, and the published research results. Most of the tool descriptions in the following sections are drawn from published research papers; these publications are indicated in **bold** in Table 9. Since these publications provide a “snapshot” of the Rigi group’s research results and understanding of component-based tool-building at the time, I have decided to incorporate most of their contents verbatimly. As a result, some sections repeat points already made. Consequently, the tools are described in chronolog-

Case studies

ical order of their development. I believe that the implemented tools represent interesting coordinates in the design space and augment other researchers' tool-building experiences.

Microsoft Office is a popular target for component-based tool-building. However, there are still not many reverse engineering tools that have customized Office components. Thus, REOffice and REVisio augment the more generic tool-building experiences of other tool builders with more specific experiences in the reverse engineering domain.

REOffice and
REVisio

The SVG graph editor leverages a Web standard introduced by the W3C to embed interactive (or *live*) documents into diverse documents (e.g., Web pages, PowerPoint presentations, and Word documents). Similar to HTML-encoding of text, which can be rendered in Web browsers, Word, Outlook, and Lotus Notes, SVG-encoded vector graphics can be embedded and rendered in diverse host applications. Thus, by leveraging a popular, standardized graphics format, the same reverse engineering information can be rendered in different contexts of use (e.g., architectural meetings discussing a graph presented in PowerPoint, or a software maintainer browsing a graph embedded in an HTML design document rendered in a Web browser).

SVG graph editor

Lotus Notes has been mostly customized in the context of industrial groupware and email solutions; to my knowledge, there are no research tools that have customized Lotus Notes to realize reverse engineering functionality. RENotes shows that Lotus Notes is sufficiently customizable to realize a rudimentary reverse engineering tool.

RENotes

Similar to RENotes, REGoLive targets an OTS product for which little or no tool-building experience exists in the reverse engineering domain. Furthermore, whereas many IDEs have been customized to realize functionality for traditional reverse-engineering tasks, REGoLive is the first attempt to customize a Web authoring tool to provide functionality for the reverse engineering of Web sites.

REGoLive

The WSAD Web site extractor is based on IBM's Websphere Application Developer, which in turn is built on top of Eclipse. The extractor's schema is defined with the Eclipse Modeling Framework. To visualize the extracted facts SHriMP's Eclipse plug-in is used. Thus, my reverse engineering tool is realized by programming against the Java APIs of three third-party Eclipse plug-ins: a research graph visualizer, an open source modeling environment, and a commercial Web authoring tool. The extractor has been developed as a response to industrial need—the reengineering of the IBM CAS Cassis Web site, which has been developed as a J2EE Web application.

WSAD extractor

Most software reverse engineering tools target traditional software systems written in one or several high-level programming languages. My own tool-building experiences in the domain of traditional reverse engineering encompass REOffice, REVisio, and RENotes. However, reverse engineering techniques can be applied not only to traditional software systems, but also to hypertext systems. This is the case because both have similar characteristics; they also share the problems of maintenance and evolution of existing systems [BBH98]. Over the last few years, dedicated reverse engineering tools have started to emerge for the largest hypertext system in existence—the World Wide Web [WBM99]. Indeed, with the ever growing complexity of Web sites, the need for Web site reverse en-

WSRE

gineering (WSRE) is also increasing. It seems that besides traditional reverse engineering, WSRE is the next most important domain for reverse engineering research. This is apparent by the number of publications as well as the existence of a dedicated workshop series, the annual *IEEE International Workshop on Web Site Evolution*.²¹⁵ I address WSRE with two of my tools, REGoLive and the WSAD extractor.

6.1 Rigi as Reference Tool

As a member of the Rigi group, I have accumulated considerable knowledge in the design and implementation of graph editors for the visualization of software structures. It seemed natural to pursue my approach of component-based tool-building by customizing host components with the goal to realize functionality that is similar to Rigi's. In a sense, Rigi can be seen as the *reference tool* whose functionality is (partially) reimplemented by my case studies. This approach has several advantages:

proven functionality: The Rigi tool—developed and maintained for over a decade, and used on several industrial projects—has proven its usefulness for program comprehension and reverse engineering. A tool that implements similar functionality can be expected to be equally useful.

existing domain expertise: I already have acquired the necessary background on how to build a Rigi-like graph editor. Since I have the necessary domain knowledge such as algorithms, data structures, and visualization techniques, I can concentrate on the new technologies and tools that I need to understand for pursuing component-based tool-building.

stable requirements: The functionality of Rigi is well understood and documented [Won98]. Thus, my case studies can concentrate on reimplementing Rigi's functionality in the context of a new tool-building approach, rather than having to struggle through elaborate requirements elicitation first. I believe that it is highly desirable to have stable requirements when experimenting with a novel tool-building approach.

tool comparisons: Since my case studies have similar functionality as Rigi, a meaningful comparison of the tools' characteristics is feasible. Examples of comparisons are the tools' implementation effort and graph rendering speed.

To better understand the case studies that realize functionality similar to Rigi, I briefly introduce Rigi in the following.

The core of Rigi is a graph editor/drawing tool enhanced with additional functionality for reverse engineering tasks. Rigi's core functionality is similar to the functionality offered by graph editors. Graphs can be loaded, saved, and layouted; the windows rendering a graph can be scrolled and zoomed; the graph's nodes and arcs can be selected, cut, copied, and pasted; and so on. Graphs in Rigi are typed, directed, hierarchical, and attributed.

Rigi's functionality

²¹⁵<http://www.websitesevolution.org/>

The types of nodes and arcs are described in Rigi's domain model. Examples of reverse engineering functionality are the computation of cyclomatic complexity, the navigation to the underlying source code that is represented by a node, and standard domain models for several programming languages.

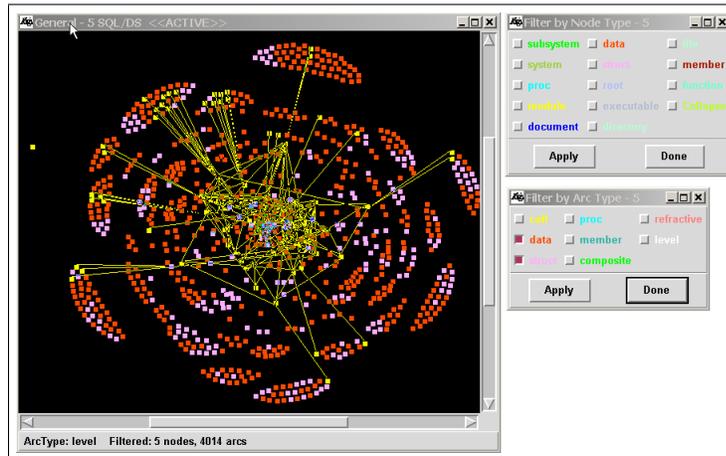


Figure 31: Software structure graph in Rigi

Figure 31 depicts a graph that visualizes parts of IBM's SQL/DS system, which is over one million lines of PL/AS code [WTMS95]. Red nodes are variables, yellow nodes are modules, and purple nodes are types. All arcs have been filtered out except for the yellow ones, which represent calls between modules (i.e., Figure 31 shows the call graph). The full graph contains 923 nodes and 2395 arcs, but in this filtered view only 189 yellow arcs are actually visible.

6.2 REOffice

The main result of the first case study conducted within the ACRE project was to establish the technical feasibility of providing sophisticated reverse engineering functionality via programmatic customization of OTS products. The implemented reverse engineering tool realizes a subset of Rigi's graph visualization functionalities; the customized OTS products are Microsoft Excel and PowerPoint.²¹⁶

In the following, features of Excel and PowerPoint for customization are discussed first, followed by an explanation of the functionality and implementation of the realized reverse engineering functionality.

²¹⁶Two members of the Rigi group, Martin and Kastelic, have also developed a pilot prototype based on OpenOffice Impress (which is an application to develop presentations similar to PowerPoint) [MK01]. This prototype showed that rendering of graphs with Impress is feasible. However, it was decided to abandon the prototype and to bundle the efforts on customizing PowerPoint instead. This decision was influenced by PowerPoint's superior scripting support.

6.2.1 Excel and PowerPoint

There are various reasons why Excel and PowerPoint have been chosen as host products for the case study. Microsoft Office has a huge user base.²¹⁷ Among the most popular products of this office suite are Word, PowerPoint, and Excel. Thus, many users have already installed Office on their machines and use it proficiently in their work. It is reasonable to expect that PowerPoint users know how to create and manipulate graphical objects, how to create and rearrange slides, how to import Excel graphs into their presentation, and so on. Similar assumptions can be made for Excel. However, because of the large functionality offered by Office applications it is difficult to make assumptions about the proficiency of target users.

User base

Both Excel and PowerPoint already provide existing functionality that can be leveraged when implementing reverse engineering tools. Excel is well suited to represent tabular information, to calculate metrics, and to visualize numeric data with charts. It has hundreds of embedded mathematical functions for complex computations, dozens of options to layout cells, and more than 10 different chart types. PowerPoint provides a variety of graphical entities that can be used to construct software diagrams and graphs. It has dozens of basic shapes, block arrows, and flowchart symbols as well as shape connectors that are routed straight, square, and curved. Each graphical entity has a large number of different rendering options for colors and styles. As a result, PowerPoint can accommodate the requirements of diverse diagrams and graphs. Since Excel and PowerPoint have been continuously maintained and enhanced over several releases, they can be seen as mature implementations with stable functionality.

Baseline

functionality

Office applications are end-user programmable via a macro recorder and built-in Visual Basic IDE. Each Office product exposes its internal state with a so-called *object model*. The object models can be programmatically queried and manipulated. Furthermore, Office applications can integrate with each other using so-called (OLE) automation. Programmatic customization of Office is simplified because each application offers Visual Basic for programming and exposes a similar object model. In the following the object models of Excel and PowerPoint are briefly introduced. The object models are documented in the Microsoft Developer Network (MSDN) Library,²¹⁸ and can be interactively explored with Visual Basic's Object Browser.

Programmatic

customization

An instance of an object model represent the internal state of the application as a set of interconnected objects. Each object belongs to a certain type. Depending on the type, the object has certain methods and properties. There are also collection objects that hold an array of objects of the same type. Collection objects such as `Shapes` offer a convenient way to iterate over all of its members. The following Visual Basic code iterates over all

²¹⁷Microsoft's share of the market for office productivity software is estimated around 90 percent [Kuc05]. Microsoft estimates that there are 300 million Office users [Bec02]; OpenOffice claims about 40 million users [Bri04].

²¹⁸<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/modcore/html/deovrMicrosoftPowerPoint2000.asp>

shapes of the active slide and sets the shapes' color to magenta:²¹⁹

```

Set aSlide =
    PowerPoint.Application.ActiveWindow.Selection.SlideRange
For Each aShape In aSlide.Shapes
    With aShape
        .Fill.Visible = msoTrue
        .Fill.Solid
        .Fill.ForeColor.RGB = RGB(255, 0, 255)
    End With
Next

```

The objects in a model reflect the (implementation) domain of the application. For example, the PowerPoint domain model has objects such as `Presentation`, `Slide`, and `Master`, whereas Excel has objects such as `Worksheet`, `CellFormat`, and `Chart`.²²⁰ Figure 32 shows an excerpt of PowerPoint's object model with objects shown in yellow and collections in blue.²²¹

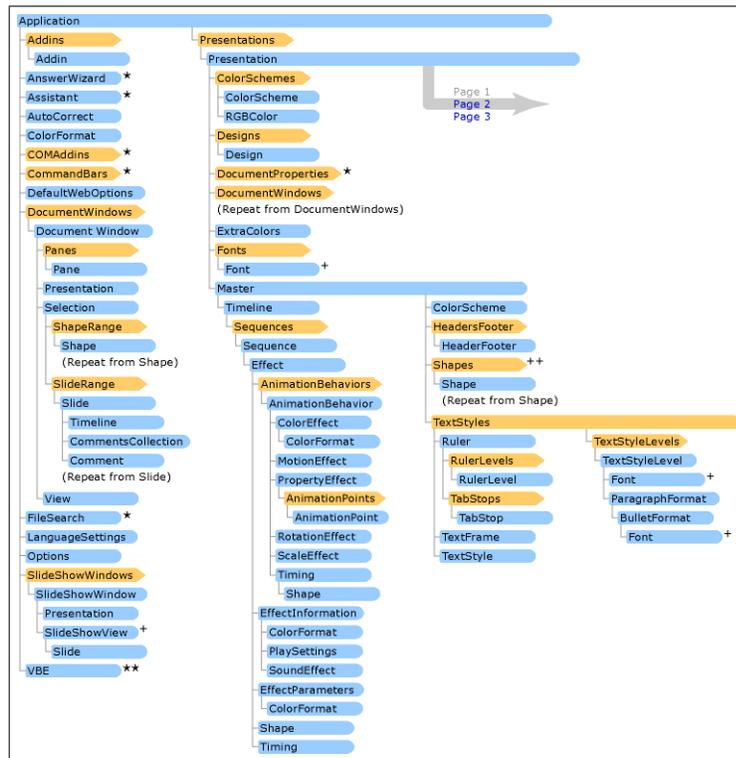


Figure 32: Part of PowerPoint's object model

²¹⁹The given code snippets in this section do not take error conditions into account. The following code assumes that a single slide has been selected.

²²⁰By comparison, the domain model of a compiler is the API to its abstract syntax tree.

²²¹<http://msdn.microsoft.com/library/en-us/modcore/html/deovrMicrosoftPowerPoint2000.asp?frame=true>

Each object model provides a top-level `Application` object that represents the corresponding Office application (e.g., `Excel.Application`).²²² An application object is the “entry point” to the application’s object model. The application objects also provide access to functionality that is common across all Office applications (so-called shared Office components). For example, an application’s menu bars and toolbars are accessible with `Application.CommandBars`, the Office assistant with `Application.Assistant`, and the file dialog with `Application.FileSearch`. The different object models share also common concepts. For example, range objects represent a set of common object that can be manipulated as whole (e.g., to give a common styling). Excel as has a `Range` object that can be populated with cells. PowerPoint has `SlideRange` and `ShapeRange` objects that hold slides and shapes, respectively. For example, the following code deletes the second and third slides in a presentation:²²³

```
PowerPoint.Application.ActivePresentation _
    .Slides.Range(Array(2,3)).Delete
```

6.2.2 REOffice Case Study

REOffice leverages PowerPoint to render reverse engineering graphs, and Excel to compute and visualize metrics data. All functionality has been implemented with Visual Basic.

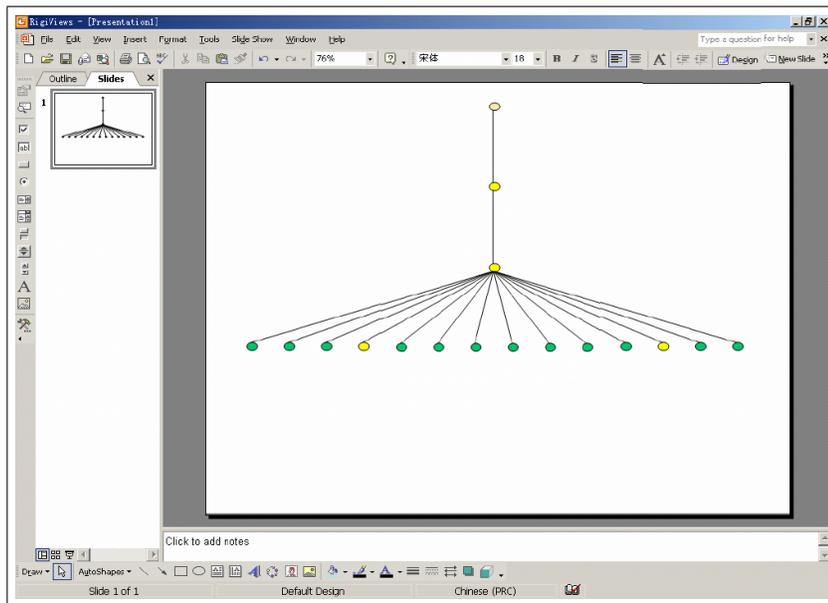


Figure 33: Software structure graph rendered in PowerPoint ([Yan03])

Figure 33 shows a screenshot of PowerPoint that renders a small software graph. The PowerPoint

²²²There are also version-specific application objects; for instance, Excel 2000 is represented with `Excel.Application.9`.

²²³The given code snippet assumes that a second and third slide actually exist in the active presentation. In Visual Basic an underscore (`_`) at the end of a line indicates that the line is continued on the next one.

user can interactively explore and manipulate the graph, which is rendered with shape objects on a PowerPoint slide. REOffice takes advantage of the following baseline functionality of PowerPoint:

slides: Each slide can hold a graph. Thus, several graphs can be kept in a single document. Slides can be dragged and zoomed. User can get an overview of the currently loaded graphs and easily navigate to a certain slide (see “Slides” view on left of Figure 33).

shapes: Circles are used to render the graph’s nodes; connectors between circles realize the graph’s arcs. User can move a nodes around while the arcs stay attached to the node. The shapes’ style options are used to color the graph.

menus: PowerPoint’s pull-down menu and shortcut menu are customized to make domain-specific editor functionality accessible to the user.

persistency: Reverse engineering graphs can be loaded and saved as a standard PowerPoint presentations.

presentation: Users can directly incorporate the graphs into other presentations.

animation: Users can take advantage of PowerPoint’s animation functionality to modify and annotate the graph (e.g., to explain a proposed architectural change of a system).

Similarly, REOffice uses Excel’s baseline functionality for metrics. Especially, raw metrics data is visualized numerically in the spreadsheet, and graphically via bar charts. The screenshot on the left of Figure 34 shows an RSF file that has been imported into Excel (`rsf_stats.xls`). RSF files can be directly imported into Excel because individual records are line-based and fields in a record are separated with tabulators. From the RSF file simple metrics data is computed that shows the number of occurrences of the different node types contained in an RSF file. The statistical data is presented in a separate spreadsheet (`rsf`). The user can compute the metrics data with the “Statistic” menu entry of the “Rigi” menu bar. The user can compute additional metrics either via cell arithmetic or Visual Basic scripting.

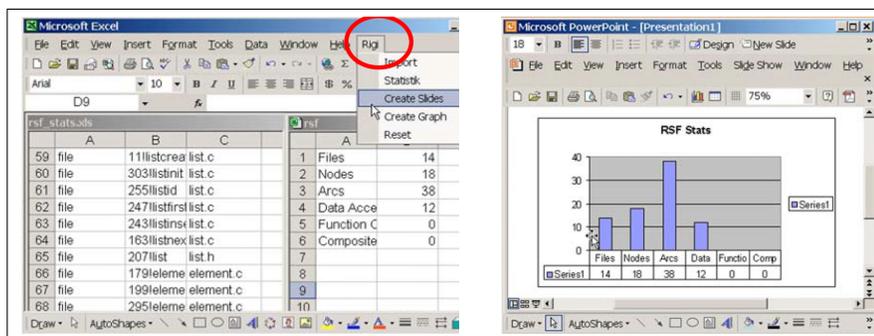


Figure 34: RSF statistics computed in Excel and exported to PowerPoint ([WYM01])

The metrics functionality in Excel also explores the possibility to interoperate with other Office tools via automation [Tur00]. For example, Visual Basic code in Excel can obtain PowerPoint’s application object and use it to create a new presentation and slide: Excel

```
Dim ppApp As PowerPoint.Application
Set pptApp = New PowerPoint.Application
Set newPres = pptApp.Presentations.Add
Set newSlide = newPres.Slides.Add(1, ppLayoutBlank)
```

Interoperation among Office tools (and other COM applications) can leverage each particular tool’s strengths. A full-fledged reverse engineering environment could be constructed out of Word (for text editing and multi-media content), Excel (metrics and table-based views), PowerPoint/Visio (diagrams and graphs), Outlook (collaboration), and Project (management). REOffice uses automation in Excel to create a new PowerPoint slide that shows the metrics data of an RSF file with a bar chart (see right screenshot of Figure 34). The user activates this functionality with the “Create Slides” menu entry (see left screenshot of Figure 34).

On the one hand, REOffice implements a subset of Rigi’s functionality. For instance, graphs in REOffice can be manipulated, but there are no sophisticated layout algorithms. On the other hand, REOffice provides features that go beyond Rigi’s functionality. For instance, metrics data can be easily manipulated and visualized in Excel.²²⁴ To leverage the individual strengths of Rigi and REOffice it is desirable to provide interoperability between both tools. Since both tools leverage different technologies, a lightweight integration approach was chosen: data interoperability based on files (cf. Section 4.2.4). In this approach, text files are used to transfer data between applications. Rigi is acting as a server, polling a certain file for modification. REOffice is writing to the file.²²⁵ Instead of defining a custom format for the file, Tcl is used. REOffice writes a Tcl script to the file that is then read by Rigi and directly executed. This approach eliminates the problem of *invalid data conflict* [HGK⁺06]. Rigi

6.2.3 Conclusions

The REOffice case study shows that it is indeed feasible to leverage OTS products to provide reverse engineering functionality. The implementation of REOffice allowed me to explore the available Microsoft technologies and to gain first-hand experiences with them. Specifically, the case study has explored programmatic customization of OTS products (via Visual Basic scripting), tight integration among homogeneous Microsoft products (via COM-based automation), and loose integration among heterogeneous products (via file-based data integration).

²²⁴To process statistics in Rigi, Tilley has integrated Tcl/Tk spreadsheet (Oleo) and bar chart (BLT) packages into Rigi [Til95, p. 61]. However, this functionality was not included into the official Rigi distributions.

²²⁵Using file modification or creation to signal an event is a standard approach to integrate software components [Del97].

6.3 SVG Graph Editor

Reverse engineering of legacy software systems is a research area that draws on visualization techniques to facilitate program understanding for software maintenance activities (e.g., with call graphs), and to communicate software architecture and design (e.g., with suitable diagrams). These visualizations become an important part of the system and maintenance documentation. In a sense, software structure graphs, such as call graphs, are aerial maps for software engineers. The results of this case study suggest that Scalable Vector Graphics (SVG) is well suited for reverse engineering visualizations. This case study presents our approach to leverage SVG for reverse engineering and system documentation activities.

Typically, graphs of software systems contain different kinds of nodes and arcs. For example, the same graph might show procedures and global variables contained in a program. Different kinds of nodes are visualized with different colors and/or shapes. Since legacy systems tend to be huge (several million lines of code), the graphs can contain thousands of nodes and arcs with different kinds of node and arc types.

Rigi is an example of a reverse engineering tool for visualizing software structures. The graph visualization and manipulation are implemented with Tcl/Tk. This implementation exhibits several drawbacks. The rendering speed is rather slow and does not scale up well for larger graphs. Furthermore, the GUI's look and feel is perceived as rather crude compared to the state of the art. Lastly, images cannot be exported (except for taking screenshots). The last point is a severe drawback, because Rigi graphs are the main result of the reverse engineering activity, which become essential system documentation [WTMS95]. Even if Rigi screenshots are clumsily integrated into the system documentation, they are mere static bitmaps that allow no interactive exploration.

6.3.1 SVG for Reverse Engineering

Intuitively, SVG seems a good candidate for reverse engineering visualizations that promises to alleviate Rigi's original shortcomings. However, a more formal approach to validate this hypothesis is needed. This section discusses in more detail how SVG meets the requirements of reverse engineering tools.

In his dissertation, Wong has identified a set of reverse engineering tool requirements [Won99]. Table 10 lists the relevant requirements in the context of this section. Rigi falls short in several of these requirements. Leveraging SVG within Rigi helps to alleviate several of Rigi's shortcomings. The following list discusses the features of SVG that facilitate reverse engineering, and identifies the addressed requirements (R1-R7):

W3C recommendation (R1): A mature, open standard such as SVG and XML is the foundation for tool support from several vendors in various products, open-source implementations, and cross-platform availability. Adoption of a tool is made easier if it conforms to accepted standards [SK01]. The widespread adoption of GXL in the reverse engineering community can be partially attributed to the fact that it is based

Requirement	Description
R1	Address the practical issues underlying reverse engineering tool adoption
R2	Provide interactive, consistent, and integrated views, with the user in control
R3	Provide consistent, continually updated, hypermedia software documentation
R4	Integrate graphical and textual software views, where effective and appropriate
R5	Use a simple, human-readable, lightweight format
R6	Support introspection of schemas
R7	Support multiple, composable, modular, dynamically extensible schemas

Table 10: Wong's reverse engineering requirements (R1-R7)

on the XML standard [HWS00].

data-centric (R2): The application-centered paradigm views documents as passive entities coded in an internal, proprietary data format. These documents can only be manipulated by a certain set of proprietary tools. In the data-centric model, the document's data format is open and can be manipulated by a heterogeneous collection of tools (which need not be predetermined) [BCV00].

Since SVG is XML-based, SVG can be easily integrated and hyperlinked with other XML-based documents (e.g., XHTML). Thus, integrated views of a software system can be constructed. For example, the Javadoc HTML documentation of a Java system could be enhanced with SVG graphics that visualize the relevant part of the class hierarchy or UML design information. The SVG document could contain hyperlinks to the Javadoc information and vice versa and the user could interactively explore the integrated information in a Web browser.

embeddable (R2) The same SVG document can be visualized in diverse host environments. For example, the Adobe SVG plug-in supports Web browsers (Internet Explorer and Mozilla) and Microsoft PowerPoint. SVG allows reverse engineering results, such as Rigi graphs, to be accessible over the Web, to be used in PowerPoint presentations for project meetings, and to be incorporated into high-quality printed documentation. Most importantly, even though the SVG document is integrated into different host environments, the behavior of the document and the document source itself is the same.

interactive (R2/R3): SVG documents can be made interactive with embedded scripting (e.g., ECMAScript). Thus, documents are no longer necessarily static. Instead, they become *live documents* [WKM02]. For example, Rigi graph manipulations (such as

filtering and layouts) can be performed interactively (e.g., during a PowerPoint presentation) to facilitate comprehension of complex reverse engineering information. In the Javadoc scenario mentioned above, an interactive SVG document could display a class hierarchy that allows the user to expand or collapse parts of the hierarchy. Live documents can be aware of their environment and modify themselves accordingly. In the Javadoc scenario mentioned above, if a Java class is added to a software system, the SVG document can “sense” the change and dynamically adapt its class hierarchy to reflect the current state of the system.

multimedia integration (R4): SVG is among the media formats that the SMIL 2.0 multimedia standard integrates [Rut01]. Interactive visualization of reverse engineering information with open, XML-based standards is an interesting new research area [NEFZ00].

XML-based (R5/R6/R7): Since SVG’s encoding is XML-based, we can leverage XML technology for storage, parsing, and document transformation and querying. Examples of different storage techniques are native XML databases (e.g., Software AG’s Tamino²²⁶), XML-extensions for relational databases (e.g., IBM’s DB2 XML Extender), and hybrid repositories (e.g., ToX²²⁷). XML transformations (e.g., XSLT and Xduce²²⁸) can be used to generate SVG from other XML documents.

The Graph Exchange Language (GXL) is an example of a popular XML-based exchange format in the reverse engineering community. Since both GXL and SVG are XML-based, GXL files can be easily transformed to SVG for visualization.

Another important reverse engineering tool requirement is reuse of existing tools and components. It is beneficial to develop reverse engineering tools that leverage existing (OTS) components and technologies. The benefits are twofold: on the developer’s side we have an increased level of reuse; on the user’s side, we offer a familiar environment to the user.

Component reuse

Reverse engineering tools tend to be developed without much reuse. The Rigi tool achieves a modest level of reuse by utilizing Tcl in its scripting layer and Tk for its GUI implementation. Primitive Tk and X11 drawing routines are used to render the graph. Standard functionality such as loading/saving, zooming, and scrolling have to be custom-implemented. This resulted in a rather crude and idiosyncratic look-and-feel of the GUI that can be frustrating to use.

With SVG we can achieve a higher level of reuse. SVG offers sophisticated, high-level graphic primitives. Interactive behavior of graphical objects can be easily achieved with scripting based on event notification. This allows the implementation of graph editors and visualizers without having to reinvent the wheel.

²²⁶<http://www.softwareag.com/tamino>

²²⁷<http://www.cs.toronto.edu/tox/>

²²⁸<http://xduce.sourceforge.net/>

SVG comes with high-quality visualization engines such as the Adobe SVG viewer and Batik. These viewers already provide functionality for scrolling, zooming, loading/saving, and searching. In a sense, SVG viewers are reusable components based on XML technology.

6.3.2 SVG Integration in Rigi

Because of SVG's benefits outlined above, we decided to leverage it to overcome Rigi's current limitation in visualization and document generation. Figure 31 is a typical reverse engineering result in Rigi. This graph is a document that we would like to easily incorporate into other host environments for presentation and documentation purposes (e.g., Internet Explorer, PowerPoint, Word, and Visio). Previously, this could be accomplished only by taking a static screenshot and importing of the resulting bitmap into the target document.

In our SVG-enabled implementation, the user can export a Rigi graph as an SVG document. The user can either write the SVG document into a file or automatically launch a Web browser to display the corresponding graph. Figure 35 shows the Rigi environment on the left side (workbench and sample graph) and the exported SVG document in PowerPoint (top right) and Internet Explorer (bottom right).

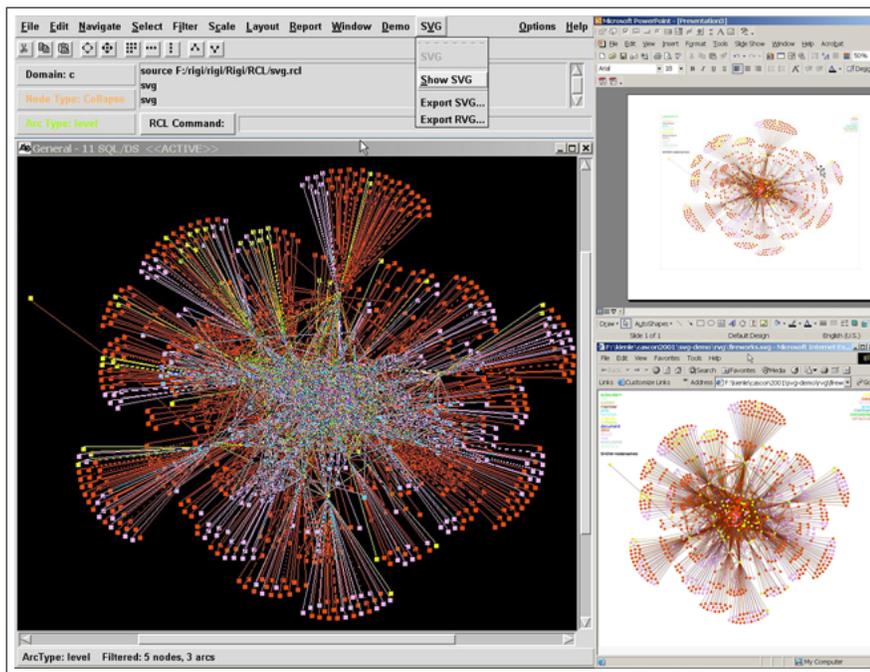


Figure 35: SVG graph in PowerPoint (top right) and Internet Explorer (bottom right)

Export of the SVG graph is accomplished with Rigi's built-in scripting capabilities. Rigi can be programmed with the Tcl scripting language, which provides an extensible core language and was designed to be embedded into interactive windowing applications. Via

Tcl, Rigi's internal graph data structure can be conveniently accessed and manipulated. We wrote a Tcl script with about 200 lines of code that adds an additional pull-down menu to Rigi's GUI (see Rigi workbench on top of Figure 35), and extracts the relevant information from the internal graph data structure and outputs it into an intermediary Rigi View Graph (RVG) file. The information in the RVG file is then transformed to an SVG document. This is accomplished with a Perl script (`rvg2svg`) of about 650 lines of code.

This two-stage design with RVG as the mediator decouples output generation from Rigi (see Figure 36). On the one hand, different reverse engineering tools can export their visualization data as an RVG file and invoke the stand-alone Perl script to generate SVG. On the other hand, different visualizations (e.g., `dot` graphs) can be generated from the same RVG file. We chose Perl to generate SVG from RVG because it has convenient string manipulation facilities and is suitable for rapid prototyping.

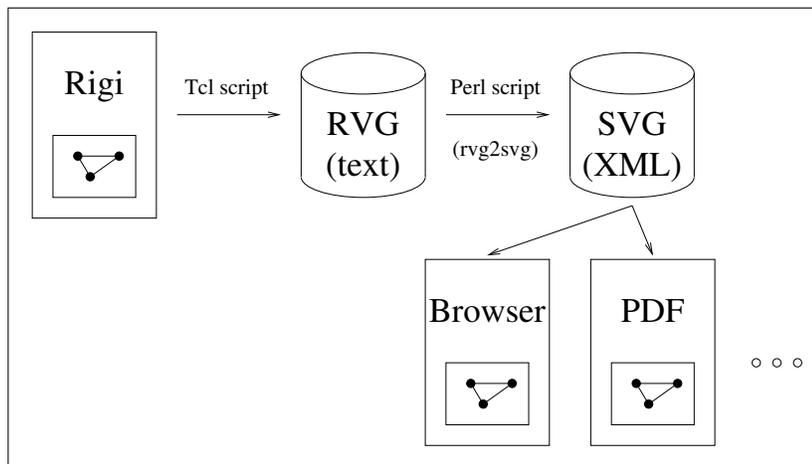


Figure 36: SVG document export in Rigi

The translation of the graphical elements from a Rigi graph to an SVG document is straightforward. Rigi nodes are translated to SVG `circle` elements, arcs are translated to `line` elements. The graph's legend (node types on the left and arc types on the right side) are `text` elements. The following code shows an SVG snippet that defines an arc along with the arc's source and destination node:

```

<g id="arcs" style="stroke-width:1">
  <line id="1052134"
    x1="5304" y1="5133" x2="3123" y2="5033"
    rigitype="0" src="1048650" dst="1048597"
    style="stroke:rgb(255,255,0);"/>
  ...
</g>
...
<g id="nodes"

```

```

onmouseover="DoOnMouseOverNode(evt)"
onmouseout="DoOnMouseOutNode(evt)"
font-size="135"
style="stroke:black;stroke-width:1;opacity:1.0;">
...
<circle id="1048650" rigitype="3"
  rigiarcs="1052133,1052134,1052115,1052510"
  cx="5304" cy="5133" r="40"
  style="fill:rgb(255,255,0);"/>
...
<circle id="1048597" rigitype="3"
  rigiarcs="1052231,1052116,1052134,1052879,1054935"
  cx="3123" cy="5033" r="40"
  style="fill:rgb(255,255,0);"/>
...
</g>

```

Every element has a unique `id` and contains several *non-standard attributes*. These are used by the embedded ECMAScript code. For example, the `line` elements contain the non-standard attributes `src` and `dst` that identify the source and destination nodes of the arc, respectively.

The `rvvg2svg` script uses Ronan Oger's SVG-2.0 module.²²⁹ The module provides an API to create SVG elements along with their attributes, to nest them properly, and to finally output indented XML code. The following Perl snippet shows how graph nodes are translated to SVG `circle` elements:

```

# include Oger's SVG module
use SVG;

# instantiate new SVG document
my $root = SVG->new( -indent => ' ',
  ...
  onload => "DoOnLoad(evt)" );
...

# create a SVG group element that contains the graph nodes
my $nodes_group =
  $root->group ( id => "nodes",
    onmouseover => "DoOnMouseOverNode(evt)",
    onmouseout => "DoOnMouseOutNode(evt)",
    "font-size" => $font_size,
    style => "stroke:black;stroke-width:1;
      opacity:1.0;" );

```

²²⁹<http://search.cpan.org/search?dist=SVG>

```

# iterate over all nodes in the Rigi graph and
# generate SVG <circle> elements for each node
my $node;
foreach $node ( keys %{$rvrg->{nodes}} ) {
    my $x = $rvrg->{nodes}{$node}{x};
    my $y = $rvrg->{nodes}{$node}{y};
    ...

    # create SVG <circle> element
    my $circle =
        $nodes_group->circle( id => $node,
                               cx => $x,
                               cy => $y,
                               ... );
}

...

# output XML
print $svg->xmlify( "-in-line" );

```

Besides Oger's SVG-2.0, there are several other Perl modules available for SVG document generation.²³⁰

We use embedded ECMAScript to provide interactive behavior of SVG documents for the reverse engineer. Figure 37 shows an SVG document rendered with a Web browser; it can be interactively explored using the features described below:

- Nodes and arcs can be filtered by clicking on the legend text. For example, to filter out red “data” and purple “struct” nodes in the SVG graph, the corresponding labels (located in the upper left corner) can be clicked. This results in the graph depicted in Figure 37.
- If the user moves the mouse over a node, its name is displayed next to the node.
- If the node has associated source code (e.g., the PL/AS code for a subsystem in IBM's SQL/DS system), clicking on the node will display the code in a Web browser. If no source code is available, a default page is displayed in the Web browser.

This functionality is implemented with about 270 lines of ECMAScript code.

6.3.3 SVG Experiences

This section summarizes some general observation that we made during our SVG development. We also compare the features of SVG graphs with Rigi graphs.

²³⁰<http://search.cpan.org/search?mode=module&query=SVG>

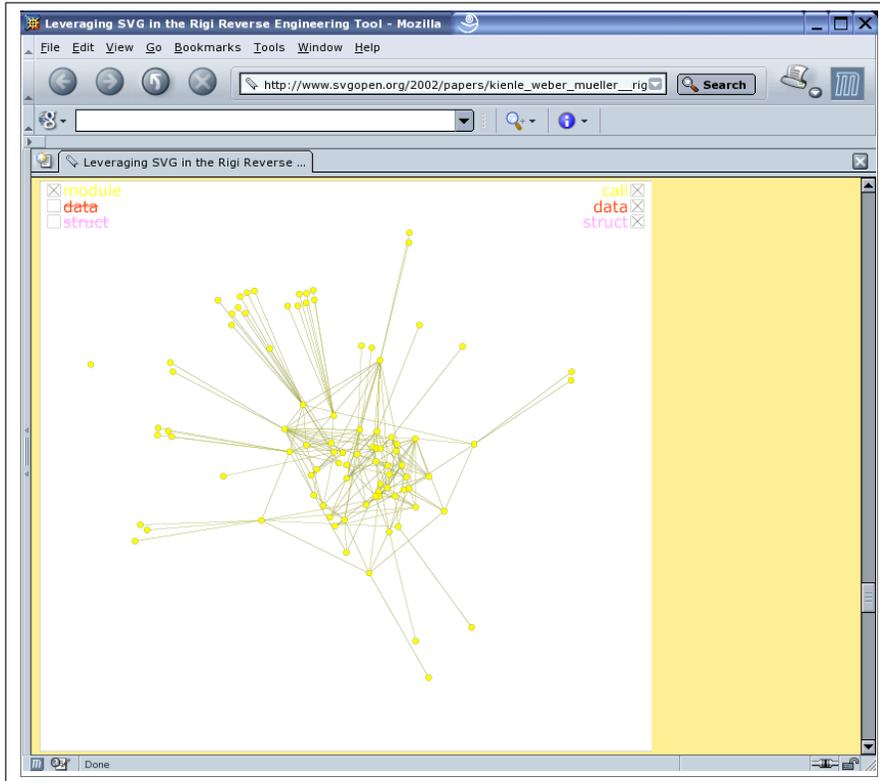


Figure 37: SVG graph embedded in Web browser

- The size of the SVG document for the graph in Figure 37 is about 932 KB. (We made no attempts to minimize the size.) In contrast, a high-resolution screen shot of this image takes more than 9 MB.
- Loading and rendering of SVG documents in Internet Explorer and PowerPoint scales up well compared to Rigi. However, the browser freezes up for several seconds while loading the SVG document in Figure 37.
- Zooming with the SVG document is fast and causes no loss of rendering quality.
- SVG file compression with `gzip` yields good results for our documents. For example, the size of the compressed SVG document for the graph in Figure 37 is about 130 KB.
- As mentioned before, Rigi's rendering of graphs is not visually appealing. SVG on the other hand has powerful and versatile rendering features that allow sophisticated visual effects (e.g. nodes with opaque colors and arc arrows with `marker` elements).
- The ECMAScript implementation of complex graph manipulations (such as filtering of nodes and arcs) is slower compared to Rigi, but sufficiently fast. Also, we made

no attempt yet to optimize the ECMAScript code, whereas Rigi's code is highly optimized.

- Displaying the node names when the user moves the mouse over a corresponding node happens instantaneously. This suggests that callbacks that manipulate the DOM of the SVG document are executed efficiently.

6.3.4 Conclusions and Future Work

The current approach uses a converter that directly generates SVG text and basic shape elements to render the graph. However, it is also possible to dynamically generate these SVG elements with ECMAScript (right after the document has been loaded). For example, Adobe's Chemical Markup Language Demo uses this approach [Ado02a].

It is an important design choice which SVG elements to generate statically in the XML document and which to instantiate dynamically with ECMAScript. Both design choices have different trade-offs that have to be carefully considered. For example, with our current, static approach, the graph is visible with SVG viewers that do not have scripting support. The information that the non-interactive graph provides is still useful for the reverse engineer. With the dynamic approach, the user would see an empty (or partially drawn) document that is of no use. To investigate the involved trade-offs in more detail, we plan another converter that uses the dynamic approach.²³¹

We also plan to further enhance the SVG exporter. Especially, we want to make the SVG documents more interactive by incorporating additional graph manipulation functionality (e.g., moving of nodes).

For historical reasons, the RVG file format (see Section 6.3.2) is text based. To better leverage XML-based technology, we plan to define an XML-based format. This would allow us, for example, to use XSLT to transform Rigi graphs to SVG.

My experiences with SVG have been largely positive. SVG's features allowed us to generate useful graphs for the reverse engineering domain that are visually appealing. These graphs are interactive and embeddable into host applications such as Web browsers and Microsoft Office. Interactive SVG graphs are a first step towards live documents. Embeddable SVG graphs are important for document generation and presentation purposes.

6.4 REVisio

As software systems grow older, software engineers increasingly find themselves faced with maintaining, extending or reengineering largely undocumented legacy applications. Before any of these tasks can be attempted, an understanding of the software at a higher level of abstraction must be gained. Reverse engineering tools can help with this challenge, yet sport a surprisingly low adoption (and even evaluation) rate in the software industry. This low penetration can be partially attributed to a lack of polish in user interfaces and

²³¹This approach has been investigated in joint work with Jon Pipitone and presented at the *CASCON 2002 SVG Workshop* [PK02].

weak support for integration with other tools, comparing poorly to mainstream packages such as the Microsoft Office suite.

One way to overcome these impediments is by grafting reverse engineering functionality onto entrenched tools. These base tools could be integrated software development environments (IDEs), but in general need not be related to programming, and include such applications as editors, shells, browsers, word processors, and personal information managers [Wal03a]. Some of these tools offer built-in extension facilities or development kits, making it easier to repurpose them.

One such flexible tool is Microsoft Visio. It is a member of Microsoft's Office suite so users can embed Visio drawings into other documents, customize its user interface in standard ways, and benefit from a widespread support base for that application. Visio is particularly well suited to our endeavor since it is built around a powerful diagramming engine to which it provides full programmatic access.

With REVisio we illustrate how domain-independent, native Visio operations (such as panning, zooming, and automatic layouts), integrate seamlessly with end-user programmed, domain-specific operations (such as filtering or metrics), to provide reverse engineering functionality.

6.4.1 Microsoft Visio

Microsoft Visio, a member of the Microsoft Office suite, is an advanced drawing tool for all kinds of diagrams: flowcharts, block diagrams, building plans, maps, et cetera. While users can create diagram from scratch, Visio also provides a number of predefined templates that reconfigure the application for specific diagram types. Templates usually include stencils (though the latter can also be loaded independently), which are coordinated collections of master shapes. Master shapes are presented in a palette to be dragged by the user onto the canvas automatically instantiating the corresponding shape, allowing the users to focus on the content of the diagram rather than their drawing skills (or lack thereof). Visio also provides a variety of other shortcuts, wizards and navigation tools that further ease the user's cognitive workload.

Visio allows users (and developers) to customize its operation on many levels [Cor01].²³² At the most basic level, users can employ their own masters, stencils and templates to customize the work environment. Fundamental drawing and transformation tools can be used to create or modify shapes, controlling their geometry, color and style. Any shape can then be promoted to a master and included in a stencil or template.

Customization
options

For more advanced customization, Visio exposes the masters' ShapeSheets. A ShapeSheet is a special-purpose spreadsheet whose cells are connected to the shape's properties, giving full control over the shape's geometry and behavioral constraints. ShapeSheets support formulas allowing complex relationships to be formed between cells, and provide limited access to certain operations defined on the Visio object model. How-

²³²<http://www.microsoft.com/office/visio/faq.asp>

ever, each ShapeSheet is attached to a single shape, so it is impossible to write formulas that take other shapes on the canvas into account.

Finally, Visio is an automation server, exposing its object model for other software to use. The object model encompasses almost all data in Visio, including canvas and shape information, ShapeSheets, menus and dialogs, giving client applications full control over all aspects of Visio. They are able to modify existing elements in any way, as well as add new ones, or simply access information about the canvas' current contents. One popular use of automation is to construct a diagram according to some higher-level information provided by the user.

Automation clients can be written in any language that supports automation (or COM), including Visual Basic for Applications (VBA), Visual Basic, C, and C++. Visio includes a VBA development environment and VBA macros can be embedded directly into Visio documents, thus simplifying both development and distribution.

A Visio solution brings together all the customized items that may be needed in the pursuit of a specific diagramming goal. Solutions combine templates, stencils, masters, customized ShapeSheets and various scripts and macros that use the automation facilities [Eds00]. A typical solution maintains the original Visio paradigm of drag-and-drop diagram assembly from a palette of master shapes, while augmenting the shapes with "smarts" appropriate to the specific application. Users can indirectly affect the solution's shapes' properties through menus or data entry to automate the tedious or complex manual manipulation that would normally be required.

6.4.2 REVisio Case Study

REVisio is a partial adaptation of Rigi (augmented with metrics) as a Visio application. Rigi is a mature and feature-rich tool, yet its rate of adoption in the industry is lower than might be expected.

Visio, on the other hand, is widely used by software developers for design and analysis tasks. Our hypothesis is that those developers would hence be more likely to evaluate and adopt a Visio reverse engineering application due to its ease of installation and beneficial learning curve. Building on Visio also allowed us to shorten the development time of REVisio by reusing Visio's extensive diagramming facilities.

This section discusses how REVisio acquires information about software systems, visualizes the software's structure, and calculates and displays metrics. All application functions are implemented using a combination of automation, ShapeSheets and customized masters, and can be controlled from a customized tool-bar that integrates smoothly with Visio's user interface (cf. Figure 38).

While REVisio is a Visio solution, it does not follow the typical drag-and-drop process for creating diagrams. Rather, drawings are initially automatically derived from imported data, and can then be navigated and modified by users with Visio's usual assortment of tools. The base data is first extracted from source code with established Rigi parsers and

Data import and
statistics

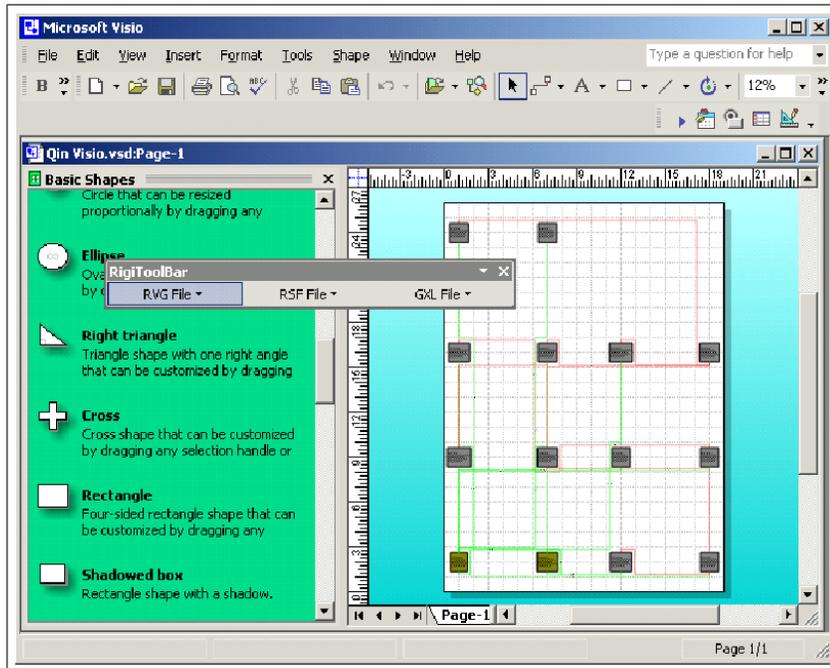


Figure 38: REVisio showing a Rigi graph

saved in Rigi Standard Format (RSF), Rigi View Graph (RVG), or Graph Exchange Language (GXL) files.

When a model is read in, REVisio presents the user with some statistics allowing them to estimate the complexity of the system. To provide an overview of the graph's contents, REVisio displays annotated pie charts that show the total numbers of nodes and arcs of different types as well as their relative proportions (cf. Figure 39). From these charts, the user can quickly intuit the model's size, the number of different types used in the model, and which ones are used most often.

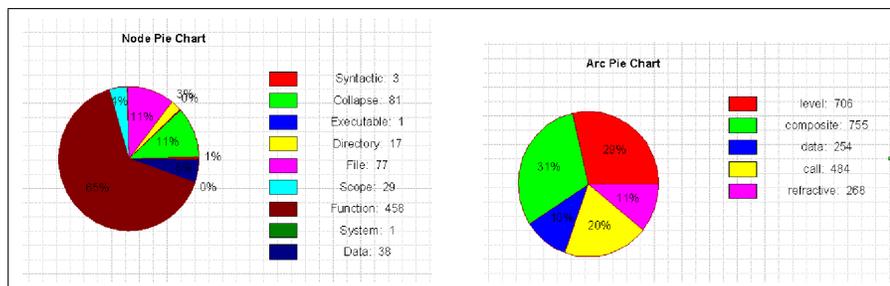


Figure 39: REVisio showing Rigi pie charts

Beyond these simple statistics, REVisio can display the graph read in from an RSF, RVG or GXL file. RSF files hold only structural information extracted from the target Structure
visualization

system, while RVG and GXL files are often augmented with view-specific attributes (e.g., displayed node location and color) that REVisio tries to preserve as much as possible. The nodes and arcs are shown using Visio shapes and dynamic connectors; the arcs leverage Visio’s automatically routed rectilinear paths (cf. Figure 38). The resulting diagram can be extended and manipulated with all the standard Visio tools; two particularly useful ones are pan & zoom and automatic layout.

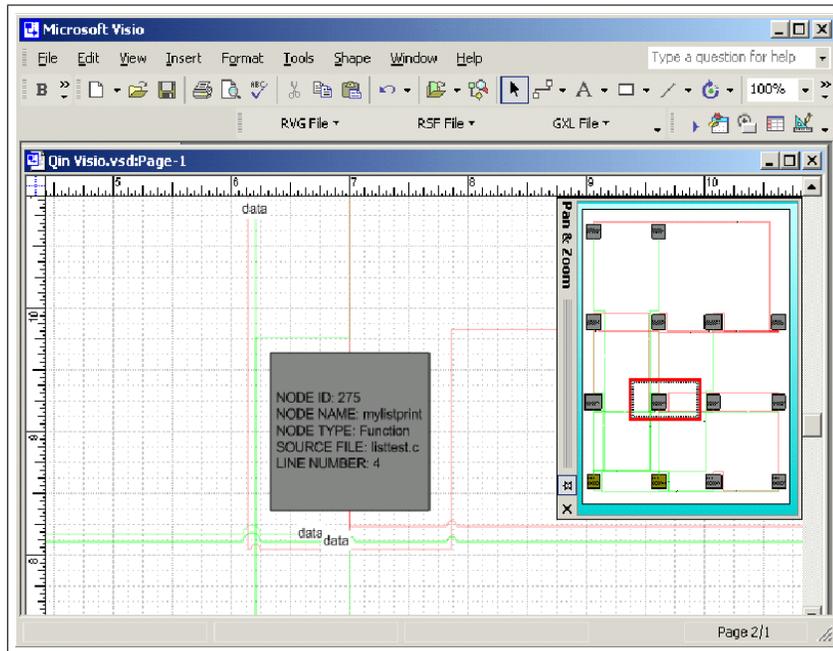


Figure 40: REVisio showing Visio’s pan & zoom tool

Visio’s pan & zoom feature, shown in use in Figure 40, provides the user with a context view that lets them focus on diagram details while maintaining an overview of the structure. The context view also gives the user direct control over the focus of the zoomed-in view, allowing them to navigate large diagrams with ease.

Visio can also automatically lay out graphs according to a number of common algorithms, improving the organization of complex diagrams. For example, Figure 41 shows the previous diagram laid out in a tree style using straight connectors, with the page automatically resized to hold the resulting drawing. Larger graphs can be automatically laid out as well (cf. Figure 42), but the operation is not always efficient and its running time may quickly become prohibitive. (The graph in Figure 42 took about 10 minutes to lay out on a reasonably equipped machine, but we have not yet made detailed performance measurements or attempts to optimize the operations.)

Inheriting these familiar diagram tools from Visio simultaneously makes REVisio easier to learn and use and shortens its development time. To support reverse engineering, REVisio introduces a few custom operations accessible through a toolbar menu. We allow

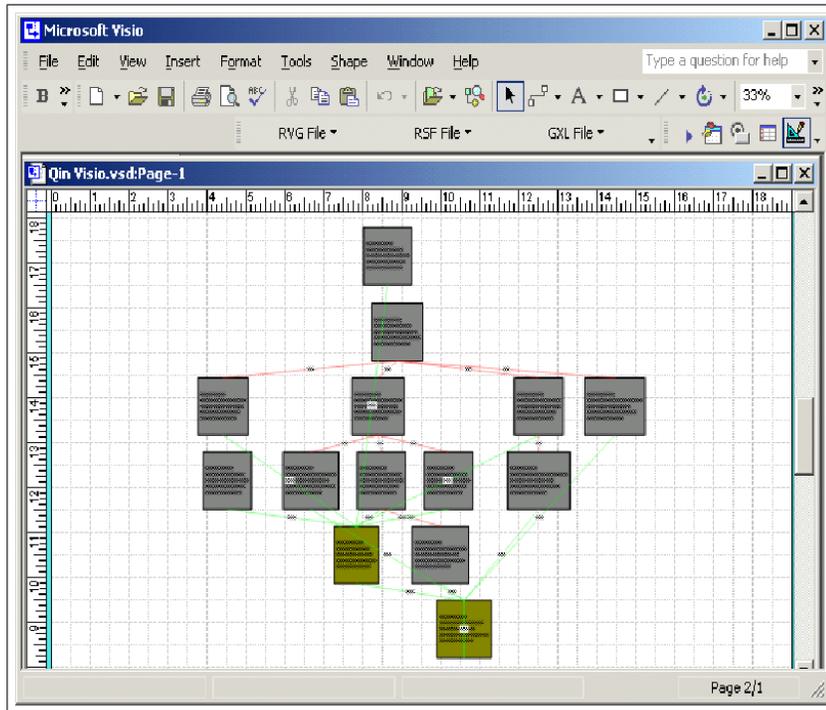


Figure 41: REVisio tree layout of a small graph

the user to select nodes by type and to selectively hide them to reduce clutter. We also provide basic graph traversal operations, highlighting all parents or children of a given set of nodes based on a given relationship type.

These rudimentary functions are obviously inadequate for all but the simplest reverse engineering tasks. They serve merely as a proof of concept, to demonstrate that it is easy to script automated operations on the graph. We expect users to be familiar with Visual Basic and to write their own ad hoc scripts as the need arises.

REVisio counterbalances the often overwhelming detail provided by software structure graphs with a variety of software metrics. We focus on classic metrics that produce numbers characterizing properties of software code [FN00]. A good metric can aid a reverse engineer to better assess certain characteristics of the subject software system and decide where to focus their attention. For example, metrics can indicate which parts of a software system have a high complexity or tight coupling.

Metrics visualization

We have initially implemented four metrics in REVisio:

- LOC (Lines of Code) counts the lines of code for each method or class.
- NMC (Number of Methods per Class) counts the number of methods for each class, thus indicating a class' size.
- CBO (Coupling Between Objects) counts the number of foreign classes referenced,

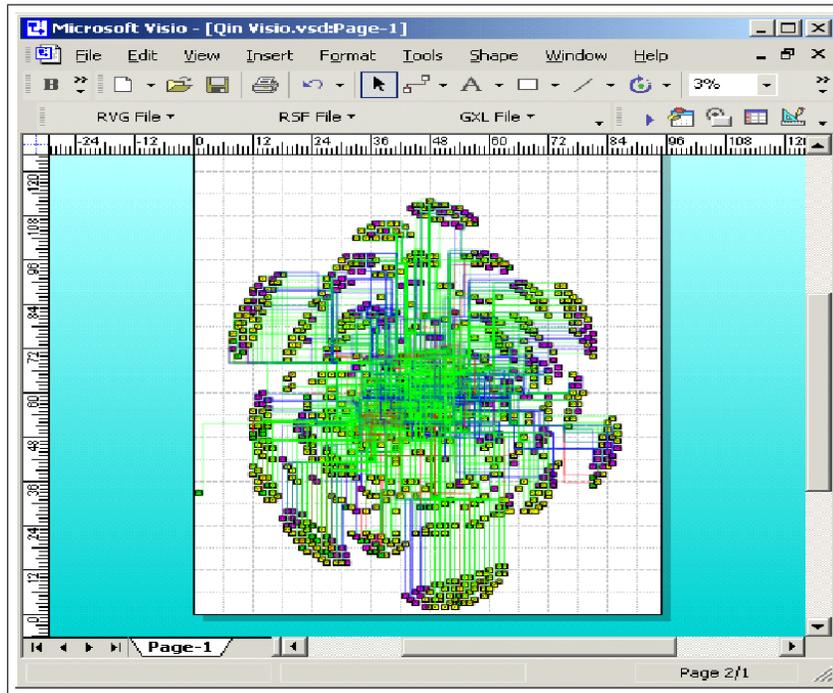


Figure 42: REVisio radial layout of a large graph

either through field access or method calls. A large CBO value indicates that a class is highly dependent on other classes [CK91].

- RFC (Response For a Class) extends the CBO by counting the number of foreign methods invoked by a class (and its value may therefore exceed the CBO value) [CK91].

REVisio computes and displays the metrics based on the structural information read in from an RSF file. Figure 43 shows a sample CBO bar chart, with the class names on the X axis, and the CBO values on the Y axis. The chart was quickly implemented by scripting existing Visio masters and takes advantage of Visio's pan & zoom feature to provide an overview and quick navigation. Other metrics, even ad hoc ones customized to the user's target system or tasks, should be equally easy to implement.

6.4.3 Related Work

Our proposed approach is to craft software engineering functionality on top of OTS products that offer end-user programmability through either a scripting language or a more general automation API. In related projects in our group we extend Lotus Notes (cf. Section 6.5) and PowerPoint (cf. Section 6.2) to visualize and manipulate Rigi graphs. Other research has leveraged OTS products as well to build software engineering tools.

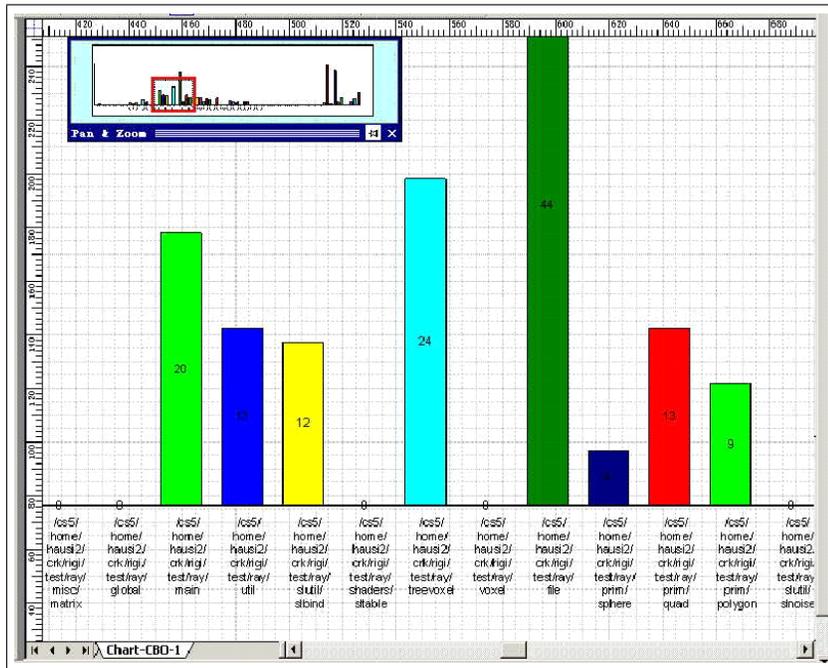


Figure 43: REVisio CBO bar chart

Riva and Yang have developed a software documentation process that uses Rigi to visualize software artifacts [RY02]. They used Rigi’s scripting capabilities to export this information to Visio as a UML model. The exporter writes files in Visio’s XML vocabulary employing predefined UML master shapes. The authors take advantage of Visio’s ability to produce HTML renderings to web-enable the documentation. It is interesting to note that their approach uses Rigi’s scripting to write Visio XML files whereas our approach uses Visio’s scripting to read in Rigi files.

Tilley and Huang report on their experiences with an industrial client in implementing a software visualization and documentation system in Visio [TH02]. Visio was selected after evaluating the visualization capabilities of several candidate tools. The authors were constrained in their technology choices by the client’s policies. For example, “the company reasonably requested that professional support be available for whichever tools were selected. This requirement immediately ruled out almost all academic and research tools.” Among the identified benefits of Visio was that the client already employed Visio in their development process and had a set of custom-developed stencils to represent their software artifacts. Similarly to our approach, the authors used Visio to visualize software artifacts in a graph.

Systä et al. have extended Rigi with support to calculate and visualize various object-oriented metrics (among them RFC and CBO) [SYM00]. Metrics data can be exported to Microsoft Excel spreadsheets. The spreadsheet is then used to visualize the metrics data

with line diagrams similar to our approach. They also use an Excel macro to generate a correlation matrix for the metrics.

6.4.4 Conclusion

By building REVisio on top of Microsoft Visio, we not only saved development effort but also lowered the barrier to adoption that reverse engineering tools face. Users can leverage previously acquired cognitive support and take advantage of widespread training opportunities, flattening the learning curve. Integration with other office tools is enhanced, resulting in a uniform user interface and the ability to easily embed diagrams into other documents. Finally, Visio (and thus REVisio) supports scripting in the popular Visual Basic language. Ad hoc scripting is critical to reverse engineering activities, and VBA may prove easier to adopt than Rigi's Tcl.

Naturally, there are some tradeoffs to our approach. Visio's diagramming engine was originally meant for interactive use, so some aspects of it can be difficult to automate. For example, some masters automatically pop up dialog boxes or are limited to some small number of nested elements. These limitations can for the most part be circumvented with careful programming and shape customization. Performance, however, may prove to be a serious stumbling block. While VBA is usually fast enough for ad hoc scripting tasks, Visio itself was clearly not designed to scale gracefully to the thousands of shapes required to display models of even moderately large systems. Low performance is a factor in tool adoption and may negate any gains made by basing REVisio on a popular tool.

Further work is necessary to validate our claims of lowered adoption barrier, as well as to improve the usability and performance of REVisio. Nonetheless, we believe that REVisio explores an interesting and potentially worthwhile avenue of research, and demonstrates the promise of the ACRE approach.

6.5 RENotes

It takes a lot of effort to go from the conceptual design for a new software engineering technique to the development of tools that support the technique and finally to the adoption of the tool in industry. Researchers who strive to have their tools adopted struggle to develop tools that satisfy the requirements placed on them by software engineers in industry. Common examples of adoption hurdles include difficult installation, lack of documentation, unpolished or awkward (or both) user interfaces, and poor interoperability with existing tool infrastructure. As a result, most research tools require a significant learning curve that disrupt the established work processes of the software engineer.

Tool development is a significant investment and should focus on the novel features of the tool. Unfortunately, the development of a baseline environment, albeit often trivial to accomplish, requires significant effort before tool-specific functionality can be tackled. This is especially true for GUI-based tools that use a visual manipulation paradigm. More than a decade ago, Grudin already advocated to leverage existing baseline environments

to build groupware tools: “If possible, add groupware features to an already successful application rather than launch a new application” [Gru94, p. 100].

In this case study, we outline a software development approach that leverages a widely-used, shrink-wrapped office tool—Lotus Notes—by building a software reengineering tool, RENotes, on top of it. Notes is the *host application* that provides the baseline environment and includes features such as document-based database, collaboration, search, and security. RENotes leverages Notes to provide software reengineering functionality such as artifact filtering and graph manipulation.

We believe that tool implementations that follow this approach have desirable features from the user’s and developer’s points of view. Spinellis draws a similar conclusion for the field of visual programming tools [Spi02]:

“As many visual programming environments are research-oriented, proof-of-concept projects, they cannot easily compete with the commercially-developed, polished, and supported commercial IDEs. In contrast, visual programming based on the reuse of proven existing components in a widely adopted IDE levels the playing field and allows research to focus on program representations and methodologies that can increase productivity rather than supporting infrastructure.”

In the following, we discuss the background information that guides our research, and the customization of Lotus Notes to implement RENotes, a reverse engineering tool that incorporates much of the functionality of Rigi.

6.5.1 Background

This section discusses tool customization and categorizes the host applications and the targeted users to provide a better understanding of the requirements of our research.

A prerequisite for our proposed tool development method is that the host tool offers sophisticated customization mechanisms.²³³ Tool customization

Support for customization can be divided into non-programmatic and programmatic customization mechanisms (cf. Section 4.2.3). Non-programmatic customization is accomplished, for example, by editing parameters in startup and configuration files or with direct manipulation at the GUI level. Programmatic customization involves some form of scripting or programming language that allows the modification and extension of the application’s behavior. Programmatic customization is more powerful, but requires a significant effort. There is an initial cost in acquiring the necessary customization skills, followed by development and maintenance costs of the customization.

The extent to which users customize their applications is an interesting question. Page et al. studied the customization changes that users made to the WordPerfect word processor [PJAA96]. A surprising 92 percent of users in their study did some form of customization, with 63 percent using macros. Page et al. summarize their findings with “users who most

²³³Such tools have been also referred to as user-tailorable computer systems [MCLM90].

heavily used their systems have the highest levels of customization” and “customization features that were simple to adapt (like the Button Bar) tended to have higher incidences of tailoring” [PJAA96].

The above study suggests that tool builders should expect that users will want to customize their software; thus, tools should offer extensive customization support while making it simple and fast to use. Indeed, tool builders have recognized the importance of making their software customizable and by now many popular tools (such as Microsoft Office, Lotus Notes, Adobe Acrobat, and Macromedia Dreamweaver) offer scripting support and APIs to accomplish customization.

The feasibility of customizing of Notes has been demonstrated by a number of large-scale projects. The following experiences were reported after one such effort [LT97]:

“Users and developers have been positive about Notes. Advantages cited include the relative ease of learning how to develop basic applications, the relatively short development cycle— from a few days to a couple of weeks per applications—and the fact that users’ suggestions and feedback can be incorporated into the applications with relative ease and can be done on a continuous basis.”

The development approach that we describe grafts domain-specific functionality on top of highly customizable tool foundations—we call these tools host applications. Host applications

There is a broad range of candidates for host applications. In our current research, we focus on office suites (e.g., Microsoft Office, Lotus SmartSuite, and OpenOffice) and productivity tools (e.g., Lotus Notes and Microsoft Project). From the programmer’s point of view, these tools have OTS characteristics.

We classify systems that use OTS components based on the scheme proposed by Carney [Car97]:

turnkey: These systems use a single OTS component on which they heavily depend. Typically, customization is quite limited and non-programmatic.

intermediate: These systems are also built on a single OTS component, but also “have a number of customized elements specific to the given application.” The amount of customization can vary, but does not fundamentally change the nature of the employed OTS component and results in a moderate amount of newly developed customization code.

mixed: These systems contain several (heterogeneous) OTS components to provide large-scale functionality that is otherwise not available. They have a significant amount of glue code and are often difficult to develop and maintain.

In our current research, we target intermediate systems (that is, only a single host application is selected and customization is done programmatically).

Another important consideration is the users that the application targets. Karsten Target users

[Kar99] reviewed 18 case studies of organizations' use of Notes and splits them into three groups: exploratory, expanding, and extensive use of Notes. Organizations with extensive use of Notes had the following commonalities [Kar99]:

“In all these cases, there were several applications that were tied directly into established work practices. Application development was conducted as a careful process, with prototypes and user involvement.”

Thus, in the best case, the host application is a fundamental part of the users' work processes. Local developers (“gardeners”) who work on customizations for their group are evidence of such a mission-critical tool. This has been observed, for example, for CAD systems [GN92]. An other example is the IBM Toronto Lab, which has a dedicated development group to customize Notes.

Host applications that are based on familiar office tools provide a number of potential benefits to users:

a familiar GUI: The user interacts with a familiar environment and paradigm. Application knowledge (a.k.a. cognitive support [Wal03a]) has been typically built up by the user over years. Since the users are already familiar with the standard functionality, they can concentrate on learning the additional, domain-specific functionality.

tool interoperability: Office tools interoperate among each other via cut-and-paste and (file-based) import/export facilities.

tool support: Popular tools come with a large infrastructure that provides useful information to the user. For example, online publications that discuss how to use a tool more effectively. Mailing lists and discussion forums help troubleshoot users' problems.

Stand-alone research tools are typically found lacking in all of the areas outlined above.

Selection of a host application is a trade-off decision. When deciding on a suitable host application, one typically has to choose among several possible candidates. A host application has to satisfy two main criteria: it has to (1) provide a suitable baseline environment for extension, and (2) be familiar to its target users. The former criterion shortens the development life cycle while the latter can accelerate the adoption. Sometimes, these two criteria may conflict with each other. The decision to select Notes was based on its large user base in companies and its customization flexibility. Although we did not formally evaluate the trade-offs, our observations and discussions with developers at the IBM Toronto Lab held up our assumptions. Notes appears to be the most pervasive application within the IBM Toronto Lab. Besides its use as a group and peer-to-peer communication infrastructure, Notes is the host application for tens of customized applications used by the Lab employees. About seven developers in the Lab are currently involved in development and maintenance of these applications.

Selection of host
applications

6.5.2 Rigi

To gain experiences and validate our tool development approach, we decided to build a reverse engineering tool on top of Notes that is similar to Rigi as a case study. Because of our previous tool building experience with Rigi, we are already familiar with the application domain and can focus on understanding customizations with Notes.

Rigi is an interactive, visual tool designed for program understanding and software re-documentation. Even though Rigi can be customized with Tcl scripting, it is a stand-alone application that is not easily integrated with other tools. Most notably, short of taking screenshots, Rigi graphs cannot be exported to other applications. Leveraging office tools to build reverse engineering functionality seems a promising approach because the reverse engineering process for a larger legacy system is both document-intensive and collaborative. An important problem is the task of organizing and archiving of obtained reverse engineering results, so that they are available for future system evolution. As described in the next section, Notes has features that address these problems.

6.5.3 Lotus Notes/Domino

Lotus Notes with Domino²³⁴ is a popular groupware solution that is used by many organizations. It is important to realize that Notes has more to offer than e-mail support. In fact, it is often mentioned as the main software for supporting collaboration and its use is extensively studied by researchers in the area of computer supported collaborative work [Kar99].

Based on the client/server model, Lotus Notes/Domino is often used to construct an internal information platform for an organization [Fal97]. In this platform, all information is represented as documents, which are stored and organized in databases. Domino acts as the server, providing database access; Notes hosts the client applications that access the database. In Notes, shared databases are the backbone that enables collaboration among members in an organization.

For many organizations Notes is a critical part of their IT infrastructure. An example is Statoil, a Norwegian state-owned oil company [MA01]. Statoil has a full company license making it one of the world's largest users of Notes. Five years after its introduction in 1992, Notes had diffused to most of the company's 17000 employees, and its use was made mandatory. Statoil's geographical distribution (40-50 sites) makes collaboration-based features attractive. The following collaborative features of Notes are leveraged: e-mail, document management, workflow, electronic archive, group calendar, news and bulletin boards, and discussion databases.

IBM itself is another example of a company that uses Notes extensively. From our observations at the IBM Toronto Lab, Notes has been customized for a wide variety of application, ranging from a simple carpool database to a project application that allows a development team to organize the work-products of their project. With the Domino Web

²³⁴<http://www.lotus.com>

server, Notes applications can be published on the Web. When an HTTP request accesses a database, Domino translates the document to HTML. Thus, certain applications can be made accessible with both Notes and a standard Web browser. At the Toronto Lab, Domino servers within the intranet allow Web access to applications.

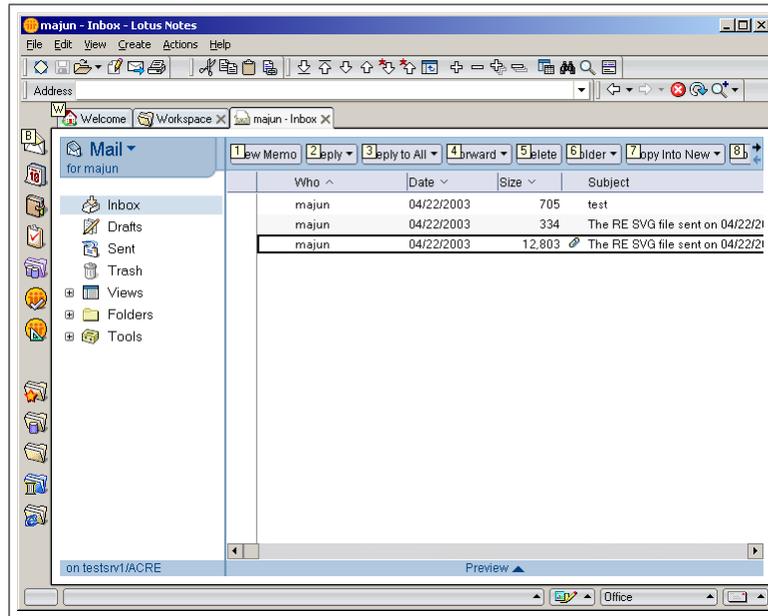


Figure 44: Lotus Notes mail application

Before deciding on a host application, it is important to understand its key features. The host's baseline functionality and customization mechanisms determines its suitability for building domain-specific functionality on top of it. The user leverages the following features of Notes' baseline environment when using RENotes:

Notes features

user interface: Notes has a mature (if idiosyncratic) user interface that has been consecutively refined over six major releases. Figure 44 shows a snapshot of the latest release, Lotus Notes 6. The user interface includes a menu bar, a bookmark bar, a toolbar, a status bar and a set of window tabs. Tabs make it easy for the user to switch to different applications and databases. Most of the GUI elements can be customized. Furthermore, each tab has its own set of task-specific menu items and buttons. The mail tab, for example, has a button to compose a new e-mail (“New Memo”) and a pull down menu with several options on how to reply to the currently selected e-mail (“Reply”).

document-based database (DBD): The documents in each database are organized with views and folders. In Notes, every document is a basic data unit that contains a set of data fields. In this respect, a document is similar to a record in a relational database; however, a document in Notes can be composed of an arbitrary number of

fields. Since all documents adhere to this schema, it is possible to access, manage, and manipulate diverse documents in a uniform manner. Notes databases support many data types, including text, pictures, sound, video, file attachments, embedded objects, and applets.

script automation: Users often take advantage of customization to automate work. A study about the customization behavior of 51 users in a Unix environment found that users “were most likely to customize when they discovered that they were doing something repeatedly and chose to automate the process” [Mac91].

Notes provides users agent and actions to accomplish automation. Actions are integrated into the GUI and activated by users (e.g., by pressing a button). Agents, which run on the server, can be triggered by certain events (e.g., expiration of a timer).

collaboration: Typical Notes applications are message exchange, information sharing, and workflow. By sending and receiving e-mails, members exchange messages in an organization. By accessing documents in shared databases on servers, distributed information exchange, retrieval, storage, and consolidation is facilitated. Workflow applications guide users through certain tasks that they have to perform as part of their work. Such guidance can reduce overhead and mistakes, thus speeding up processes.

search: Notes has automatic search capabilities as well as full-text indexing support. Users can enter keywords in the search bar to retrieve matching documents sorted by significance. For example, users can search a certain folder in the mailbox database.

security: In many large organizations, access to information in databases needs fine-grained access control as well as security mechanisms for authentication. Every Notes user has a digital ID and access can be granted at different levels, from the server down to individual document fields.

6.5.4 RENotes Case Study

RENotes is our reverse engineering application that we built as a case study to gain experiences with our approach to tool-building. It leverages Notes features wherever possible, supplementing them with custom functionality where needed. In this section, we describe RENotes’ architecture and implementation, list the supposed benefits to its adoption, and relate our experiences with implementing and using the application.

RENotes has been implemented with a standard three-layer architecture. Its major components are shown in Figure 45. RENotes represents the structure of the system under examination with a typed, directed, attributed graph, similar to the one used in Rigi. The graph is initially produced using existing source code parsers (e.g., Rigi’s `cparse` for C, or the CPPX fact extractor for C++ [DMH01]) and saved in Graph Exchange Language (GXL) format [HWS00]. GXL is an XML-based exchange format popular in the reverse

Architecture and
implementation

engineering community. Its syntax is defined with an XML Document Type Definition (DTD).

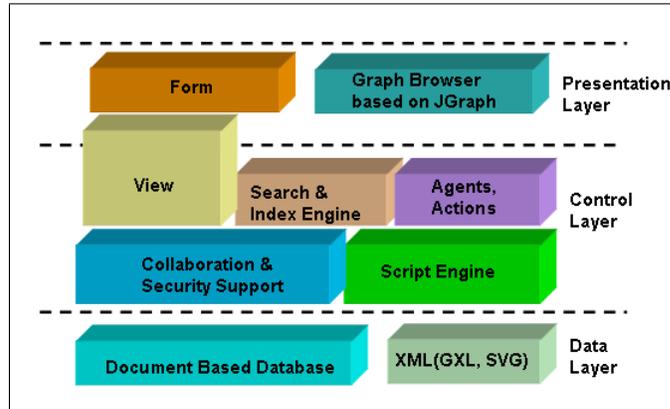


Figure 45: RENotes' layered architecture

The generated GXL file can be imported into RENotes as a Notes database through a Java agent. To perform the transformation, the GXL file is parsed into an XML Document Object Model (DOM) tree. The XML DOM tree is then traversed and elements are converted into the Domino Object Model.

Notes exposes its internal state with the Domino Object Model [TCG⁺02]. This model allows programmatic access to and manipulation of the databases and application services. It has been implemented for a broad range of languages, including the Formula language, Lotus Script, Visual Basic, JavaScript, and Java. Each object defines a set of properties and methods. For example, the `NotesDocument` object represents a document in a database and has the method `AppendItemValue` to add a new field to the document.

The graph to database mapping is simple: each node and arc is mapped to a separate new document with an automatically generated unique identifier. The type and other attributes of each graph element are saved in the corresponding document's fields. Figure 46 shows a database with a small graph of 14 nodes. During the import, the graph is also checked against a source code language-specific domain schema, encoded in XML and held in a separate Notes document. This ensures that the graph is well-formed and meaningful so that other tools can use it safely.

Once the data has been imported, users can manipulate the documents with all the usual Notes tools. They can search for specific nodes or arcs by keyword (cf. Figure 47), or create filtered, sorted views of the (automatically indexed) database based on complex queries. The documents can also be accessed through the standard automation features of Notes, allowing users to write ad hoc scripts to perform more complex operations such as bulk attribute changes or transitive closures on the system's directed relationships. The user can also select from a set of predefined scripts that perform typical reverse engineering tasks such as to find all callers of a function or accessors of a field. These existing scripts

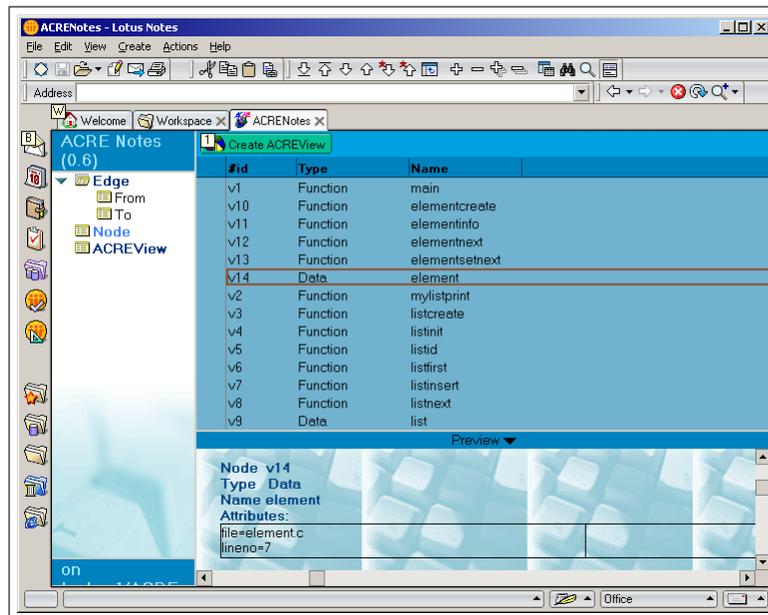


Figure 46: Nodes in a sample RENotes database

provide useful templates as a starting-point for users who want to write their own scripts.

Access to the RENotes databases is controlled by the security features of Notes; RENotes defines some common user roles with various degrees of privilege, restricting users' actions with fine granularity. All this functionality is leveraged unchanged from Notes and should be familiar to its users.

The generic textual list views provided by Notes are often not optimal for exploring the structure of a system, so RENotes provides a custom-built graph browser. The user first selects a subset²³⁵ of nodes to visualize. References to the nodes are gathered into a perspective document from which the user can launch the RENotes graph browser (see Figure 48).

Visualization

The browser—written in Java using the open-source, Swing-based JGraph²³⁶ graph editing toolkit and embedded in the RENotes database—provides a visual representation of the nodes and all relationships between them. All graphical elements are connected to the underlying Notes documents, using the domain schema to map their properties to visual attributes. The browser offers basic navigation, manipulation (most notably filtering of nodes and arcs), and layout controls to help the user investigate the system's structure.

When the user exits the graph browser, its state is saved in the corresponding perspective document, encoded as XML. This allows the developer to resume exploration of the graph in a subsequent session. Each perspective document thus represents a separate persistent view on the system graph; the same node may be independently present in many

²³⁵The subset may, in fact, be the whole graph.

²³⁶<http://jgraph.sourceforge.net/>

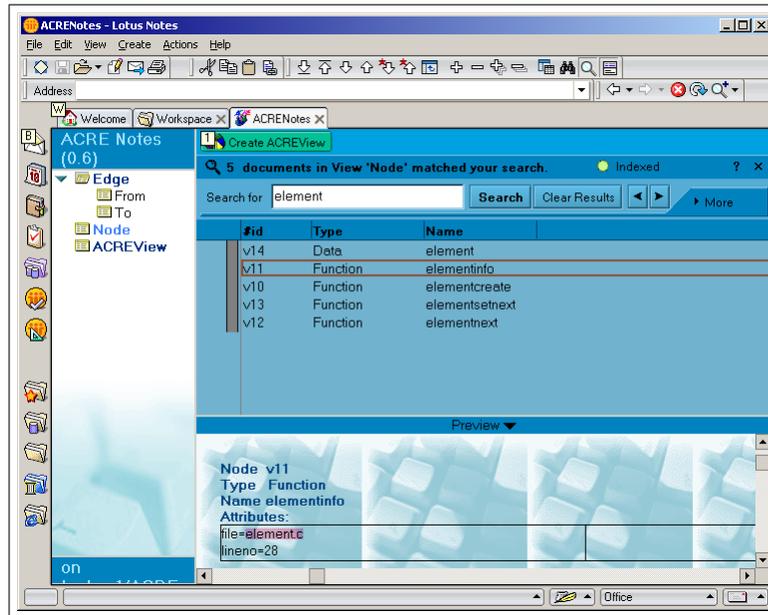


Figure 47: Keyword search of a RENotes database

perspectives. The graph visualization can also be saved in Scalable Vector Graphics (SVG) format [Fer01] so that it can be embedded into other documents, or shared with people who are not using RENotes or do not have the access privileges necessary to open a perspective document.

6.5.5 Conclusions

The RENotes project has generated some interesting insights and potential benefits to adoption, though more work is necessary to further evaluate the effectiveness of our approach.

Building our application in Lotus Notes rather than as a stand-alone application greatly reduced our development effort. The benefits of reusing Notes functionality more than offset the overhead of adapting to a new development environment, reducing lines of code by an (estimated) order of magnitude. This also allowed us to build several prototypes to verify and test the feasibility of our ideas.

The JGraph framework also proved helpful: the graph browser application weighs in at less than 4,000 lines of code. By comparison, Rigi has about 30,000 lines of C/C++ code, though it provides a richer feature set than RENotes does. Overall, the majority of the effort was directed at leveraging the potential the Notes environment provides, as opposed to writing new code from scratch.

As with any framework, there are also some limitations. Notes does not let Java clients control its user interface, preventing the graph browser from tightly binding visualized node selection to a Notes document view. While other APIs (for example, the Notes C API) may

Development
experience

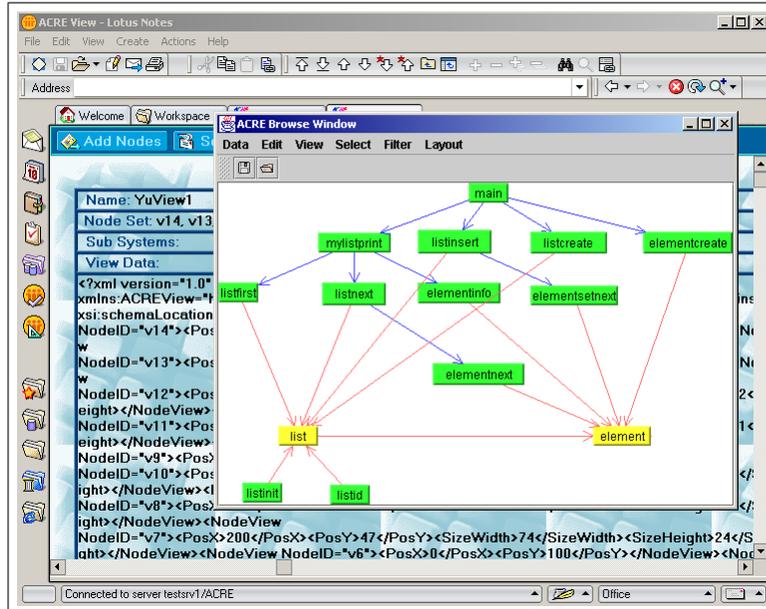


Figure 48: Visualization of a RENotes database

afford more control, we did not consider these languages suitable for rapid development of reliable software. There are also potential issues with scaling RENotes to handle larger systems. While we did not try large-scale experiments, we are cautiously optimistic on this count because the Notes database kernel has been refined and optimized over the last decade.

RENNotes made substantial strides towards improved adoption by building on top of Notes. The relatively large base of Lotus Notes users (as compared to research prototypes) means that support is plentiful and basic training is easy to obtain. Since RENotes reuses many of the features of Notes, the learning curve is significantly lowered for existing users, and even new users benefit from the support network of Notes. Notes also has a mature, coherent user interface and many convenient tools, such as search and filters. Together with the popularity of Notes, this should make RENotes more appealing to domain experts with no reverse engineering experience.

Benefits to adoption

Notes support for ad hoc end-user programmability is also critical to reverse engineering efforts. Many reverse engineering tasks are tedious to perform manually and, if so accomplished, cannot be recorded and reused. Notes is fully scriptable and even offers users a choice of programming languages that cover the spectrum of formality, letting them pick a tool appropriate for the task at hand. The abundance of scripting options also makes it more likely that a user already knows at least one of them, further lowering the barrier to adoption, and comparing favorably with research prototypes that normally support at most one scripting method.

Building on Notes lets us leverage its support for collaboration, which should become

more important as reverse engineering projects grow more complex. RENotes databases integrate well into the Notes workspace and can be linked to other databases employed by the user. By providing GXL import and SVG export capabilities, RENotes also integrates well with applications outside Lotus Notes, making it easier to fit into an existing workflow.

In future work, we will add more of the Rigi functionality and reverse engineering capabilities to RENotes. Furthermore, we want to observe how effectively Notes' baseline environment is utilized by RENotes users and how users will integrate RENotes into their work environment and processes.

Future work

6.6 REGoLive

Many Web sites are highly complex software systems [Off02]. They uniquely combine features found in databases, distributed systems, hypertext, and highly secure systems. Unfortunately, Web sites are often constructed and maintained in an ad hoc manner by developers that have little or no background in software engineering [Pre00]. Consequently, Web sites often do not follow Web site development standards and lack documentation [TMSM02]. As a result, many Web sites are difficult to understand and maintain. Given that often more than half of the time for the maintenance of traditional software systems is spent on program understanding, approaches that help Web site understanding should greatly contribute to ease development and maintenance activities.

Whereas early Web sites have been constructed with little dedicated tool support, there are now a number of sophisticated Web site authoring tools available such as Microsoft FrontPage, Macromedia Dreamweaver, and Adobe GoLive. These tools have the capability to generate and maintain Web sites. Offutt notes that they "provide point-and-click ability to create Web pages that can even be enhanced with dynamic HTML and JavaScript while requiring very little knowledge of HTML and programming" [Off02]. Such sophisticated tool features are mostly geared towards forward engineering (i.e., construction and deployment of the Web site). However, the maintenance aspect of Web sites is today at least as important as their initial construction.

In the past, software development tools (such as compilers) had very limited program comprehension functionalities; such functionality was provided by other general (e.g., `grep`) or specialized (e.g., `etags`) stand-alone tools [TW00]. Whereas modern integrated development environments (IDEs) are primarily geared towards forward engineering, they now have a growing range of reverse engineering and program comprehension functionalities such as generation of UML class diagrams, views of inheritance trees, direct navigation from references to declarations, and refactoring support (e.g., Eclipse). Similarly, Web authoring tools have now functionalities that are useful for reverse engineering and program comprehension. This trend benefits software engineers, because many of their maintenance tasks combine forward engineering with reverse engineering activities.

However, IDEs and Web authoring tools are still missing more sophisticated comprehension functionality such as interactive graph visualizations, which are commonplace in reverse engineering and program comprehension tools. Tool builders of advanced com-

prehension functionality can take advantage of existing, popular products that developers already use, and integrate their comprehension functionality into them. This approach promises to increase adoptability compared to offering these functionalities in stand-alone research tools, because developers can use the new functionality within their familiar environment.

For example, let's assume that a maintainer wants to better understand a Web site that has been built with GoLive by using a stand-alone reverse engineering graph of the site's link structure. This graph might expose a page that does not adhere to the site's linking conventions. In this case, the maintainer would prefer to directly navigate from the graph (which renders the page as a node) to the corresponding GoLive entity, which might be a static HTML page or a JSP-generated one. To manually trace a graph node to the corresponding GoLive entity is a potentially complex, cumbersome, and error-prone activity, which greatly mitigates the benefits of the reverse engineering graph. Conversely, maintainers are more likely to appreciate and adopt a reverse engineering graph if it is seamlessly integrated into GoLive.

In previous research we have already implemented software engineering functionality on top of Visio, Excel, PowerPoint, and Lotus Notes. Here, we discuss our experiences utilizing a commercial Web authoring tool (i.e., Adobe GoLive) as a host product and programmatically extending it with additional capabilities for Web site comprehension.

6.6.1 Related Research Tools

Part of the functionality that REGoLive offers is similar to other stand-alone program and Web site comprehension tools. Many research reverse engineering tools (e.g., Rigi, SHriMP, Moose, and Ciao/CIA) are based on the extraction of static information from source and on the visualization of this information as typed, directed graphs. Research involving reverse engineering tools has demonstrated that they can help program comprehension and ease maintenance.

Similar tools have emerged that target the domain of Web sites, facilitating, for example, architecture recovery [HH02] and maintenance activities [KWMM03]. Ricca and Tonella have developed the ReWeb tool, which consists of an extractor, analyzer, and viewer for static Web sites [RT01b]. The extracted Web site is represented as a typed, directed graph. The ReWeb tool has several analyses that operate upon the graph structure. Most of these analyses are inspired by traditional compiler (flow) analyses and mapped to the Web site domain (e.g., dominators of a page, shortest path from the start page to a page, and strongly connected components). Results of analyses are visualized with AT&T's `dotty` (an extensible graph editor). Similarly to ReWeb, the Rigi reverse engineering environment has been extended to visualize the structure of crawled Web sites [MM01a]. The HTML crawler is hand-written in Java.

Hassan and Holt have developed coarse-grained lexical extractors for HTML, JavaScript, Visual Basic, SQL database access, and Windows binaries [HH02]. During the extraction process, each file in the local directory tree that contains the Web site is

traversed, and the corresponding file's extractor (depending on the file type) is invoked. Extracted facts are represented in the Tuple Attribute format. All extractors' output is first consolidated into a single file and then visualized as a graph with PBS [FHK⁺97].

6.6.2 Adobe GoLive as a Host Product

We chose to use GoLive as host product for our tool implementation because of its popularity, maturity, and extensive customization support. For Web site authoring, GoLive is among the most popular tools according to a study conducted in March 2005 [Sec05].²³⁷ Furthermore, GoLive is a mature product that has evolved through several major releases; we are using Version 6.0. Other popular Web authoring tools such as FrontPage and Dreamweaver have similar capabilities and could have been picked as well.

Existing Comprehension Capabilities Before extending GoLive, we first investigated its existing features and how to utilize them for reverse engineering and comprehension functionality. Specifically, we identified existing program comprehension capabilities already present in GoLive using Tilley's Reverse-Engineering Environment Framework (REEF) as guidance [Ti00] [Ti98a].

The REEF identifies reverse engineering tasks (e.g., redocumentation and program analysis), and defines three canonical reverse engineering activities: data gathering, knowledge management and information exploration. It has been used to investigate the capabilities of a number of tools that aid program comprehension, including a study of several Web authoring tools [TH01]. Following the framework, we have identified the following capabilities for GoLive:

redocumentation: The process of retroactively providing documentation for an existing software system is called redocumentation.

In GoLive, developers can use design diagrams (cf. Figure 49) to record their initial design of the Web site structure. If such a diagram exists, it can be a good starting point for redocumentation. A design diagram shows pages, (potential) navigations between pages, and hierarchical relationships between pages. Similar to UML diagrams, the design can contain annotations. Navigations that are proposed in the design diagram, but have not been realized yet, are shown in the Pending Links view. The Navigation view shows the hierarchical structure of the site. Thus, these views can be used to assess differences between the proposed design and the actual site.

program analysis: Currently, Web authoring tools offer relatively limited analyses compared to sophisticated flow analyses such as used for aliasing, slicing, clustering, or clone detection.

GoLive has Clean Up Site and Remove Unused commands to remove colors, font sets, links, et cetera that have been defined but are not used in the actual Web site.

²³⁷The study is based on crawling of Web sites and examining the pages' generator meta tag.

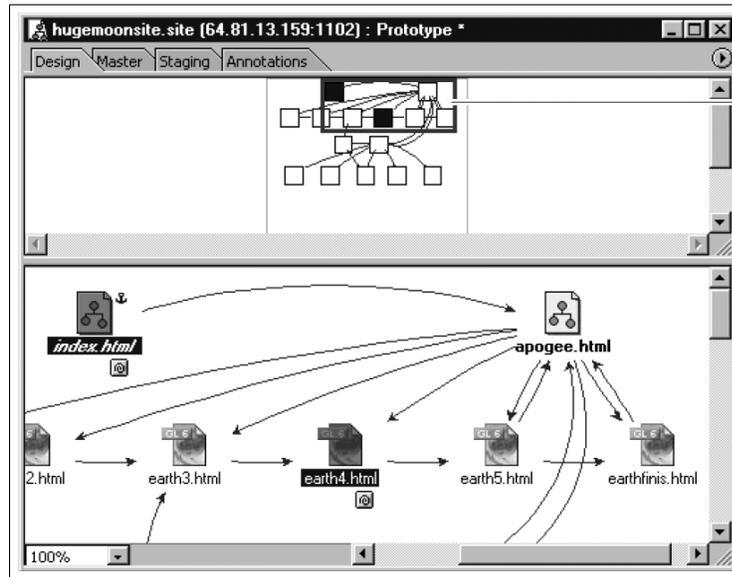


Figure 49: GoLive Design view ([CF03])

Fix Errors reports missing files and links. Check External Links tests external links whether they are still valid.

data gathering: This activity gathers raw data about the system (i.e., artifacts and relationships between artifacts).

GoLive has a number of parsers for markup documents (which create markup trees). Furthermore, a whole Web site can be imported into GoLive from HTTP, FTP, or WebDAV servers (including associated components such as images, CSS files, and script library files).

knowledge management: This activity structures the data into a conceptual model of the application domain.

Typically, the conceptual model of a Web site shows pages and navigations between pages. GoLive has several views to summarize, navigate, and manipulate this information. As discussed before, the Navigation view shows the hierarchical organization of pages. The In & Out Links view is a link management tool that graphically shows the links to or from a selected file. Similarly, the Links view (cf. Figure 50) shows the recursive link structure starting from a certain file.

information exploration: This task is the most important and most interactive of the three canonical activities. The software engineer, typically a maintenance programmer, gains a better understanding of the subject system with interactive exploration of the information that has been obtained by data gathering and knowledge management.

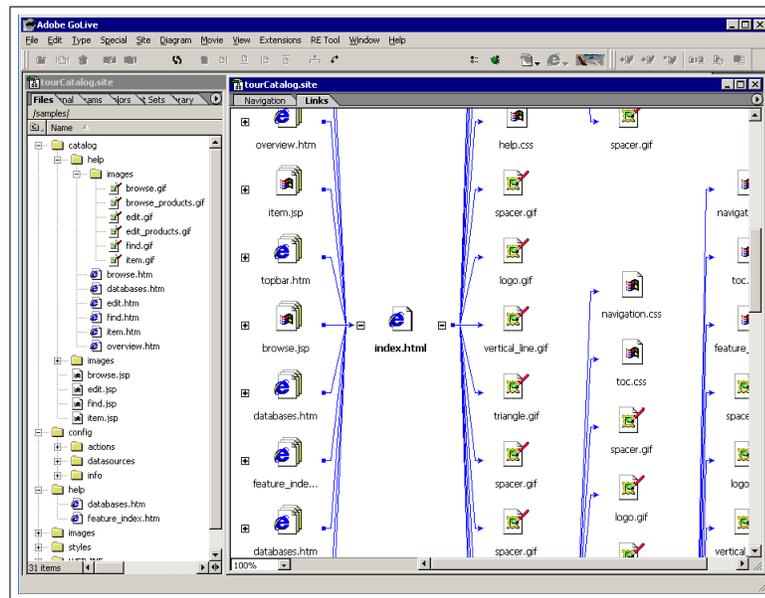


Figure 50: GoLive Links view

GoLive represents information with views. There are a large number of views, showing various properties of the Web site. The Files view lists the files (e.g., pages, images, and scripts) belonging to a Web site. Some views focus on a single page (e.g., Source Code Editor and Layout Preview), while others show relationships between pages (e.g., In & Out Links and Navigation).

Whereas GoLive offers information exploration with views, it has no graph visualization, which is the preferred visualization of most program comprehension tools. As a result, information in GoLive is dispersed over several views. A complementary graph visualization providing a unified view of a Web site and allowing sophisticated manipulations such as building of hierarchies would be desirable.

Extending GoLive The GoLive Extend Script Software Developer's Kit (SDK) enables customization via so-called Extend Scripts [Ado02b]. Such a script mainly consists of a dedicated HTML/XML file called `Main.html`, which contains custom tags (identified with the prefix `jsx`) and JavaScript code to define the extension. At start-up, GoLive interprets these tags and scripts to load an extension into the GoLive environment. The custom tags declaratively specify extensions that the Extend Script implements such as menus, dialogs, palettes, inspectors, and custom tools.

Depending on the extension, the Extend Script has to implement a number of JavaScript callback functions, which are invoked by GoLive in order to signal events. For example, at application start-up, GoLive calls each extension's `initializeModule` function. To give a flavor of how Extend Scripts look, here is a rudimentary example:

```
<html><body>
  <jsxmodule name="MyAlertExtension">
    <script>
      function initializeModule() {
        alert ("Hello, World!")
      }
    </script>
  </body></html>
```

GoLive's SDK provides numerous JavaScript objects and methods to programmatically manipulate files and folders as well as the contents of documents written in HTML, XML, JSP, et cetera. The document content that has been read into memory is made available in GoLive through a Document Object Model (DOM), which allows to query and to manipulate markup elements. Thus, batch processing of changes to an entire Web site can be easily accomplished.

Interestingly, since Extend Scripts are essentially HTML/XML documents, they can be easily edited in GoLive itself.

6.6.3 REGoLive Case Study

Program comprehension tools are usually handcrafted and stand-alone. Our tool-building approach for REGoLive is different, because we are leveraging an existing product and augment it with program comprehension functionality. When doing this, we can take advantage of the (program comprehension) functionality already offered by the host product, focusing on the missing pieces. As a result, GoLive users engaged in program comprehension can seamlessly transition to REGoLive's functionalities. The functionality that we added shows that our tool-building approach is both feasible and practical.

Viewpoints As discussed in the previous section, GoLive already offers a number of rudimentary features that can aid program comprehension. Most of these features are offered via views, which provide software engineers with information about a Web site's assets (such as pages, links, and database schemas). Even though these views have not been explicitly designed with program comprehension in mind, they are reasonably well suited for it. However, as explained later on, these (tool-centric) views are not usually sufficient to expose the complex interrelationships of a Web site.

To comprehend a software system, an engineer typically works with several different viewpoints. For traditional software systems, written in a high-level programming language, among the more important viewpoints are the development (or source code) view, the build-time view, and the run-time view [TG01]. During comprehension, software engineers frequently switch between viewpoints, mentally constructing relationships between them. For example, when exploring the run-time view and observing a certain behavior, the software engineer typically wants to know which parts of the source code are responsible

for this behavior. There are tools to construct such mappings for traditional systems, ranging from symbolic debuggers and trace information to concept analysis. However, tools such as GoLive do not provide comparable mappings for Web sites.

Similarly to traditional software systems, one can distinguish different viewpoints for the Web site domain. We believe that the following ones are most important for Web site comprehension [KWMM03]:

developer view: The view of the Web site that a developer (using a Web authoring tool such as GoLive) sees.

server view: The view of the Web site that a maintainer sees on the Web server. This view is the result of telling the Web authoring tool to deploy the site. (Deploying a site with GoLive is achieved via FTP or WebDAV.)

client view: The view of the Web site that a client (typically using a Web browser) sees.

To understand a Web site, maintainers typically explore the site in the client view (potentially semi-automatically using a Web spider) and, during this process, construct mappings to the server and developer views. The reverse engineering community has developed a number of tools to aid program comprehension in the client view (e.g., ReWeb [RT01b] and Martin and Martin's tool [MM01a]), and the server view (e.g., Hassan and Holt's tool [HH02]). But, to our knowledge, there are no research tools available that support the developer view. This is not surprising because program comprehension tools are typically stand-alone, operating independently of a concrete development environment or else Web authoring tool. Furthermore, all of these tools offer support for a single view only, failing to provide relationships between different views. For example, to our knowledge no (academic) Web site comprehension tool currently allows the maintainer to trace a page that is displayed in a browser to the corresponding asset in the Web authoring tool that has been used to construct the page.

The available program comprehension functionality of Web authoring tools (including GoLive) addresses almost exclusively their own developer view. This is a result of these tools' emphasis on forward engineering, in which a site is first constructed (working in the developer view) and then deployed. Examples of the few functionalities in GoLive outside of its developer viewpoint are the FTP and WebDAV views (which offer server-view information about the server's file system) and the Layout Preview (which show an approximation of the client-view rendering of an HTML page).

Following these observations, we believe that having the Web developers in control—that is, they being able to generate different interactive, consistent, and integrated views from GoLive, and to establish mappings between the views—facilitates Web site maintenance.

Comprehension of Generated Contents The importance of the viewpoints and the mappings between them for program comprehension is most apparent for generated contents.

In this case, mappings between views are often not straightforward. For example, dynamic server-side scripts (such as JSPs or servlets), which are deployed on the server side, can generate multiple HTML pages (e.g., based on different user inputs) in the client view [GKM05c]. When generating a page, the script may integrate data from Web objects or databases. To facilitate comprehension and maintenance, REGoLive makes these mappings explicit for the Web developer.

Similar to ReWeb and Martin and Martin’s tool, REGoLive’s Client view (cf. Figure 51) allows the Web developer to explore the Web site from the user’s perspective. The Client view offers a coarse-grained abstraction of the Web site, which identifies pages and their link-relationships. This allows maintainers to get a quick overview of the site structure and its complexity. In contrast to Martin and Martin’s tool, the Client view also offers more fine-grained information, visualizing for example inner page components such as forms and their associated relationships. Maintainers can use filters to show/hide certain artifacts and relationships. Filtering makes it possible for maintainers to fine-tune the visualization, showing only information that is appropriate for a particular task. Furthermore, detailed information about each node (e.g., its source file location) is accessible through a pop-up menu (cf. the “node properties” window in the graph view of Figure 52).

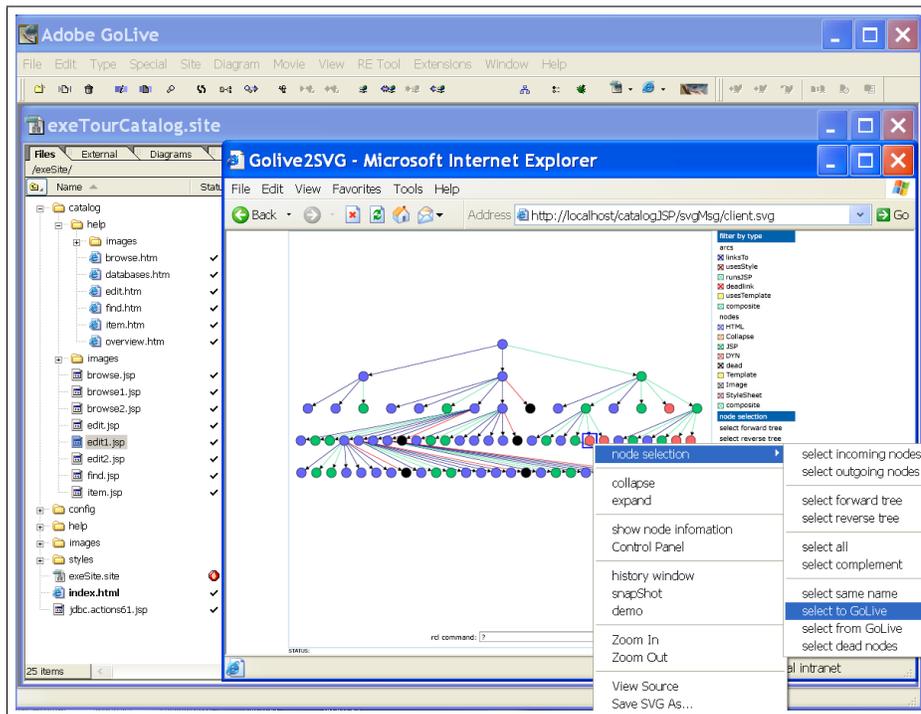


Figure 51: REGoLive Client view graph

Figure 51 shows an example of a Client view graph generated from a medium-size GoLive demo Web site consisting of about 400 files (including 11 HTML files and 14

JSPs). The graph is rendered with SVG in Internet Explorer. The top panel on the right side of the graph shows the node/arc types and their filtering status. Different node colors are used to identify different site artifacts. For instance, blue nodes denote static HTML pages (i.e., there is a one-to-one mapping to the corresponding HTML file on the server), whereas green nodes show JSP-generated HTML pages (i.e., the generating JSP files can be found on the server). During browsing, a single JSP file can generate multiple, dynamic HTML pages (DYN); such pages are shown as red nodes. We render all of these generated pages as distinct nodes, because each one represents a different execution path triggered by the user's browsing activity.

REGoLive also provides mappings from Client view entities to GoLive Developer view entities. These inter-view relationships are a unique feature of our tool. With a pop-up menu (cf. menu item “select to GoLive” in Figure 51) it is possible, for instance, to directly navigate from a graph node to the corresponding entity in GoLive, and vice versa. When a node in the REGoLive graph is selected, the corresponding entity in GoLive's tree view is highlighted and the corresponding document is opened. These features make it easier for maintainers to seamlessly transition between REGoLive's and GoLive's functionality.

REGoLive's Server and Developer view visualizations offer comparable functionalities to the Client view, differing mainly in the types of artifacts and provided mappings. Server view graphs are similar in spirit to the visualizations provided by Hassan and Holt's tool. However, our tool has fewer extractors; only facts from HTML files and JSPs are handled.

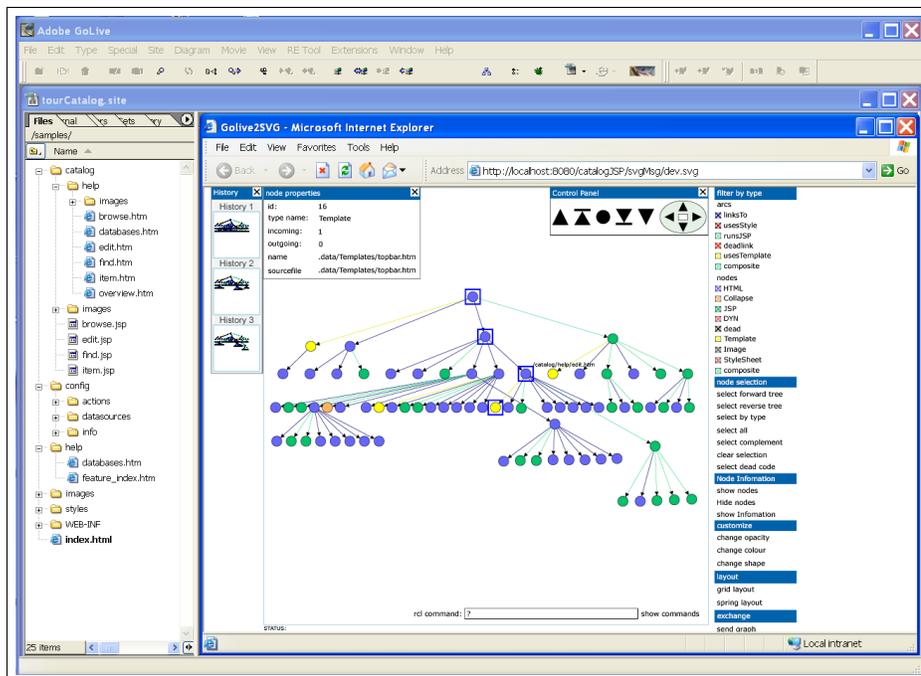


Figure 52: REGoLive Developer view graph

The Developer view graph (cf. Figure 52) focuses on GoLive-specific entities such as *templates* (shown as yellow nodes) [CF03]. Whereas JSPs are an example of an on-the-fly generative mechanism in the server view, templates are an example of a generative mechanism in the developer view. Templates are a mechanism that allows a consistent look of Web sites by restricting the parts of a page that can be edited in the Layout editor. However, once a Web site has been published with GoLive, it is difficult to determine whether a particular file on the server has been generated via templates or not. This makes it difficult to maintain a site because a manual change in a server file can be overwritten unknowingly when information based on templates is subsequently regenerated. REGoLive's Developer graph allows maintainers to quickly assess whether a Web site uses templates and which pages are based on templates. Conversely, if maintainers have to make changes to a template (e.g., changing styles or headers), they can assess the propagation of their change. Besides templates, REGoLive can also visualize the use of *smart objects*, which offer pre-packed (JavaScript) functionality that can be placed on a Web page via drag-and-drop [CF03]. Examples of smart objects are timestamps, pop-up menus, and automated redirects based on users' browsers. A Developer view graph exposing both smart objects and templates can be useful to assess the extent of GoLive-specific entities when migrating a Web site to another Web authoring tool or to another technology.

We have experimented with REGoLive using several demo sites that are part of GoLive. These Web sites are well suited for our purposes because they are fairly sophisticated, combining general Web technologies (i.e., HTML and JSP), a database back-end (i.e., MySQL accessed via JDBC), and GoLive-specific features such as templates. Gui's thesis contains more detailed reverse engineering and maintenance scenarios [Gui05].

REGoLive Implementation When implementing new functionality via extension of a host product, one has to first identify the suitable extension mechanisms and existing functionalities. This section describes how we extended GoLive and what existing functionality could be leveraged.

Most reverse engineering environments have a similar conceptual architecture, consisting of the following four component types: repositories, extractors, analyzers, and visualizers. In the following, we discuss the key functionalities of REGoLive separately for each component type:

repository: In REGoLive, information is stored in files on the local file system. This approach is simple and portable since GoLive's API supports all necessary file manipulations. Extracted information is written in RSF and GXL formats [HWS00].

The schema consists of entities such as HTML files, server-side script files (e.g., JSPs), images, CSS files, text documents, as well as GoLive-specific entities such as templates and smart objects. Components of HTML files such as forms are modeled as well. Relationships include links between HTML pages, execution of server-side files, uses of templates, smart objects and CSS files, et cetera. Except for the GoLive-

specific information, most of the modeled information is similar to the schemas used by other research tools [HH02] [KWMM03] [DLFT02].

extractors: For REGoLive, different extractors are needed to retrieve artifacts from the Developer, Server, and Client views. From the developer's perspective, the subject system includes Web pages and various tool-specific objects; from the server's perspective, it includes all the source code of the Web site; from the client's perspective, the subject system consists of the generated pages sent from the server and rendered in the Web browser.

Extraction of artifacts from HTML is easily accomplished in GoLive with the DOM, which makes it possible to traverse a document's complete markup tree [Ado02b]. The developer-view extractor parses all files that are contained in GoLive's Files view (which is part of the Site window). This extractor also identifies generated components, such as templates and smart objects. The server-view extractor works on a server side copy of the Web site, which can be obtained from GoLive's FTP browser (through which the Web site was originally deployed). To obtain the pages for the client-view, REGoLive's spider is used to crawl the site.

analyzer: The REGoLive analyzer infers relationships from the extracted information and builds multiple, layered hierarchies as higher-level abstractions to reduce the complexity of understanding large Web applications. It produces a spatial site structure, at various levels of details, and identifies and builds the mappings between the three viewpoints. The constructed information can be saved in RSF or GXL format, or directly sent to the visualizer. The analyzer also collects dead links in the client view (accessing GoLive's links status information).

visualizers: The REGoLive visualization engine renders a graphical presentation of the site structure in a graph editor. Extensions in GoLive can obtain references to a draw object, which has methods to render lines, ovals, rectangles, text, and images [Ado02b]. Whereas the capabilities of draw objects are sufficient for simple graphs, we found that the rendering of a graph with hundreds of nodes on the Palette Dialog is too slow, especially if user interactions require repainting. Furthermore, programming of graph interactions is difficult because graphical objects have no associated mouse or keyboard events.

Since rendering of graphs in GoLive is not feasible, we decided to render the graph outside of GoLive with SVG in a Web browser, which the user can launch directly from GoLive. The graph is loaded into our SVG Visualization Engine (cf. Section 6.3), which is a generic graph visualizer (written with SVG and JavaScript), which can be easily customized for a particular domain. With the visualizer, the user can explore, annotate, filter, select, rearrange, and layout graphs.

However, for security reasons, SVG's JavaScript code is prohibited to access the local file system. To work around this restriction, we let GoLive act as a file server,

using a Web service interface [Gui05]. The SVG visualizer requests a local file from REGoLive (via sending a URL to GoLive). The file is then sent to the SVG visualizer as a Web service response.

6.6.4 Discussion of Tool Requirements

In this section we address the specific benefits and drawbacks of our tool-building approach; we first discuss the tool-user's, followed by the tool developer's perspective.

Tool User To guide our discussion, we address how REGoLive meets a number of selected tool requirements (cf. Section 3.2). Researchers in the area have (repeatedly) identified these general, non-functional tool requirements, which are independent of particular reverse engineering tool functionalities and domains [DT03]. Thus, tools that meet these requirements can hope that users will perceive them as more useful. According to these requirements, reverse engineering tool should be:

scalable: We have tested REGoLive with several sample Web applications provided with GoLive. One of these consists of about 400 files. Extraction and analysis of the sites do not cause performance problems, even though all files have to be programmatically opened in order to parse them. Generally, REGoLive does not negatively impact GoLive's performance.

The SVG visualization can handle graphs with hundreds of nodes adequately. Furthermore, the performance of the visualizer can be maintained by managing the graph at various abstraction levels.

extensible: Tilley states that "it has been repeatedly shown that no matter how much designers and programmers try to anticipate and provide for users' needs, the effort will always fall short" [Til95]. Constantly arising new technologies require extensibility, for instance, for changing repository schemas or extractors.

GoLive itself can be extensively customized and extended via JavaScript, which makes it possible to integrate new functionality seamlessly into the existing environment. Since our extensions are available in source in GoLive's plug-in folder, others can adapt REGoLive to their needs. However, generally extensions that are based on scripting languages are challenging to manage from a software engineering point of view.

interoperable: The environment should be able to interoperate with diverse components [JCD03]. The following integration levels can be distinguished (cf. Section 4.2.4): data (i.e., components using a standardized exchange format and managing the data as a consistent whole), control (i.e., components can send messages to each other), and presentation (i.e., components have a common look-and-feel from the user's perspective, reducing cognitive load).

Since GoLive's API allows programmatic file manipulation, data interoperability via reading and writing of files can be easily achieved. REGoLive generates SVG files, which are then visualized in a Web browser. However, our SVG visualization does not achieve presentation integration since the graph is not visualized using GoLive's native drawing capabilities. Both performance problems and limitation of GoLive's API are responsible for this decision, which, from an adoption perspective, is a significant drawback. For example, the user interface of the SVG visualizer is quite different from GoLive's interactions.

In REGoLive, control integration is achieved via Web services, which allow us to send messages between REGoLive and the SVG viewer. For example, selecting a graph node in SVG sends a message to GoLive to select the corresponding entities in the views.

adoption-friendly: A tool is only useful if it is actually used [Won99]. Intuitively, to encourage adoption, a new tool should be easy to install, have a favorable learning curve, offer documentation and support, et cetera.

REGoLive is easily installed by copying its `Main.html` to GoLive's module extension directory. Similarly, it can be easily uninstalled. REGoLive creates its own pull-down menu in GoLive and is thus conveniently accessible. Users of REGoLive can inspect the JavaScript source and make changes to better suit their needs; they can also easily redistribute these changes to other users. Third-party contributions such as REGoLive can be, for instance, submitted to the Adobe Studio Exchange.²³⁸

exploratory: Visualized information should be represented in different views, easy to browse, and interlinked [DLT00]. Views in GoLive are well suited for interactive exploration. Program comprehension extensions can take advantage of this and add their own information to either existing or new views. This approach benefits tool adoption since the user is already familiar with the concept of GoLive's views and can smoothly transition between the built-in and new views.

Our experiences suggest that REGoLive is able to satisfy most of the user requirements. It is scalable, extensible, exploratory, and adoption-friendly in some respects. Data and control interoperability have been achieved; however, the SVG graph visualization lacks presentation integration, which makes REGoLive less adoption-friendly.

Tool Developer From the developer's point of view, building on top of a product that offers many functionalities with potential for reuse has greatly reduced our development effort and facilitated rapid prototyping. Our JavaScript implementation of REGoLive was realized with about 2,500 lines of code. The SVG visualization engine that we reused consists of about 7,000 lines of JavaScript code. GoLive facilitates rapid prototyping with a

²³⁸<http://share.studio.adobe.com>

built-in JavaScript editor and debugger. New or changed JavaScript code can be immediately executed and tested.

Whereas the amount of writing code is greatly reduced, time has to be spent up-front with understanding GoLive's architecture, extension mechanisms, and API. GoLive's API is quite large and described in about 600 pages of documentation [Ado02b] [Ado02c]. As our solution to communicate via Web services shows, it is often necessary to experiment with the available APIs and to come up with innovative workarounds to realize some functionalities.

Before extending GoLive it is also necessary to first master its functionality in order to better understand its existing (comprehension and maintenance) capabilities and suitable places for extensions. GoLive is a fairly large and sophisticated application; the end-user book that we used has about 900 pages to describe its full functionality [CF03]. It took the implementor about four months to familiarize herself with GoLive's functionality and extension capabilities. REGoLive was then implemented in about two person-months.

6.6.5 Conclusions

In this case study, we have realized a tool, REGoLive, for Web site comprehension by reusing an existing OTS product, GoLive. The tool implements a sophisticated analysis and visualization, providing the reverse engineer with three different viewpoints (i.e., developer, server, and client) of the target site. When implementing REGoLive, we were able to leverage GoLive's existing capabilities (e.g., for parsing), but extending GoLive for sophisticated graph visualization could not be realized. This is an example of a typical trade-off when using (black-box) components [CW98]. If the component (or product) does not support the implementation of a certain requirement, then the requirement has to be either dropped, adapted, or implemented outside of the component.

We have assessed REGoLive based on tool requirements identified by researchers in the field. The results of this assessment suggest that REGoLive can meet most requirements and can compete with stand-alone research tools. Furthermore, we believe that REGoLive is more usable and adoptable for users that are already familiar with GoLive, compared to a stand-alone tool. Since REGoLive is integrated within GoLive, Web developers and maintainers can smoothly transition between GoLive's basic program comprehension capabilities and REGoLive's advanced views.

However, these hypotheses need further evaluation, for example, with a user study. The ideal participants for such a study are GoLive users with a similar skill set. One group of users would be assigned a set of Web site comprehension tasks using only the original GoLive, whereas another group would perform the same tasks with the help of REGoLive. We could then observe, collect and compare feedback from each group and find out how REGoLive facilitates Web site reverse engineering.

6.7 WSAD Web Site Extractor

This case study describes my implementation of a fact extractor for J2EE Web applications. Fact extractors are part of each reverse engineering toolset; their output is used by reverse engineering analyzers and visualizers. My fact extractor has been implemented on top of IBM's Websphere Application Developer (WSAD). The extractor's schema has been defined with the Eclipse Modeling Framework (EMF) using a graphical modeling approach. The extractor extensively reuses functionality provided by WSAD, EMF, and Eclipse, and is an example of component-based development. With this case study, I show how I used this development approach to accomplish the construction of my fact extractor, which, as a result, could be realized with significantly less code and in shorter time compared to a homegrown extractor implemented from scratch. I have assessed my extractor and the produced facts with a table-based and a graph-based visualizer. Both visualizers are integrated with Eclipse.

6.7.1 Background

The Internet has experienced a tremendous growth since its introduction. It has become a ubiquitous, readily-available commodity, similar to the telephone. In most cases, Web sites are the main access points to the Internet for clients. Thus, they have a strategic importance and are a critical asset for many organizations (commercial as well as non-commercial ones). In fact, for many companies (e.g., online retailers such as Amazon) the Internet is the only communication channel to do business with customers. Since the number of clients that have access to the Internet is already large and still growing, Web sites are expected to become more and more important assets.

Besides static Web sites that are exclusively realized with HTML (so-called brochureware [TH01]), there are increasingly dynamic sites that are realized with advanced technologies such as AJAX [Gar05] and J2EE Web applications [SSJ⁺02]. As a result, advanced Web sites are highly complex software systems [Off02]. There is now a broad range of development tools that support the construction of Web sites. Examples range from relatively simple tools such as Microsoft FrontPage and Adobe GoLive to complex ones such as IBM Websphere Application Developer (WSAD) and Vignette StoryServer.²³⁹ Google supports AJAX-based development with the Google Web Development Toolkit.²⁴⁰

In contrast to traditional reverse engineering that targets legacy systems written in languages such as Cobol, C/C++, and Java, Web site reverse engineering (WSRE) proposes to apply reverse engineering approaches to Web sites. Indeed, complex dynamic Web sites exhibit many similarities to traditional software systems. Thus, traditional reverse engineering approaches such as dependency graphs can be applied to Web sites. Reverse engineering functionality to understand Web sites complements existing functionality that is already offered by Web site authoring tools such as FrontPage, GoLive, and WSAD. Whereas these

²³⁹<http://www.vignette.com>

²⁴⁰<http://code.google.com/webtoolkit/>

Web authoring tools already offer rudimentary comprehension functionality, they are still missing more sophisticated comprehension functionality such as interactive graph visualizations [GKM05b] [TH01].

In order to offer reverse engineering functionality for Web applications it is necessary to first extract suitable facts. In the following, I describe the realization of a fact extractor for Web sites that leverages the WSAD 5.1.2 tool.²⁴¹ WSAD is realized with proprietary plug-ins that extend the open-source Eclipse IDE. My fact extractor is realized as a plug-in that extracts information from J2EE Web projects. The extractor's domain model is defined with the Eclipse Modeling Framework (EMF) [BSM⁺03]. The extracted facts can be also exported as Rigi Standard Format (RSF), which is a popular exchange format in the reverse engineering community [Won98] [Mar99]. This makes it possible to integrate loosely the fact extractor with external tools.

An important goal for realizing the fact extractor was the attempt to reuse existing functionality as much as possible. Instead of hand-crafting an extractor, which is a tedious and error-prone endeavor, we wanted to leverage functionality that is already provided. This approach greatly simplifies the development effort and leads to a more stable, predictable, and maintainable extractor.

The WSAD-based extractor reuses functionality offered by the following components:

WSAD: WSAD has a parsing framework that can handle J2EE Web applications. This includes HTML, XHTML, Servlets, JSP pages, tag libraries (. t l d), JavaBeans, and Web deployment descriptors.

EMF: The schema of the extractor is defined as an EMF model. Since EMF relates modeling concepts directly to their implementation in Java, models can be used to automatically generate code to create model instances, to navigate the instance model, and to persist instance models (as XML files). Furthermore, EMF can generate (rudimentary) editors to create and manipulate model instances.

Eclipse: The Eclipse IDE makes it possible to integrate the extractor with WSAD and other Eclipse plug-ins seamlessly. For example, the extractor is registered as a *project builder* [Art03], which means that it is automatically invoked whenever the underlying fact base changes. I also have implemented a simple tabular visualization of the extracted facts that is seamlessly integrated with the IDE as a *view*. Furthermore, the SHriMP plug-in²⁴² was used to visualize the extracted facts as a dependency graph in Eclipse.

²⁴¹More recent versions of WSAD have been re-branded by IBM as Rational Application Developer (RAD) (<http://www.ibm.com/software/awdtools/developer/application/>).

²⁴²<http://www.thechiselgroup.org/creole>

6.7.2 Fact Extraction for Reverse Engineering

“It seems that in reengineering the language is the variable and the problems are the constants.”

– van den Brand et al. [vdBSV98]

Reverse engineering efforts start with fact extraction from the target system’s sources. Extracted facts are typically stored in a repository or as in-memory representation, which is then queried by program analyses or visualizations.

In this section, I denote extractors for WSRE purposes as *Web site extractors*. Web site extractors are used, for instance, to maintain Web sites [MM01a] [RT01b], to study the evolution of Web sites [LON01a] [WBM99], to recover the architecture of Web sites [HH02], to migrate legacy Web technology (e.g., from Net.Data to JSP [LLHX01]), to obtain metrics for Web sites [WBM99], to check conformity to Web standards,²⁴³ and to assess accessibility for disabled users [BBDM01] [Kir02].

For traditional reverse engineering, the source base is typically homogeneous, written in a single high-level programming language. In contrast, Web sites employ a number of technologies besides HTML and hence have a highly heterogeneous source base. Despite these differences, fact extraction in both domains faces remarkably similar challenges. Potential problems that extractors confront are *irregularities* in the source base such as syntax errors, dialects, embedded languages, grammar availability, and customer-specific idioms [KS03a]. However, it should be noted that these irregularities cause problems because there is a mismatch of assumptions between the expectations of the extractor’s user and the functionality offered by the extractor. For example, if the user expects the extractor to be robust and to ignore syntax errors, a parser that expects the sources to conform to a certain syntax is unsuitable.

Often, there is the assumption that fact extraction is a trivial, routine part of the reverse engineering process that needs to be done only once at the beginning of the reverse engineering activity. However in reality, because of extraction problems, this step often turns out to be highly iterative, requiring significant time and resources to accomplish. Thus, it is important to select carefully a suitable extractor to minimize potential problems up-front. Because of extractor bugs or mismatched assumptions, Kazman et al. give the advice to

“validate the source information that is extracted. . . . A detailed manual examination and verification of a subset with the elements and relations against the underlying source code should be carried out to establish that the correct information is being captured” [KOV03].

Homegrown vs. Component-Based Extractors To construct an extractor, two different approaches can be distinguished:

homegrown: Such an extractor is developed from scratch, possibly using scanner or parser

²⁴³<http://validator.w3.org/>

generators (prominent examples are Lex and Yacc),²⁴⁴ or scripting languages such as Perl and Python.

component-based: A component-based extractor leverages an existing compiler, IDE, or other suitable tool that processes the sources and provides a suitable interface to access information about the sources. Examples of (front ends of) compilers that have been leveraged by extractors are the Gnu Compiler Collection (GCC), Edison Design Group (EDG), and IBM Visual Age C++ (cf. Section 5.2.2).

Besides their way of construction, extractors can be also classified according to their domain. We distinguish between traditional reverse engineering of high-level programming languages and reverse engineering of Web sites.

Table 11 gives examples of available extractors, classified by their construction approach (homegrown vs. component-based) and domain (tradition reverse engineering vs. WSRE). For each extractor, its implementation languages and/or tools are given in round brackets. The sources that the extractor can handle are given in square brackets. Looking at this matrix, an observation can be made: Whereas for traditional reverse engineering homegrown as well as component-based extractors are available, Web site extractors are almost exclusively developed in a homegrown fashion. This lack of component-based extractors might be explained with the relative immaturity of the WSRE domain. To my knowledge, besides the extractor that is discussed in this section, there is only one other component-based extractor, REGoLive (cf. Section 6.6).

The WSAD Web site extractor is similar to REGoLive in the sense that both leverage functionality provided by OTS products. The REGoLive extractor is written in JavaScript and uses GoLive's API to access the DOM of a parsed HTML file. Our extractor is written in Java and uses WSAD interfaces to access relevant information. In contrast to other extractors that can rely on published and documented APIs, the WSAD extractor uses unpublished interfaces.

The decision to follow a homegrown vs. component-based approach to extractor construction has a significant impact on the development effort, future maintenance, and testing of the extractor. These implications are briefly discussed in the following. Generally, leveraging components to build an extractor has similar implications to component-based development of software systems [HC01].

The development of an extractor is a significant investment in terms of time and effort.²⁴⁵ The source language that an extractor supports is typically non-trivial and the building of an extractor requires a fair amount of expertise in parsing, compiler construction, and schema development. Reusing the functionality of an existing component or tool can greatly reduce the effort to obtain the necessary facts for reverse engineering. However, when deciding on a component for reuse it is beneficial to have an understanding about the

²⁴⁴Note that with this approach the grammar specification is not already provided, but needs to be constructed as part of the extractor's development effort.

²⁴⁵Unfortunately, few researchers report the effort that went into an extractor. However, the available data suggest that an extractor can be developed within 2–4 months [MW03] [BH91] [BH92] [ALG⁺03].

Domain	Tool-building approach	
	homegrown	component-based
traditional reverse engineering	<ul style="list-style-type: none"> • A* (Awk/Yacc) [C, etc.] [LR95] • Bauhaus (Cocktail) [C] [CEK⁺00a] • tawk (Ponder) [C, etc.] [GAM96] • Scruple (C) [C, PL/AS, etc.] [PP94] • Columbus (C++) [C++] [FBTG02] • Keystone (Flex/C++) [C++] [MCG⁺02] • Rigi parser (Yacc) [C] • Revealer (Perl) [C, Java, etc.] [PFGJ02] • Riva (Perl/Python) [C/C++] [Riv04] ... 	<ul style="list-style-type: none"> • CPPX (GCC) [C++] [DMH01] • gccXfront (GCC) [C++] [PM02] [HMP03] • g4re (GCC) [C++] [KMP05a] • TkSee/SN (SN) [C++] • Moise and Wong (SN) [C/C++] [MW03] • vacppparse (VisualAge) [C++] [Mar99] • JavaDB (Eclipse JDT) [Java] • FORTRESS (EDG) [C++] [GPB04] • GENOA (EDG) [C++] [Dev99a] • Dali (Imagix 4D) [C/C++] [KC99] • Huang (SNiFF+) [C] [Hua04] ...
WSRE	<ul style="list-style-type: none"> • Warren et al. (Lex/Yacc) [HTML] [WBM99] • Martin and Martin (Java) [HTML] [MM01a] • ReWeb (Java) [HTML and JavaScript] [Ric03] • Hassan and Holt (Perl) [HTML, VBScript, etc.] [HH02] • Ricca et al. (DMS) [HTML] [RTB02] • Synytskyy et al. (TXL) [ASP] [SCD03] ... 	<ul style="list-style-type: none"> • REGoLive (GoLive) [HTML, XML, JavaScript, etc.] (cf. Section 6.6) • my extractor (WSAD) [J2EE]

Table 11: Fact extractor matrix

component's extraction technology. Lexical approaches such as Source Navigator (SN) are typically more robust, but can yield false positives or negatives. In contrast, syntactical approaches are more brittle (i.e., they can easily break if the source does not adhere to the expected syntax).

Once an extractor has been developed, it also needs to be maintained. The evolution of a programming language can result in a steady stream of language enhancements as exemplified by C++ and Java. Evolution progresses at such a speed that C++ compiler vendors have problems to keep up with the latest enhancements. Developers of research tools have less resources to support extractor evolution. For Web site extractors the situation is even worse because new Web technologies are constantly introduced, and existing technologies such as XML-based standards evolve rapidly. If the extractor relies on a component, it can take advantage of new component versions that support the most recent version of a language or technology. On the other hand, newer versions of components may need to be obtained and integrated (e.g., because older versions are no longer maintained or are not available for new platforms) even if there is no immediate benefit for the extractor. Furthermore, since the extractor relies on the specific functionality offered by a certain component, there is the danger of vendor lock-in.

Component-based extractors often do not have access to the leveraged component's source code. This is the case for OTS products such as VisualAge, GoLive, and WSAD. This means that it is often impossible to debug a component and to understand its inner workings. Components can have subtle bugs that are difficult to track down. For instance, Huang made the experience during an industrial project that "SNiFF+ has its share of bugs" [Hua04]. Once bugs are detected in a black-box component, they cannot be immediately fixed. One has to wait for an improved release that fixes the problem or modify the parsed sources to work around the bug. For instance, when using Imagix 4D, O'Brien had to comment out definitions of bit masks in C preprocessor code [O'B01]. Lastly, it can happen that missing functionality of a component is detected relatively late after significant effort has been spent already to utilize the component. For example, Moise and Wong have used Source Navigator to extract cross-language dependencies between Java and C, but found that certain dependencies could not be recovered that they needed because "SN did not provide enough information for a deep static analysis" [MW05].

Multi-Language Extractors Many extractors are tailored towards a single source. However, for a (global) analysis to be truly useful in a heterogeneous environment, language boundaries must be crossed. This is especially the case for sophisticated dynamic Web sites that combine diverse technologies. So far there are few multi-language (or polylingual) extractors for traditional reverse engineering (e.g., FORTRESS [GPB04], Moise and Wong [MW05], GRASP [HHBM97], and PolyCARE [Lin95]).

There are Web site extractors that handle (almost) exclusively hyperlinks (e.g., Warren et al. [WBM99], and Martin and Martin [MM01a]). However, Web site extractors increasingly extract relationships between heterogeneous sources. For example, to provide

suitable information to reverse engineers that have to understand ASP-based sites, Hassan and Holt extract information from HTML, VBScript, COM source code, and COM binaries [HH02]. Synytskyy et al. describe how island parsing can be used to effectively parse ASPs that intermingle HTML, Visual Basic, and JavaScript [SCD03]. As described in the next section, because of the complexity of J2EE Web applications my extractor is also multi-language.

Generally, the understanding of complex frameworks such as .NET and J2EE require the extraction and interrelations of facts from multiple sources. Pinzger et al. discuss an approach to reconstruct the architecture of COM+ applications, which requires extractors for Visual C++ and Visual Basic as well as extractors for meta-data and configuration information from type libraries and the Windows registry [POG03]. Han et al. discuss architecture reconstruction for J2EE Web applications. However, they use a manual approach to reconstruction and hence do not implement extractors [HHN03].

6.7.3 Leveraged Functionalities

My extractor leverages WSAD, a commercial OTS product, to extract facts from J2EE Web applications. Besides the functionality of WSAD, the extractor also uses functionality provided by EMF and Eclipse. The following sections discuss the leveraged functionalities.

Websphere Application Developer WSAD is an IDE for the development of J2EE applications. This includes J2EE Web applications, which are developed as Web projects. Our Web site extractor uses WSAD 5.1.2.²⁴⁶ A Web project consists of a variety of sources: HTML pages, Cascading Style Sheets (CSS) files, graphics files, Java Server Pages (JSPs), Java servlets, JavaBeans, Java code, Web deployment descriptors (`web.xml`), Tag libraries, et cetera. WSAD's project navigator view shows a hierarchical listing of all resources that are part of a project (see top left view in Figure 56).

WSAD distinguishes between dynamic and static Web projects. The latter contain primarily HTML pages. Web projects can be packaged and then published to a server in the form of a Web archive (WAR) file. WSAD is a full-fledged Web site authoring tool. It has wizards to create new Web sites, an interactive Web page designer for HTML and JSP pages, a Web site designer to specify high-level structure, and specialized editors for CSS files and deployment descriptors. WSAD also has a links view that gives a simple graphical rendering of the relationships in a Web site (see bottom left view in Figure 56). For instance, this view shows hyperlinks between HTML pages and include-relationships of JSPs. Existing Web sites can be imported by crawling a URL or by providing a WAR file.

My extractor compiles diverse information from Web projects. Most Web site extractors focus on the extraction of hyperlinks between static HTML pages. However, J2EE Web projects have additional relationships between diverse resources that are important for reverse engineers to comprehend a dynamic Web site. WSAD maintains a data structure

²⁴⁶This release supports the Servlet 2.3 and JSP 1.2 specifications.

that keeps track of (hyperlink) relationships, which my extractor leverages. Examples of relationships are hyperlinks in HTML pages, references to tag libraries in JSP files, and Servlet mappings. For each project resource, WSAD's link repository can be queried for links that are contained in it. Depending on the actual kind of the queried resource, different parsers are invoked transparently that extract the links. The link repository uses an underlying cache to avoid repeated parsing of resources.

```

1  IFile file;
2  ILinkCollector collector;
3
4  ILinkTag linktags[] = collector.getLinks(file, null);
5  for( int i = 0 ; i < linktags.length ; i++ ) {
6      ILinkTag linktag = linktags[i];
7      IPath linkpath = linktag.getTargetResourceFullPath();
8      String link = ((IGeneralLinkTag) linktag).getAbsoluteLink();
9      String enclosingTag = linktag.getTagName();
10
11     System.out.print("Link in " + file.toString()
12                     " to " + linkpath.toString() );
13 }

```

Figure 53: Querying the WSAD link repository

Figure 53 gives a Java code snippet that shows how the link repository can be queried. The extractor iterates over all available resources in the Web project and requests the links for each one (line 4). Thus, the source of a link-relationship is always a Web application resource in the current project such as a HTML file, JSP, or Web deployment descriptor. The extractor then iterates over the returned links (line 5), obtaining relevant information to determine the type of link (lines 7–8). For hyperlinks, the extractor checks whether the link is internal (i.e., points to another resource in the project), or external (i.e., it cannot be resolved). Examples of external links are URLs that point to another Web site (using the `http` protocol),²⁴⁷ or links that contain email addresses (using the `mailto` protocol).

In order to use the extractor on a certain Web project, the user first has to associate the extractor with the project. This is accomplished by right-clicking a Web project and selecting the appropriate menu entry. Once the extractor is associated with a Web project, it extracts facts from the project's sources and stores them as an EMF model file (with `.webmodel` extension). The extractor also generates an RSF file from the EMF model, which is stored in a newly created `rsf` folder (see Package Explorer view on the left in Figure 57). Once the extractor is associated, it automatically regenerates the EMF model and RSF file whenever a resource in the Web project changes.

Eclipse Modeling Framework EMF is a modeling framework for Eclipse. It supports the declarative specification of a (data) model with UML, XML, or a Java skeleton. I

²⁴⁷The extractor does not verify the validity of external links. Thus, an external link might be actually a dead link.

discuss here only the modeling with a UML class diagram because this is the approach that I use to specify the extractor's schema. Following the approach of the OMG's Model Driven Architecture (MDA), EMF offers UML as a means for describing a model. EMF uses a set of simple modeling constructs that are easy to comprehend. Furthermore, the Java code that is generated from an EMF model is "natural and simple for Java programmers to understand" [BSM⁺03].

Because we use a modeling framework, the extractor's schema is explicitly (and externally) defined [JCD02]. EMF stores models as XML files (using the `.ecore` extension). Model instances are also stored in XML files and reference the EMF model that they adhere to. This approach makes it easy for other reverse engineering tools to parse and transform the extractor's output. For instance, one could write an XSLT script that converts the output to the GXL exchange format [HWS00].

EMF models are graphically defined with Rational Rose²⁴⁸ or Omondo's Eclipse-UML²⁴⁹. EclipseUML is an Eclipse plug-in that seamlessly integrates with EMF. I use EMF 1.1.1.1 and EclipseUML to maintain the extractor's schema. Thus, the schema is formally and graphically documented. Figure 54 shows a screenshot of EclipseUML with our Web model.

The representation of the Web model in UML is intuitive. UML classes, association, and attributes represent EMF classes, references, and attributes, respectively. UML inheritance links are used to model type relationships of EMF classes. EMF also has a number of predefined data types such as strings (`EString`), integers (`EInt`), and booleans (`EBooleanObject`).

My Web model is similar to models that have been introduced by other researchers to represent Web sites for reverse engineering. There are a number of models to represent the structure of crawled Web sites [RT01a] [MM01a] [WBM99]. Since these models represent the *client view* [KWMM03] of a site (cf. Section 6.6.3), they are mostly concerned with HTML pages and hyperlinks. In contrast, models that target the *deployment view* of a site have to also represent entities that reside on the Web server (and hence are hidden from clients) such as server pages, CGI scripts, configuration files, and database access. There are a number of models that address the deployment view [BMT04] [DLDPAC01] [HH02] [Con99]. Our model targets the developer view of WSAD J2EE applications.

A `WebSite` in my model (cf. Figure 54) consists of a number of `Pages`. A `Page` can be a file containing HTML, XML, Image data, JSP code, CSS, JavaScript, et cetera. A special kind of XML file is the `WebDeploymentDescriptor`. Depending on the kind of `Page`, it can contain `WebObjects`. A `WebObject` can be an `Anchor` (HTML files), `Form` (HTML files), embedded JavaScript code (HTML files or JSPs), or `Hyperlinks`. Links are classified into `ExternalHyperlinks` and `InternalHyperlinks`. The latter can be links to HTML files, JSPs, tag libraries, or servlets. For servlets, the extractor identified the corresponding Java class (using the API provided by the Eclipse Java Development Tools (JDT)).

²⁴⁸WSAD uses Rational Rose to define schemas for JavaBeans, Web applications, and tag library descriptors.

²⁴⁹http://www.eclipsedownload.com/download/free/eclipse_3x/index.html

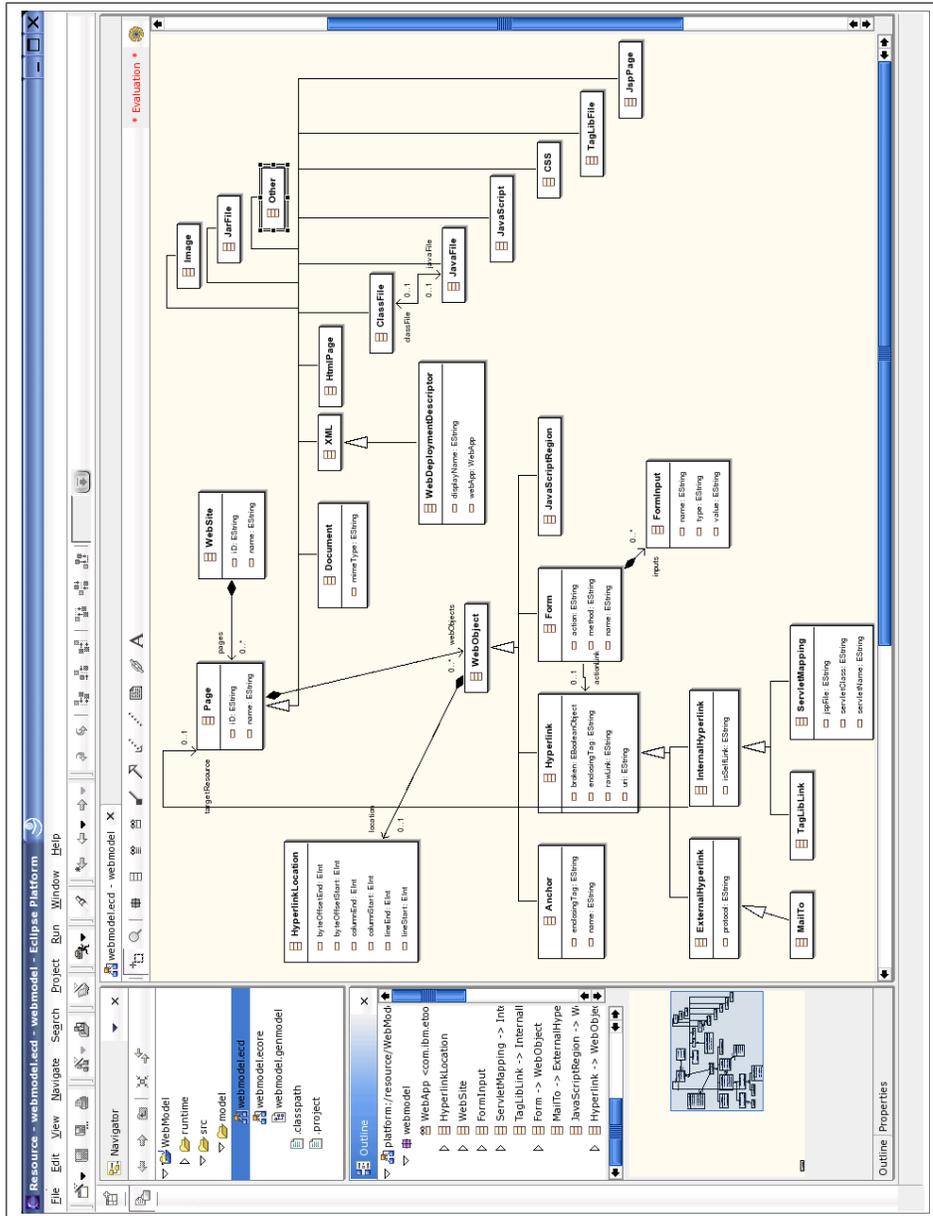


Figure 54: EMF Web model (shown with Omondo EclipseUML)

```
1  IFile file;
2  URI resourceURI =
3    URI.createPlatformResourceURI( "/a/path/out.webmodel" );
4  Resource resource =
5    new ResourceSetImpl().createResource( resourceURI );
6
7  WebmodelFactory webmodel = WebmodelFactory.eINSTANCE;
8  HtmlPage htmlpage = webmodel.createHtmlPage();
9  htmlpage.setID( file.getFullPath().toString() );
10 htmlpage.setName( file.getName() );
11
12 resource.getContents().add( htmlpage );
13 resource.save(Collections.EMPTY_MAP);
```

Figure 55: Creating an EMF model instance

The extractor stores facts that adhere to the schema defined in the UML model. EMF transforms the UML model into an EMF model (`webmodel.ecore`). From this model, EMF generates Java code: An EMF class is mapped to a Java interface and corresponding implementation class; attributes and references are translated to accessor methods; `EString` and `EInt` is mapped to `java.lang.String` and `int`; and so on. The generated Java code for our model is about 13,000 LOC. This code implements the model and provides a factory to conveniently create model instances. The EMF framework provides functionality for persistency of model instances.

Figure 55 provides a code snippet that shows how Web model instances can be created, initialized, and stored with EMF. To make model instances persistent, one has to first create a so-called resource, which is a container for model instances and is associated with a file (lines 2–5). Model instances can then be added to the resource (line 12) and written into the associated file (line 13). EMF provides a factory object to create model instance (line 7). An instance such as a `HtmlPage` can be created with the factory (line 8) and its attributes then initialized (lines 9–10).

EMF can also generate a simple table-based editor that makes it possible to interactively view and edit model instances. The top right view in Figure 57 shows the editor. The editor can be easily customized to provide different images and text for different kinds of model entities.

We have also defined a separate EMF model to configure the extractor. Currently, it is only possible to tell the extractor to ignore certain files or kinds of files. With the generated EMF editor, users can conveniently configure the extractor within Eclipse.

6.7.4 Case Studies

To test the suitability of my extractor for the implementation of reverse engineering tools, I have conducted two case studies. The first one uses the facts obtained from the extractor in an Eclipse view. The second one exports the extracted facts to RSF and uses the SHriMP

visualizer plug-in to render a graphical visualization of the facts. In the following both case studies are discussed.

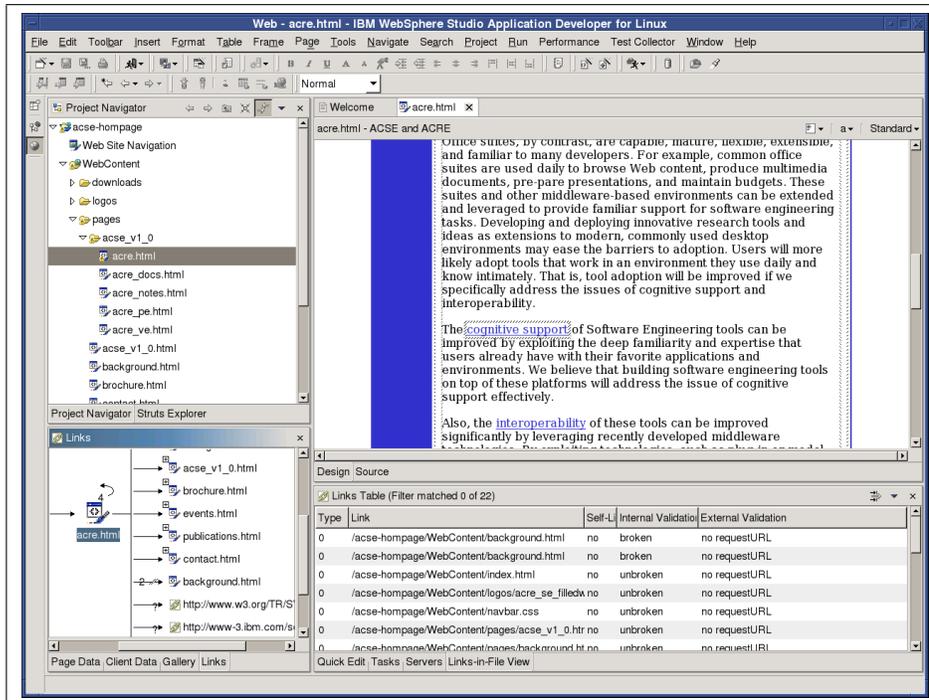


Figure 56: Broken links in the ACSE home page

For both case studies we use the ACSE home page that is available at <http://www.acse.cs.uvic.ca>. This Web page has 12 HTML pages, ten images, three PDF files, and a style sheet. The site has been developed with GoLive. We import the site as a WSAD Web project with WSAD's built-in Web site crawler. The extracted model has 26 Page instances and 197 Hyperlink instances. Besides the ACSE home page, we have also started to use a larger site, Cassis, that has been developed with WSAD by IBM Toronto CAS [KFY⁺06]. The functionality of this site is only accessible to members and researchers of CAS. The extracted model of this site has 105 JspPages, 12 Images, 141 ClassFiles, 10 JarFiles, 400 InternalHyperlinks, and 26 ExternalHyperlinks. Because of confidentiality concerns I only describe my experiences with the ACSE home page.

Links Table View The links view provides a list of all the links that are contained in a certain file. The bottom right view in Figure 56 shows a screenshot. The view is implemented as an Eclipse ViewPart that automatically refreshes its contents when the user clicks on a file in the Project Navigator (top left view). It shows the link's type in the first column ("0" means project-relative links), the link's target in the second column (relative to the project root for project-relative links), and whether WSAD's link repository has identified the link as broken or valid (fourth column).

The view also implements a filter that can suppress links by their type and target. If the user clicks on a link in the view the corresponding editor is opened and positioned at the link's location. In Figure 56, the links view shows the links for the `acre.html` page. The first two links in the view are broken and WSAD's HTML editor highlights the location of the first broken link.

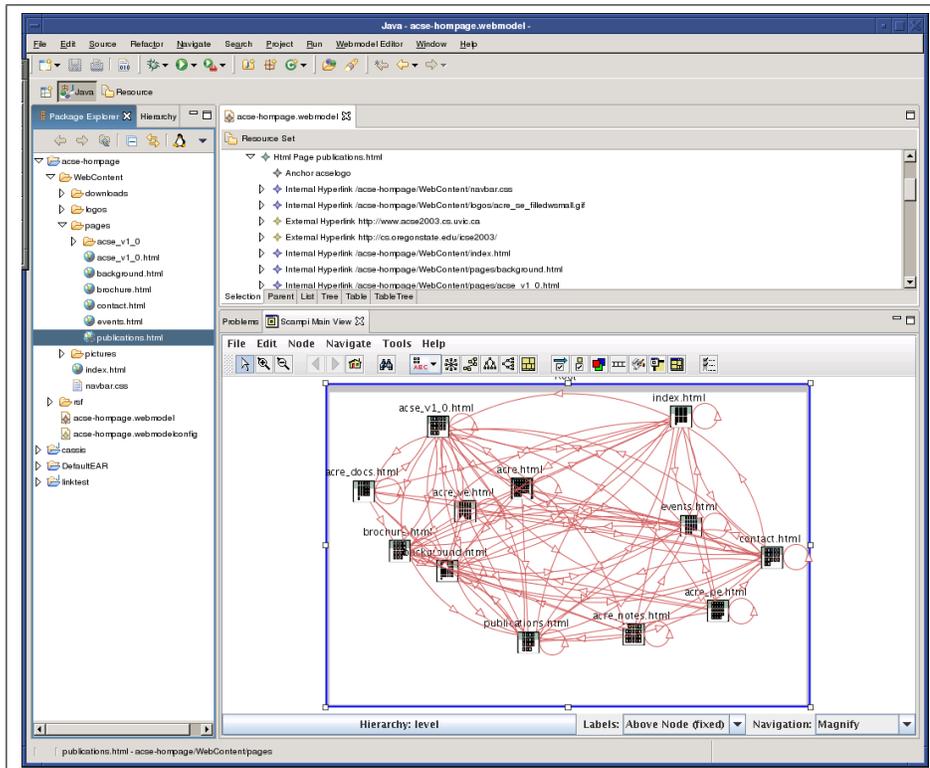


Figure 57: Reverse engineering of the ACSE home page

SHriMP Graph View This case study uses the SHriMP plug-in to visualize the link structure of a Web site as a graph. SHriMP is an interactive editor that visualizes nested graphs with node containment and animates the graph when the user navigates or opens up nodes. Graphs can be filtered by node and arc types.

Figure 57 shows SHriMP's visualization of the ACSE home page. Pages are visualized as nodes; Hyperlinks between Pages are visualized as arcs. The view is filtered to show only HtmlPages and InternalHyperlinks between the HtmlPages. The nodes have nested sub-nodes that contain WebObjects. Unfiltered graphs and graphs that expose many Hyperlinks are often too cluttered to understand. To obtain meaningful views it is necessary to apply filters and to introduce hierarchical decompositions. This is especially the case with the Cassis site that is an order of magnitude more complex than the ACSE site.

SHriMP supports the RSF and GXL exchange formats. The easiest way to integrate

my extractor with SHriMP is to export Web model instances into an RSF file [MSM01]. This approach exhibits some drawbacks. Even though both the extractor and SHriMP are visually integrated into Eclipse the tools are weakly coupled. The user of SHriMP has to manually reload the RSF file whenever it changes.

Future Work The case studies have confirmed that my extractor is suitable to serve as a front-end for other reverse engineering tools that want to analyze or visualize Web sites. The extractor produces a Web model instance without perceptible delay. WSAD parses and caches the Web project when the user works with it and as a result the extractor introduces little computational overhead. Whenever the Web project changes, the extractor has to regenerate the whole model instance (and the imported RSF file). In practice this has not caused performance problems yet. Still, I am considering to move to an *incremental builder* [Art03] that “patches” up model instances. However, this would complicate the extractor’s implementation and future maintenance.

Currently SHriMP obtains the facts from the exported RSF file. Since SHriMP’s core architecture consists of a number of JavaBean components [BSM02], it is possible to easily provide a new data source. I want to add a new “Data” Bean to SHriMP that directly reads from the extractor’s EMF model instance. This would also allow SHriMP to load facts incrementally.

6.7.5 Experiences

“When you learn Eclipse you’ll spend much more time reading code than writing code.”

– Beck and Gamma [GB04]

In this section I report on our experiences with leveraging WSAD for extractor development. WSAD is a commercial product that has matured through several versions and has a large user base. As a result, it is a stable component and I did not encounter bugs in the link repository when implementing the extractor. WSAD is a black-box component which means that its source code is not available to the public. However, I was able to get access to WSAD’s sources during a 3 month stay at the IBM Toronto Lab. Even though all Eclipse plug-ins expose extension points in so-called manifest files, source code access was crucial to realize the extractor.

The implementation of the extractor was different from my previous experiences with component-based extractor development because I could not rely on an official, documented API. For instance, when developing a Web site extractor with GoLive, we had access to 600 pages of documentation that was provided with the SDK [GKM05b].

With WSAD, the functionality that I needed was buried in the sources and spread over several plug-ins. It took a significant time to first locate the code for the link repository and to understand its workings. Since there were few comments in the code, I had to use the Java debugger to understand complex invocation sequences. I also had to experiment with the API (using small test cases) to gain a better understanding of its functionality

and meaning.²⁵⁰ Thus, significantly more time was spent understanding code than writing code. My extractor is realized in about 2,000 lines of Java code. Realizing an extractor that collects facts from such diverse sources as needed for J2EE Web application with comparable few code was achieved by leveraging WSAD's functionalities to the greatest extent possible. The resulting code is quite dense and I hope that it will be comparably easy to maintain.

Except for one documented bug that I had to work around, EMF provided a stable environment as well. EMF has spared me tedious code writing that would have been otherwise required to realize a persistent data model for the extractor. The generated model editor was useful as a browser to inspect and debug the generated facts. I also used the editor to construct small test cases for the RSF export. Furthermore, EMF encourages an iterative development style with rapid model changes. Model changes are confined to editing the UML model and regenerating the code. This was invaluable because the model had to be changed and augmented often, especially in the early development phases where I did lack a thorough understanding of the domain of J2EE Web applications.

6.7.6 Conclusions

I have described the construction of a component-based extractor for the reverse engineering of J2EE Web applications. The extractor leverages the functionalities offered by WSAD and EMF. As a result, it was possible to rapidly construct an extractor with little code that obtains facts from highly heterogeneous Web applications, which use technologies such as HTML, JSP, and servlets. The implementation of the extractor was complicated by the fact that WSAD's internal interfaces are complex and largely undocumented. I have also conducted two case studies that use my extractor to generate visualizations for a sample Web site. These case studies confirmed that the extractor is sufficiently efficient, stable, and complete to provide facts for reverse engineering analyses and visualizations.

The idea to build reverse engineering tools from reusable parts is not new and many researchers have followed this approach. For example, it is quite common to build extractors for high-level programming languages on top of compiler front-ends. However, to my knowledge, there are very few experiences with component-based Web site extractors that leverage commercial Web authoring tools. My extractor addresses this gap. My experiences with building a component-based Web site extractor using WSAD are mostly positive. I encourage other researchers to follow this approach to tool-building, and to report on their experiences.

6.8 Summary

This chapter discussed my own tool-building experiences, which have been conducted under the umbrella of the ACRE project. Six different tools have been constructed, demonstrating a broad spectrum of leveraged technologies. I have gained experiences with

²⁵⁰For example, what constitutes a "self link" in the link repository was not evident.

four distinct host products (i.e., Microsoft Office, Lotus Notes, Adobe GoLive, and IBM WSAD), and a W3C standard (i.e., SVG, rendered with Adobe's viewer). The functionality of the implemented tools mostly relied on programmatic component customization via scripting in Visual Basic or JavaScript, and programming in Java.

The characteristic of my tools all fall within the target design space outlined in Section 5.1. Most notably, the target design space assumes a component of external origin that can be customized programmatically and exposes an interoperability interface. A tool is constructed by first selecting a single component in the form of an OTS product, and then customizing it to realize the tool's functionality. Besides my tools, there are tools that have been developed by other researchers that either fall within the design space or come close to it. Three examples of such tools (i.e., VDE, Desert, and Galileo) have been presented in detail in Section 5.3.

Both this and the previous chapters constitute rich sources of tool-building experiences. Many of these experiences fall within the target design space and are directly applicable to my proposed tool-building approach. In the next chapter I use these experiences as a knowledge base to distill guidelines for component-based tool-building in the form of recommendations and lessons learned.

7 Recommendations and Lessons Learned

This chapter discusses recommendations and lessons learned for my component-based approach to build reverse engineering tools. The approach is scoped by tool requirements (cf. Section 3.2) and the target design space (cf. Section 5.1), which codify the characteristics of the realized tool and its host components.

I assess the ramifications of my tool-building approach with two stakeholders: tool user and tool developer. Both stakeholders can be seen as representing idealized roles in the tool development process. The *tool user* is the end user of the system, influencing the tool's requirements, assessing its usefulness and usability, and making an adoption decision.²⁵¹ The *tool developer* is responsible to realize the system. Typically, the tool developer is a single researcher or a research group building the tool.²⁵²

I make the following assumptions about the goals of the tool user and tool developer:

tool user: The tool user wants a tool that meets his or her requirements.

tool developer: The tool developer wants an effective approach that enables him or her to construct a tool that meets the tool user's requirements.

The stakeholders' goals are addressed with two perspectives: tool and process. The tool perspective uses the output of my tool-building approach—the realized case studies discussed in Chapter 6—to assess whether it has the potential to produce tools that both satisfy the tool user's requirements, and whether they are feasible to build for the tool developer. Part of an effective tool-building approach is a suitable process that gives guidance and structure for the tool developer and tool user. Lack of a process makes it more difficult for the tool developer to build tools in a repeatable and predictable way. The process perspective describes a framework for defining an effective process for the tool developer.

The tool and process perspectives are addressed in Section 7.1 and Section 7.2, respectively.

7.1 Tool Perspective

In this section, I evaluate the effectiveness and feasibility of my tool-building approach by looking at the approach's products—namely, the software engineering tools that have been realized with it. The tools that I need as input to conduct the evaluation are mainly provided by my own tool-building case studies (cf. Chapter 6), but stem also from applicable experiences reported by other researchers. When assessing the case studies, I distill general observations that hold across the case studies into lessons learned.

²⁵¹There may be other relevant stakeholders besides the tool user such as the customer (i.e., the entity that pays for the development or purchase of a system) [BCK98, p. 8]. A tool user can also take the role of a customer representative.

²⁵²Besides implementation work, the tool developer may fill other roles such as project manager, domain expert, architect, designer, tester, evaluator, and technical writer [Cen97, sec. 1.3].

Following my assumption of the tool user's and tool developer's goals, these case studies and experiences are qualitatively assessed with the tool requirements obtained by my requirements survey (cf. Section 3.2). The survey's requirements are objective in the sense that they are based on a comprehensive literature review of the reverse engineering domain.

Other researchers have also used tool requirements to evaluate the effectiveness of their tools or approaches. In his dissertation, Wong proposes the concept of a Reverse Engineering Notebook [Won99]. Instead of fully implementing the Notebook, he discusses three case studies that realize aspects of the Notebook (i.e., SQL/DS, RevEngE, and the Software Bookshelf). These case studies are evaluated with 23 general and 13 data requirements for reverse engineering tools. For each requirement, Wong indicates whether it is met "mostly" or to a "small extent" [Won99, sec. 6.6]. Wong also describes lessons learned from his case studies; in fact, he states that "a major theme of the dissertation is learning from the successes and failures of studies" [Won99, p. iii]. Similar to Tichelaar, Wong's requirements are subjectively derived from his own experiences. There is no discussion why he believes that the requirements are met mostly or only to a small extent.

Related evaluations

Tichelaar follows a requirements-driven approach in his dissertation to evaluate the Moose tool [Tic01]. He first states six requirements for reengineering environments [Tic01, sec. 5.1], and then briefly assesses his tool with them [Tic01, sec. 5.9.2]. However, Tichelaar's requirements are subjective in the sense that they are almost exclusively based on his own experiences. He then uses his requirements to evaluate a single tool only; thus, he is not able to synthesize lessons learned.

In the following, various experiences are distilled into lessons learned for my proposed tool-building approach.²⁵³ Sections 7.1.1–7.1.5 address whether the resulting tools can meet the survey's quality attributes: scalability, interoperability, customizability, usability, and adoptability. Note that these sections are mirroring Sections 3.2.1–3.2.5.

7.1.1 Scalability

"Galileo's response time is largely dependent on package performance. ... We spent significant effort understanding Visio's drawing speed and exploring ways to use Visio that would give us better drawing performance."
– Copenhafer and Sullivan [CS99]

The constructed tool should be able to fulfill the tool user's scalability expectations. If the visualizer is realized with a host component, it is important to explore scalability issues early on. Generally, components are able to handle the rendering of dozens of graphical objects. This is sufficient for small to medium software graphs. However, the rendering of larger graphs can cause problems in terms of screen updating and rendering speed.

Visualizer scalability

SVG is able to visualize large graphs. However, the scalability of SVG depends on the actual SVG visualizer. Furthermore, embedding of SVG into Microsoft Office documents can change the rendering performance. Interactive manipulation of graphical objects can

²⁵³Some lessons learned draw from material that has been previously published in a paper for *ACSE 2004* [KLM04].

cause performance problems. To implement drag-and-drop of nodes in the SVG graph editor efficiently, implementation tricks were necessary to achieve a smooth interaction.

Microsoft Office products are able to render small to medium sized graphs, but should not be expected to scale up to large graphs. The rendering speed of graphical objects is typically insufficiently fast to allow interactive work with large graphs. As my experiences with REOffice and REVisio show, a graph with a thousand nodes can take several minutes to render in PowerPoint and Visio. The Galileo tool made similar experiences with Visio [CS99]. However, different host components and different versions of the same host components can differ in their performance characteristics. For example, in Visio 4.1 the drawing speed of objects increases quadratically with the number of shapes already present on the page; in contrast, Visio 5.0's behavior is linear [CS99]. Furthermore, the drawing speed of Visio 5.0 can be improved by disabling screen updating while making bulk changes to the drawing.

Lesson Learned 1: Benchmark the scalability of the visualization functionality early on (e.g., via an exploratory prototype) before committing to a certain host component.

A scalability benchmark can be conducted quickly and cheaply. For example, to assess a component's rendering speed of a software graph, it is sufficient to implement a simple prototype that renders an appropriate amount of graphical objects (e.g., lines and boxes). Similarly, the manipulation of a graph (e.g., applying of a layout algorithm) can be simulated by moving every graphical object to a different location.

Extractor components can have scalability issues in terms of the time it takes them to produce the facts for a software system. The performance of extractors is less of a concern if they operate batch-style and are not interactive. Generally, extractors that are based on compiler front-ends and compiler technology can be expected to have performance characteristics comparable to compilers, which means that they meet users' scalability expectation. The tools in my case studies deal with coarser-grained facts, which means that comparably less data needs to be transferred and stored. As a result, I did not experience extractor scalability problems. The REGoLive and WSAD extractors use the host component's API to extract facts. This approach does not result in noticeable delays for the tool user. The Rigi C++ extractor uses VisualAge C++'s API to extract facts as a side effect of the compilation process; this also does not negatively impact the compilation performance.

Extractor scalability

Lesson Learned 2: Coarser-grained extractors based on host components do not cause scalability problems.

Since coarse-grained extractors produce comparably few facts, repository scalability is unlikely to be an issue as well.

7.1.2 Interoperability

“One of the thorniest implementation problems is the problem of integrating software components that were not designed with integration in mind.”

– Steve McConnell [McC97]

The interoperability between host components can be classified into three approaches: data, control, and presentation integration (cf. Section 4.2.4). From the tool user’s perspective, presentation integration is often desirable because it enables a seamless transition between the functionalities offered by different host components. However, presentation integration is only feasible if there is a common wiring standard for component integration. The WSAD extractor and its associated visualizers are all Eclipse plug-ins, which allows them to seamlessly extend Eclipse’s GUI. The Galileo tool relies on the Microsoft Active Document standard to integrate Word, Visio, and Internet Explorer into a single top-level window. Wrappers and bridges can be used to integrate heterogeneous components; however, this solution can turn out to be brittle. The SHriMP tool is using an AWT-SWT bridge [SDF⁺03] to integrate with Eclipse, but this approach has several undesirable effects (e.g., risk of deadlocks and lost keyboard events) [RLS⁺03].

Presentation
integration

Lesson Learned 3: Presentation integration is only feasible if the host components support the same wiring standard.

In contrast, data integration that is based on the reading/writing of information from/to files is often easy to implement, understand, debug, and maintain. REOffice, the SVG graph editor, REVisio, and RENotes all support data interoperability with certain exchange formats (e.g., RVG, RSF, and GXL). With this approach, the visualizer host components needs to be customized with a reader for the exchange format (implemented in a scripting language such as Visual Basic). The use of exchange formats is a proven interoperability technique in the reverse engineering domain, and equally applicable for component-based tool-building. Alternatively, the extractor can also produce a file in a format that is native to the visualizer. For example, an extractor can produce Microsoft Office XML files.

Data integration

Lesson Learned 4: File-based data interoperability is very effective for integrating the extractor and visualizer host components.

File-based interoperability can be used to realize rudimentary control integration. One component can poll a file for some change (e.g., file existence or modification) caused by a different component [Del97]. Once a change is detected, the component can read the file and interpret its contents. This approach has been successfully used with Rigi and Microsoft Office products to achieve interoperability [Yan03, sec. 5.3].

Control integration

Interoperability of RENotes’ visualizer with Lotus Notes is based on Java. The visualizer is launched from Lotus Notes with a so-called Java Agent. The Swing-based visualizer then uses Lotus Notes’ Java API to access the database that contains the graph model. GoLive realizes control integration using Web services, passing messages between

the GoLive and SVG visualizer components. The RevEngE tool realizes control integration between Rigi and REFINE using the Telos repository and a message bus [Won99, sec. 6.3]. Solutions for control integration are often ad hoc, brittle, and based on experimentation. Different host components often require the use of quite different mechanisms to achieve control integration. Advanced integration mechanisms can be difficult “to implement and keep working” if the host components are updated [Won99, sec. 6.3].

7.1.3 Customizability

The ability to customize host components is a necessary prerequisite for my tool-building approach. Fortunately, many (OTS) products support customizability via scripting or API programming.

If a component offers both an API and scripting, often the latter is preferred. For example, tools that leverage Microsoft Office often rely on Visual Basic instead of using another COM-aware language such as C++. This is the case, for instance, with REOffice and REVisio, as well as VDE and Galileo. The SVG graph editor, implemented in JavaScript, shows that scripting languages can be used to write applications with thousands of lines of code in a disciplined manner and using design patterns. Scripting of customizations eases experimentation and allows rapid prototyping of functionality. In a research environment, these benefits outweigh potential drawbacks of scripting such as lack of a strong type system, and worse maintainability.

Leveraging scripting

If the component is carefully selected, its existing functionality can cover a significant part of the tool functionality. As a result, significantly less code needs to be written and subsequently maintained. Reiss reports that for Desert’s editor, “FrameMaker provides many of the baseline features we needed” [Rei99]. The Rigi C++ extractor was developed rapidly with VisualAge C++. As a result, it was possible to “develop the parser much faster than if a parser had been written from scratch” [Mar99]. The parser has under 1,000 lines of C++. In contrast, the hand-coded Rigi C extractor has more than 4,000 lines of rather complex Lex, Yacc, and C++ code. We made similar experiences when implementing RENotes; the JGraph component was used to implement the graph visualization. This implementation has about 4,000 lines of Java code. In contrast, Rigi has about 30,000 lines of C/C++ code.

Leveraging existing
functionality

Researchers also report beneficial experiences in leveraging XML and XMI for data integration [MSM01] [RLS⁺03]. For example, Rayside et al. state, “leveraging industrial tool infrastructure can significantly reduce the amount of effort required to develop tool prototypes. In our case, the MOF frameworks essentially eliminated most of the mechanics of the data integration work, and let us focus on more interesting problems” [RLS⁺03].

All of my case studies benefit from a significant amount of functionality that is provided by their host components. OTS products provide a GUI along with many objects that can be programmatically created and manipulated. REOffice, REVisio, and the SVG graph editor mostly leverage the drawing capabilities of their host products; RENotes leverages textual representations of relational data and the underlying repository. The WSAD

extractor leverages various plug-ins to extract information and visualize it textually and graphically.

Lesson Learned 5: Host components enable rapid prototyping of tool functionality by leveraging existing functionality and scripting languages.

Besides customization of host components, scripting languages are also beneficial to prototype interoperability mechanisms with other host components. Ousterhout claims that “for gluing and system integration, applications can be developed five to ten times faster with a scripting language” [Ous98].

Customizations of components is limited by the functionality that the API and/or scripting language provides. As a consequence, customizations of host components without the use of source code modifications always run the risk that certain desirable tool functionalities cannot be realized at all, or with less fidelity. Unfortunately, missing functionality is hard to identify up-front. Often, limitations materialize unexpectedly after a commitment for a certain component has been finalized and substantial development effort has been already invested. When implementing REVisio, it turned out that some pre-defined masters did pop up user dialogs. The values requested by the dialogs could not be given programmatically. In the end, it was necessary to re-implement most of the master’s functionality with low-level operations.

Customization
limitations

Lanza cautions that the use of visualization host components such as AT&T graphviz can cause “problems like lack of control, interactivity and customizability” [Lan03]. Implementing graph visualization functionality in my case studies revealed customization shortcomings in the customizability of host components. In SVG, the interactive manipulation of graphical objects needs to be implemented in JavaScript with low-level modifications of the underlying DOM. The resulting code is difficult to understand and potentially brittle with respect to changes in the environment. Lotus Notes has two types of textual representations (Form and View), but does not provide native graphical objects that could be used to represent graphs. GoLive has support for graphical objects, but it is too restricted to implement an interactive graph editor effectively. The deficiencies in GoLive’s customizability were detected relatively late in the development process and were quite unexpected. Other researchers also point out the lack of support for customizability. Based on their customization experiences with several commercial components, Egyed and Balzer say that “sadly, ‘real world’ COTS tools are often only partially accessible and customizable, greatly limiting their reuse” [EB01]. Galileo and VDE had to cope with the lack of event notifications of Microsoft Office products. VDE’s developers complain that “detecting events initiated through the native PowerPoint GUI was a serious problem. Although a COM interface *could* make relevant events available, the interface implemented by PowerPoint97 *does not*” [GB99]. Galileo’s developers had to cope with undocumented restrictions of Microsoft Office components: “Due to [the] lack of package documentation, we discovered certain limitations of the packages only after working with them extensively. In some cases, the limitations were quite serious” [CS99]. Reiss reports the following limitation

in the FrameMaker API: “While FrameMaker notified the API that a command was over, it did not provide any information about what the command did or what was changed” [Rei95b]. Gray and Welland note that Rational Rose cannot be customized to change the line width of inheritance links in UML diagrams [GW99, sec. 4.4].

Lesson Learned 6: Limited customizability of host components often cause problems and require unexpected workarounds.

Using components can drastically reduce the development effort when building new tools. However, to customize a component effectively, a significant learning effort is often required. This is especially the case if the component offers sophisticated programmatic customization based on an API or framework. For example, tool developers that wish to customize VisualAge C++ should be aware that “regardless of how good the programming interfaces are, [it] is a big system, and there is a steep learning curve” [Kar98]. Part of the difficulty of learning the API is explained by the complexity of the C++ language [Mar99]. Even though the available extensions in Montana are a fixed set, it is a large number. The preprocessor, for example, has about 50 event notifications. Furthermore, there are three different extension mechanisms. For Eclipse and WSAD, the number of extension points is even larger. In fact, since any plug-in can define extensions, the number of useful extensions for a tool-builder is open-ended. Not surprisingly, Eclipse developers report that “the steep learning curve for plug-in developers in Eclipse is hard to master” [BBZ03]. Eclipse promises significant leverage for tool builders, but at the likely expense of a significant learning curve. Generally, tool developers need to be careful not to be overly optimistic about the required effort to master a component [BPY⁺03b, Risk 14].

Customization
learning curve

It improves the learning curve if a component can be explored via scripting, because the tool builder can more easily experiment and explore. The learning curve can be also significantly flattened if supporting documentation is available that gives concrete customization examples. Studies have shown that many users customize by adapting existing sample code [GN92]. Code samples that show how a host component’s API is used most effectively may be available with cookbooks, (online) articles, discussion forums, and so on. An active user base can be invaluable in providing support to resolve questions about use and customization of a host tool. For example, Vidger et al. say that “most of the COTS components we integrated had a large user base. Examples and workarounds built up by this user base were invaluable for solving many problems we encountered” [VGD96]. Other researches have made similar observations [HTT97]. The developers of VisualAge C++ propose a book of idioms to make tool builders more productive with the idea that “once the idioms have been learned, tool writers can be very productive” [Kar98]. Adobe maintains the FrameMaker Developer Knowledgebase, which has some typical examples for how to customize FrameMaker via its C API. Jahnke et al. report that the customization of UMLStudio was easily accomplished because the product ships with a collection of sample scripts. As a result, “it took us approximately four days of work to develop the required GXL export/import functionality by ‘customizing’ these scripts” [JWZ02]. Customization

of Eclipse is typically done by adaptation of existing code: “Often, if the programmer wants to accomplish something, she has to search the Eclipse source code for that functionality, copy it and customize it to suit her needs” [BBZ03].

All of my tool development efforts did leverage existing code samples in one form or another. With Microsoft Office, the macro recorder can be effectively used to first generate and then inspect Visual Basic code. Whereas the generated code is often awkward, it is very useful to rapidly learn how to call the API in order to accomplish certain tasks. When implementing the SVG graph editor, I adapted code samples that showed how to access and manipulate objects in the DOM to get started. Furthermore, I reverse engineered SVG sample applications from the Adobe SVG Zone. Even though Lotus Notes is extensively customized for industrial applications, it was difficult to find good sample code. A free book from the IBM Redbooks series was useful to get an understanding of LotusScript and Java Agents [TCG⁺02]. It was also surprisingly hard to find good sample code for customizing Microsoft Office products [WKM02]. To implement the WSAD extractor, I extensively copied and adapted code snippets from Eclipse and WSAD—“Monkey See/Monkey Do” [GB04, p. 40].

Lesson Learned 7: Knowledge in the form of existing customizations reduces the tool developer’s customization effort and learning curve.

7.1.4 Usability

The usability of a tool depends on many factors. From the perspective of the tool user, fewer heterogeneous host components can increase the likelihood that the tool is more usable. Franch and Maiden say

Tool uniformity

“users can use a system more easily when they interact with fewer different COTS components with different communication styles and interfaces. A few dependencies with a high number of interacting components will lessen the usability of the system” [FM03].

A single host component promotes uniformity. This is the case with the WSAD extractor. REOffice, REVisio, and the SVG graph editor use a single host component for visualization. The graph data is read from file, produced by some extractor. This approach is easy to understand for the user, and decouples the task of graph generation from graph inspection and manipulation. RENotes and GoLive both require the user to switch between two nonuniform host components. This increases complexity and negatively affects usability.

Sullivan makes the important observation that

Tool familiarity

“requirements shared by many tools include demands for user interfaces that support graphics, windows, icons, and mouse-based interaction; functions for editing viewing, printing, annotation and storage of technical drawings, capabilities for general data management; and for advanced editing of textual

representations. . . . these key requirements appear to be well matched to the capabilities of existing application packages.” [Sul97].

The functionalities that are needed for a tool but already provided by its host components constitute the tool’s *baseline functionality*.²⁵⁴ This baseline functionality reduces the tool developer’s customization effort, but equally important, increases the tool user’s familiarity. Usability of the tool is also enhanced if the user is already familiar with the host component (or its interaction style). For example, a graph visualizer that leverages the host component’s graphical objects enables the user to use existing knowledge about how to select, copy, paste, rotate, delete, connect, print, and otherwise manipulate graphical objects. This is the case for REOffice and REVisio as well as VDE and Galileo. Depending on the host component, other baseline functionalities can be leveraged as well. For example, users who are familiar with PowerPoint can use slides to organize and manipulate graphs. They can easily copy graphs by copying slides, reorder graphs by reordering slides, save graphs by saving slides, annotate graphs by editing slides, and so on. Users who are familiar with Lotus Notes, can readily leverage its group support to discuss graph visualization with other researchers. The developers of Galileo report about the following experiences of users: “When asked if their familiarity with the packages helped them use Galileo, nearly all said it helped at least a little, and several said it helped a lot” [CS03]. The ETHS hypertext system leverages the baseline functionality of Emacs, augmenting it via Emacs Lisp scripting with special hypertext editor commands. ETHS uses Emacs “because it is the single most used editor in our department . . . users familiar with GNU Emacs only have to learn a dozen new editor commands instead of learning to use a whole new editor in order to use the hypertext system” [Wii92, sec. 3.3].

Baseline functionality can be divided into *horizontal* and *vertical* functionalities. Horizontal functionality are applicable over a range of application domains such as text editing and drawing capabilities. Vertical functionality builds on horizontal functionality, but is specialized for a certain domain such as text editing for report creation, or source code editing for programming. Similarly, drawing capabilities are useful in the domains of CAD, presentations, and software diagrams. For example, the Desert editor uses the existing horizontal baseline functionality of text editing provided by FrameMaker to build additional horizontal functionality for source code editing on top. Reiss says,

“Because it was designed to be a document preparation system, FrameMaker provides an effective starting point for our prototype. Many of the basic features necessary for document creation and editing are provided, allowing effort to be concentrated on more specialized features” [PFK96].

Some host components already provide (rudimentary) functionality for reverse engineering, further increasing the tool user’s familiarity. For example, many IDEs already have class browsers and class hierarchy views, which provide an aid for program comprehension.

²⁵⁴Coppit and Sullivan have dubbed this the tool’s superstructure [CS00b].

Rational Rose can reverse engineer the APIs of DLLs and Jar files [PT02]. The reverse engineering capabilities of GoLive have been discussed in Section 6.6.2.

Lesson Learned 8: Leveraging host components that are uniform and familiar to the tool user benefits usability.

Tool users' familiarity with a component can be exploited most effectively if the tool's extra functionality is orthogonal to the host components' baseline functionality. This means, that the additional features should not modify or disable existing functionality that the user already knows and expects to work. For example, tool-specific (graphical) objects should not disable common operations such as copy, paste, and undo.

7.1.5 Adoptability

Tool adoption depends on many factors. Besides technical considerations such as tool features, which typically dominate the discussion of tool adoption, there are also many other issues that affect a tool user's adoption decision. In the following, I give a few examples. Often tool developers pay scant attention to documentation even though it decreases complexity and helps trialability for the tool user. The use of host components can simplify the production of documentation for the developers on the one hand, and enhance usability of documentation for the user on the other hand. For example, Microsoft Office's Assistant can be customized in Visual Basic to provide tool-specific help. The Eclipse help system makes it easy to make HTML-based information available to users. Popular host components can reduce the need for documentation: "Because such components are also popular stand-alone applications, users are often already familiar with them, and much of the application documentation applies when the application is used as a component" [CS00a]. The market share of host components can also improve a tool's adoptability. For example, the popularity of PowerPoint was an important factor for the developers of VDE: "Visio is a commercial product with many similarities to PowerPoint, provides extensive visibility into its state-change events. It might provide a better technical fit to our needs, but lacks PowerPoint's enormous user base" [GB99]. Familiarity of the target users with a certain host product improves usability and helps adoptability. Reiss explains the decision to use FrameMaker as a host component with the fact that "we wanted an editor that programmers would actually use. This meant that the base editor must be familiar to them, preferably one they were using already" [Rei99].

Designing a tool for customizability has a beneficial impact on adoptability. Customizability of a tool helps adoption because it allows users to tailor the tool for their individual needs. Customization is a form of end-user development, which has been shown to be beneficial for user satisfaction [McG04]. Fischer and Girgensohn state that "end-user modifiability is not a luxury, but a necessity in cases where the systems do not fit a particular task" [FG90]. However, the tool has to deliver observable benefits to its user rapidly and with no or little customization almost instantaneous. Tools that require extensive customization

as an up-front investment by the tool user to get productive are jeopardizing their adoption. Experiences reported by Gorton and Zhu with the ARMIN tool illustrate this problem [GZ05]:

“ARMIN provides extremely flexible abstraction and visualization models, but does not support out of the box visualizations. ARMIN’s scripting language is powerful enough to produce virtually any required views, but it takes additional effort to create the scripts. Although this may not be a problem for research purposes, it is likely to be an inhibiting factor for wider industrial use. A suggestion would be to pre-package useful scripts or even replace certain scripts with custom GUIs for generating common architectural views.”

Lesson Learned 9: Adoptability needs to address issues that go beyond purely functional or technical aspects.

Research tools are often developed without explicitly considering their adoption. Attempting to get a certain research tool adopted in industry is often an afterthought of its developers. Even if researchers are concerned about tool adoption, they often design for an abstract user and not a real industrial client. All of my case studies exhibit this shortcoming. The case studies are proof-of-concept prototypes that realize well-established reverse engineering concepts and functionalities, but they lack feedback from industrial users.

Building the tool first and then throwing it at developers with the idea that they start using it does not work. If researchers are serious about tool adoption, they have to address it during the entire life cycle of tool development. This means, that at the beginning suitable industrial partners need to be found that are willing to invest time and resources. Conversely, researchers have to be willing to understand the needs of industrial tool users. A long-term commitment of both the tool builder and the industrial partner is necessary—encompassing joint tool design and evaluation—to foster tool adoption. There are few examples of researchers who have closely collaborated with industry when building a tool, and who have published their experiences. The developers of TkSee made the effort to observe real software developers in industry when designing their tool. Feedback from real users was obtained to further improve and evaluate the tool. Similarly, the developers of Galileo did collaborate with NASA’s Langley Research Center to make their tool practical for real-world tasks.

Lesson Learned 10: Adoptability needs to be addressed during the tool’s entire development life cycle.

7.1.6 Discussion

The tool perspective distills experiences as a number of lessons learned. Table 12 summarizes these lessons and identifies the relevant case studies that provide support for them. Case studies (1)–(7) correspond to the tools discussed in Sections 6.2–6.7. I believe that the

lessons learned provide valuable advice for researchers who want to follow a component-based tool-building approach.

Lesson learned	Case studies						
	(1)	(2)	(3)	(4)	(5)	(6)	(7)
1. Benchmark the scalability of visualizer functionality.	•	•	•				•
2. Coarser-grained extractors do not cause scalability problems.	•	•	•	•	•	•	•
3. Presentation integration has to leverage a wiring standard.				•	•	•	•
4. File-based data interoperability is very effective.	•	•	•	•	•		•
5. Host components enable rapid prototyping of tool functionality.	•	•	•	•	•	•	•
6. Limited customizability of host components often causes problems.			•	•	•		•
7. Customization samples reduce development effort and learning curve.	•	•	•	•	•	•	•
8. Host components that are uniform and familiar benefit usability.	•		•	•	•	•	•
9. Adoptability must address also non-technical aspects.	•	•	•	•	•	•	
10. Adoptability is a concern during the whole life cycle.	•	•	•	•	•	•	

Case studies: (1) REOffice, (2) SVG graph editor, (3) REVisio, (4) RENotes, (5) REGo-Live, (6) WSAD extractor, (7) others

Table 12: Summary of lessons learned and supporting case studies

The applicability of a lesson learned can be assessed in the light of the supporting case studies. All lessons learned are supported by case studies. Importantly, I also found supporting experiences from researchers for most lessons learned. For the CeBASE repository, Rus et al. classify lessons learned into hypothesis, observation, good practice, and bad practice [RLSB02]. Lesson 2 can be seen as good practice that is widely applied. The lessons about scalability, interoperability, customizability, and usability classify as observations that are supported by concrete experiences. The lesson about usability is a hypothesis that should be confirmed with user studies. The lessons about adoptability can be seen as hypothesis because they are not backed up with formal experiments and empirical data; they represent my beliefs after several years of tool-building and research in reverse engineering.

7.2 Process Perspective

“We do not promote RUP (Rational Unified Process) for all projects, CMM (Capability Maturity Model) level 5 for all organizations, XP (Extreme Programming) for all teams, or object-orientation for all applications. Each problem, organization, and project has its own characteristics, requiring a range of techniques and strategies—but never none!”

– Laplante and Neill [LN04]

All software development projects should use a well-defined process. Tool-building in an academic environment is no exception to this rule. According to Kruchten, without such a process the “development team will develop in an ad hoc manner, with successes relying on the heroic efforts of a few dedicated individual contributors. This is not a sustainable condition” [Kru99, p. 15]. A process should provide guidance on what work products should be produced when, how, and by whom. A well-defined process enables a repeatable and predictable way of building software systems.

Need for a process

To meet the tool developer’s goal of an effective tool-building approach, it is necessary to provide him or her with a suitable development process. This process needs to provide the necessary guidelines on how to build tools that meet the tool user’s requirements in a repeatable and predictable way. This section shows that it is feasible to define a suitable process for my tool-building approach and identifies its characteristics (in term of process requirements and work products).

Approach

A suitable process for my tool-building approach needs to accommodate the approach’s inherent constraints. These *process constraints* are as follows:

Process constraints

reverse engineering domain: The tools are developed for the domain of reverse engineering within an academic research environment. Thus, it has to meet the process requirements identified in my survey (cf. Section 3.3).

component-based software engineering: The realized tools have the characteristics of component-based systems. Thus, the process should accommodate CBSE (cf. Section 4.1.6). Specifically, the process has to incorporate the typical stages of a CBSE process.

target design space: The resulting tool’s characteristics have to fall within the target design space (cf. Section 5.1).

Importantly, each individual tool-building project has its own unique characteristics such as the tool’s requirements and functionality, the selected host products, the degree of technical uncertainty, the complexity and size of the problem, the number of developers, the background of the development team, and so on. This is especially the case for tool-building projects in academia, who can differ significantly from each other. Can a single process be proposed under these circumstances? IBM’s Object-Oriented Technology Center (IBM OOTC) cautions:

Need for a tailorable process

“We need to analyze the software development problem before we design development approaches” [Cen97, p. 14]. “No one process is the right one for all projects. Radically different projects require different development processes” [Cen97, p. 32].

Similarly, Cameron argues for a process that can be tailored to a particular software development project [Cam02]:

“The diversity of IT projects frustrates any direct attempt to systematize the processes used for their development. One size just won’t fit all. . . . A more flexible and configurable approach to process guidance is needed, a way of tailoring the process to the needs of each particular project.”

Examples of such tailorable (a.k.a. customizable or configurable) processes are RUP, IBM Global Services’ process framework, Cockburn’s Crystal methodology family, and Extreme Programming [Kee04] [Cam02].

Vessey and Glass believe that approaches to systems development should be strong (i.e., domain specific) as opposed to weak (i.e., domain independent) [VG98] (cf. Section 3.3.5). An example of a strong development approach is Extreme Researching (XR), which is a process specifically designed for applied research and development in a distributed environment that has to cope with changing human resources and rapid prototyping [CCM05]. It is an adaptation of Extreme Programming and has been developed by Ericsson Applied Research Laboratories to support distributed telecommunications research.

7.2.1 Process Framework

“The most useful form of a process description will be in terms of work products”

– Parnas and Clements [PC86]

The dilemma with proposing a process is as follows: On the one hand, we need a development process for tool building; on the other hand, the individual projects seem too diverse to be accommodated by a single, one-size-fits-all process. Furthermore, even if there was a single process being able to satisfy all the diverse tool-building projects, it is not reasonable to expect that researchers would readily abandon their own approach and adopt a new one. Since it is not practical to define and mandate a full process, I propose a *process framework* (or skeleton) instead.²⁵⁵ This framework addresses the constraints of my tool-building approach, but needs to be instantiated by researchers to account for the unique characteristics of their own development projects. The process framework is composed of a set of work products. A *work product* is “any planned, concrete result of the development process” [Cen97, p. 623].²⁵⁶ The process framework’s focus on work products is inspired

²⁵⁵An approach that provides guidance on the creation of a new process is called process framework or process construction kit [Coc03, p. 11].

²⁵⁶Different processes have slightly different notions of work product, artifact, et cetera. This process framework

by IBM OOTC's process, which is described in the book *Developing Object-Oriented Software: An Experience-Based Approach* [Cen97]. It focuses on work products because often there is agreement on what should be produced, but not on how it should be produced. The developers of the process say, "it was decided that an approach that standardized on work products but allowed for individual choices on particular techniques used to produce work products was the best fit" [Cen97, p. 4]. As a result, concrete work products provide necessary guidance for tool-builders without unnecessarily constraining them. The IBM OOTC process can be characterized as *work product oriented*.²⁵⁷ The viability of this approach has been demonstrated by a large number of mentoring engagements and projects inside IBM. It is incorporated as the object-oriented portion of the IBM Worldwide Solution Design and Delivery Method, and has evolved into the IBM Global Services Method [Cam02] [GAB⁺03, ch. 2]. OOTC and the Global Services Method have been used in a number of different domain as well as for forward and reverse engineering.

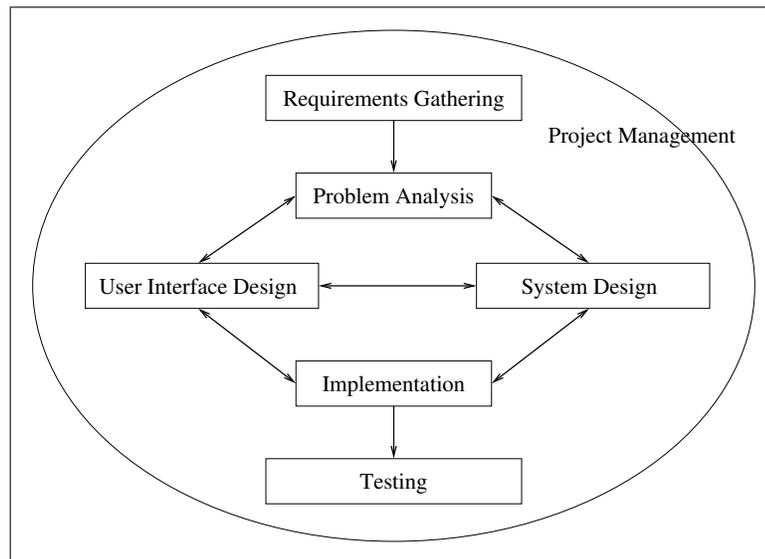


Figure 58: Relationships of development phases ([Cen97, fig. 7.1])

IBM OOTC's process contains 48 work products, which its authors "have found useful in practice" [Cen97, p. 77]. The work products are grouped into the following development phases: requirements gathering, project management, problem analysis, user interface modeling, system design, implementation, and testing. Figure 58 depicts the development phases and their relationships. As part of the IBM Global Services Method, there are now over 300 work product descriptions [Cam02]. These work products are organized into six domains: engagement, business, organization, application, architecture, and operations

OOTC work products

uses the definition of IBM OOTC's process. In contrast, RUP introduces *artifact* as one of the four primary model elements, defining it as "a piece of information that is produced, modified, or used by a process" [Kru99, p. 40].

²⁵⁷This makes the process an output-oriented approach for managing projects as opposed to a task-oriented approach [Sin03].

[Tho04].

The work products defined by this dissertation's process framework are as follows:

Process framework
work products

Intended Development Process (cf. Section 7.2.3): This work product instantiates a suitable tool development process.

Functional Requirements (cf. Section 7.2.4): This work product describes the tool's functional requirements.

Non-functional Requirements (cf. Section 7.2.5): This work product describes the tool's quality attributes.

Candidate Host Components (cf. Section 7.2.6): This work product identifies and describes suitable host components to construct the tool.

Host Product Customization (cf. Section 7.2.7): This work product shows strategies how a certain host product can be customized.

User Interface Prototype (cf. Section 7.2.8): This work product mandates the construction of a user interface prototype.

Technical Prototype (cf. Section 7.2.9): This work product mandates the building of technical prototypes to explore tool construction issues.

Tool Architecture (cf. Section 7.2.10): This work product documents the tool's high-level architecture.

The work product oriented approach of my process framework has been inspired by OOTC. However, the framework's individual work products are not the same as the ones described by OOTC because they are motivated by the process constraints of my tool-building approach. The process framework describes a minimum set of work products. A tailored version of the framework is free to introduce additional ones, as appropriate. Specifically, a tool-building project that experiments with a novel or unproven technology could introduce dedicated work products to make sure this risk is addressed adequately. For example, if a new reverse engineering tool wanted to employ XML databases as a repository—an unfamiliar approach to the project members—a Repository Technical Prototype work product could be defined. This work product could prescribe tasks such as creation of a sample schema, and benchmarking with realistic data sets.

The process framework has a number of desirable characteristics:

Process framework
characteristics

lightweight: I have argued before that a tool-building process should be lightweight (cf. Section 3.3.4). In keeping with the requirement of a lightweight process, the process framework itself defines only a minimal set of work products.

tailorable: The framework is tailorable in the sense that researchers are free to extend the process framework with their own work products. Researchers can also omit work products if their tool-building approach has different process constraints. For

instance, a research project may wish to incorporate an explicit traceability requirement for its process,²⁵⁸ or to eliminate the Technical Prototype work product if they have sufficient understanding about the technology.

separation of concerns: The work product oriented approach leads to a separation of concerns: Work products are independent from (1) tools and notation, and (2) development process and techniques.

adaptive: Adaptive processes (cf. Section 3.3.4) are desirable for tool-building projects because they often have an unstable environment (e.g., the composition and size of the development team can change considerably during its life-time). The work product orientation and separation of concerns of the process framework “addresses the risk of losing ground when tools, notations, techniques, method, or process need to be adjusted by allowing us to vary them while maintaining the essence and value of completed work” [Cen97, p. 15]. Thus, a process adaptation is characterized by changing the *how* and *when* while keeping the *what*.

reuse: The process framework defines a reusable set of work products. These can be reused by other researchers in other tool-building projects. In the best case, this leads to a situation where existing work products are reused whenever possible, reserving the definition of new work products for genuinely new information [Cam02].

7.2.2 Work Product Template

The set of work products of the framework are explained in more detail in the following sections. The work products descriptions use a template to provide a common structure:²⁵⁹

description: A general description of the work product.

purpose: The purpose served by developing this work product, including discussion about the impact of not having the work product, and reasons for not choosing the work product.

process constraints: Identifies the process constraints (i.e., reverse engineering domain, component-based software engineering, and target design space) that are addressed by the work product.

technique: A general description of how the work product can be constructed.

advice and guidance: Experiences and lessons learned that a tool builder may find useful.

references: Identifies sources that describe similar and related work products.

²⁵⁸Traceability is addressed, for instance, by RUP [Kru99, p. 142] and IBM OOTC’s process [Cen97, sec. 4.0].

²⁵⁹This template draws from template elements of the IBM OOTC process [Cen97, sec. 8.2] and the IBM Global Services Method [Cam02].

I believe that the use of a template to structure information in a standard way makes the work products easier to understand and apply. Templates are firmly established in the pattern community to structure patterns. Alexander introduced the idea of a pattern template in his book *A Pattern Language* [Ale77]. Gamma et al. took up the idea, using a template for describing their design patterns [GHJV95]. Researchers in other areas have started to use templates as well. For instance, Klein and Kazman structure the description of each Attribute-Based Architectural Style (ABAS) with the same four parts [KK99].

7.2.3 Intended Development Process

“Reuse the documented development process of a previous, successful project. . . . Beg, steal, or borrow some existing guidelines.”

– IBM OOTC team [Cen97]

description: The Intended Development Process work product outlines the process of the tool-building project. It addresses the entire project life cycle, describing at a high level (1) the development process itself, (2) techniques and notations that support the process, and (3) tools to automate and aid in the creation of work products. For example, the format of the requirements document may be presented, the way end users can be involved in the process may be discussed, the modeling notation for analysis and design may be specified, and the implementation language along with tool support may be stated.

When describing the process, the individual phases (or workflows) should be outlined along with their work products. For instance, RUP defines six core workflows: business modeling, requirements, analysis and design, implementation, test, and deployment. Furthermore, the key characteristics of the process should be identified. A process for my tool-building approach must exhibit the following characteristics:

feedback-based (cf. Section 3.3.1): If applicable, work products should be verified with feedback obtained from tool users. For instance, Functional Requirements and Non-functional Requirements work products should be assessed for completeness and appropriateness based on user feedback. The User Interface Prototype work product needs to be tested with users.

There are several approaches that emphasize feedback from tool users who can be incorporated into a process: user-centered design [FB04], user engineering [Vre03], participatory design [Gra01, p. 380], interaction design [Coo99], and the usability design process [GGB03]. One of the four core values of Extreme Programming is feedback [Bec00]. This is accomplished, for instance, with short release cycles and an on-site customer. In the context of a research project, Chirouze et al. say, “XP practices aim to provide feedback that is constant. Feedback is probably the main value that is expected when building a proof of concept. The obsession of XP for feedback is definitively valuable to researchers” [CCM05].

iterative (cf. Section 3.3.2): The process has to accommodate the iterative construction of work products. This means that existing work products can be altered in later iterations (e.g., iterative refinement of the Functional Requirements work product). Revisiting of work products is needed for both feedback-based and prototype-based development. With the notable exception of the waterfall model, all now popular processes (including IBM OOTC's process) are iterative.

An iterative process is needed for component-based development because it allows to address critical risks in early iterations, and has the flexibility to accommodate unforeseen events such as the release of a new OTS version. Boehm and Abts recommend that “given the vagaries of requirements in COTS-based software development, developers should adopt or modify more dynamic, risk-driven process models” [BA99].

prototype-based (cf. Section 3.3.3): The process requires work products constructed as prototypes. A User Interface Prototype enables user feedback early on in the tool-building project. User-centered design uses early prototypes for evaluation with users [Bev99]. In the face of uncertainty, Technical Prototypes can detect problems early on and help mitigate risks. For instance, a Technical Prototype can be used to assess the impact of an OTS upgrade [HTT97].

Several processes address prototypes. The spiral model proposes to use prototypes to study and assess risks [Sch96] [BA99]. SEI's Evolutionary Development method “facilitates effective user involvement, including hands-on experiences with a real product or prototype” [BCK98, sec. 18.3]. RUP recommends to use both evolutionary prototypes and exploratory throwaway prototypes as appropriate [Kru99, p. 161]. The IBM Global Services Method has two work product descriptions that address prototyping: Technical Prototype and User Interface Prototype [Cam02]. Lastly, rapid prototyping approaches (obviously) advocate the use of evolutionary or throwaway prototypes [GB95].

lightweight (cf. Section 3.3.4): The tool-building process has to be lightweight. A lightweight process is easier to learn and remember and has a better chance of being applied as intended. Specifically, there should be as few work products as possible, and the descriptions of the work product should be as concise as possible.

component-based (cf. Section 4.1.6): The process has to accommodate component-based software engineering. Specifically, the process has to provide guidance in assessing and selecting suitable (OTS) components, in adapting or customizing of the selected (OTS) components, and in assembling the component-based system. To accommodate these component-oriented tasks, the process framework defines several work products: (1) Candidate Host Components, (2) Host Product Customization, and (3) Tool Architecture. For OTS-based applications, Yang et al. identify three primary activities: (1)

assessment, (2) tailoring, and (3) glue-code development [YBBP05]. These activities are covered by the above three work products, respectively.

There are a number of processes that address component-based development. Morisio et al. propose a process based on observations from fifteen OTS-based development projects at NASA [MSB⁺02] [MSP⁺00]. There is also a process framework for OTS-based applications grounded in the analysis of both academic and industrial OTS-based applications [YBBP05] [PY04] [BPY⁺03a]. Henderson-Sellers describes an instantiation of the OPEN process for component-based development [Hen01]. SEI's COTS-based System Initiative has developed process frameworks (APCS [COP03] and others [BOS00]), provides an instantiation of the APCS framework that is compatible with RUP [AB02b] [AB02c], and gives guidelines on using the Capability Maturity Model Integration (CMMI) for COTS-based applications [TAB03]. There are also component-based processes that support product lines (e.g., Kobra [ABB⁺01] [ABLZ00]).

Besides the above characteristics, a process may have other ones: object-oriented,²⁶⁰ risk-aware,²⁶¹ architecture-centric,²⁶² use-case-driven,²⁶³ and so on.

To summarize, instead of proposing a concrete process, this work product identifies important characteristics that a process should have. This gives the tool builder the flexibility to define or choose a process and to tailor the process if necessary, while still providing guidance on how a suitable process should look like.

purpose: This work product makes the development process explicit and provides rationalizations for the process elements. Making the process explicit makes it easier to detect omissions and shortcoming of the approach.

Without this work product there is the risk of confused tool developers who are not aware how the project should be run, and repeated questioning and rationalization of why and how something is done.

This work product is important because it documents to what extent the process constraints of the tool-building approach are met. It may be omitted if the tool developer is a single person who has a track record of successful tool-building projects.

process constraints: This work product explicitly requires a process that satisfied the process requirements of the reverse engineering domain (i.e., feedback-based, iterative, prototype-based, and lightweight). It also requires from the process to accommodate the use of components for software construction.

²⁶⁰IBM OOTC's process is object-oriented. Korson and Vaishnavi list 17 different object-oriented analysis and design methods [KV92].

²⁶¹Boehm's spiral model specifically strives to mitigate risks in software development projects [Boe88].

²⁶²RUP is an architecture-centric process. It defines specific artifacts related to architecture (Software Architecture Description and Architectural Prototype), and a dedicated worker, the *architect*, who is responsible for the architecture [Kru99, p. 86f].

²⁶³RUP is a use-case-driven process [Kru99, ch. 6]; IBM OOTC is a scenario-driven process [Cen97, ch. 4].

technique: Developing a dedicated tool-building process is not easy. One should reuse existing knowledge whenever possible. Tool developers can start with their current process and identify changes and additions that are necessary in order to comply with the work product's process requirements. If the tool developers cannot articulate their current process, they can develop a descriptive life cycle model to make it explicit [Sca01].

Researchers who do not have an established process to start with can follow Cockburn's approach to just-in-time methodology construction [Coc03, sec. 3.6]. Alternatively, the process may be defined with the help of a tailorable process or process framework. For instance, RUP is a highly configurable process, which provides a lot of flexibility for tailoring it to specific needs.²⁶⁴

advice and guidance: A smaller tool-building project can employ an agile approach that has comparably little "ceremony" such as XP [ASRW02]. Generally, agile methods are characterized by close customer/developer interaction (i.e., feedback/prototype-based), rapid development cycles (i.e., iterative), and an approach that is easy to learn (i.e., lightweight) and modify (i.e., adaptive) [AWSR03]. Alternatively, one can start from a lightweight RUP configuration such as the dX process²⁶⁵ [GBN98] or Larman's process [Lar04].

As argued by Boehm and Turner, it is often better to build up a process as opposed to tailor it down [BT03, Observation 5]. The authors of the dX process recommend: "It may be that you need more formality, than dX supplies. If so, you can extend dX by carefully and sparingly adding practices and artifacts" [GBN98].

references: The IBM OOTC process has an Intended Development Process work product [Cen97, sec. 10.1]. There are also Analysis/Design Guidelines work products [Cen97, sec. 11.1] [Cen97, sec. 13.1], who describe the nature of analysis/design work products, and provide guidelines on the process parts that are concerned with analysis/design. RUP has a Development Case artifact (part of the management set of artifacts) that describes the actual process instance used by the project [Kru99, ch. 17].

²⁶⁴However, this can be also seen as a disadvantage because it cannot be used "out of the box." Fowler says, "RUP is a process framework and as such can accommodate a wide variety of processes. Indeed this is my main criticism of RUP—since it can be anything it ends up being nothing. I prefer a process that tells you what to do rather than provide endless options" [Fow03]. IBM has introduced the Rational Method Composer, which offers tool support for authoring, deploying, and tailoring of RUP [Hau05].

²⁶⁵In fact, dX is a configuration of RUP that happens to realize XP; turn "dX" upside down and you get "XP" [Fow03].

7.2.4 Functional Requirements

“Unless we are solving a trivial problem, new or different requirements will appear”

– Philippe Kruchten [Kru99, p. 53]

description: The Functional Requirements work product states what the proposed tool is to do. It describes the tool’s intended functions and its environment. Functional requirements can be captured as a list of system features, possibly prioritized by the tool user.²⁶⁶ Features are “services to be provided by the system that directly fulfill a user need” [Kru99, p. 141].

Since the requirements can be seen as a contract between the customer and the developers, they should be stated in such a way that they are understandable for both tool developer and tool user.

purpose: This work product captures the tool user’s expectations of the tool’s functionality. It provides a basis for communication between the tool users and tool developers, enables estimating of the development effort, and allows the defining of the user interface requirements.

Without this work product, tool developers do not have a clear grasp of what they are supposed to develop, and there is no possibility to verify the functional correctness of the tool (e.g., via test cases).

process constraints: This work product need not fulfill any process constraints. However, tools should consider to meet the functional requirements identified by this dissertation’s survey (cf. Section 3.2.7).

technique: Several simple techniques can be used to establish a feature list for a tool. Related tools that provide similar functionality can be inspected to identify useful features. Tool descriptions and tool comparisons published in the literature can also provide hints (cf. Section 3.2). Section 3.2.7 identifies important functional requirements for the reverse engineering domain (organized by tool component types). Requirements can be elicited from tool users via questionnaires, requirements workshops,²⁶⁷ and observations.

Functional requirements can be expressed in various ways. RUP advocates use cases.²⁶⁸ IBM OOTC’s process elaborates use cases into scenarios. A scenario is

²⁶⁶A simple approach to prioritize items is MoSCoW: (1) must have this, (2) should have this if at all possible, (3) could have this if it does not affect anything else, (4) won’t have this time but would like in the future (<http://en.wikipedia.org/wiki/DSDM>).

²⁶⁷The SOMA approach uses joint user and developer workshops for requirements capture and analysis [Gra01, sec. 8.13]. There are other approaches such as DSDM workshops and SEI’s Quality Attribute Workshop [BEL⁺03].

²⁶⁸A use case is “a sequence of actions a system performs that yields an observable result of value to a particular actor” [Kru99, p. 94].

characterized by a set of assumptions (initial conditions) and a set of outcomes. XP captures requirements as user stories [Coh04].²⁶⁹

advice and guidance: Functional requirements should be seen as flexible. Previously elicited functionalities can become obsolete because of user feedback, changing problems, and technology; or may turn out to be infeasible to realize. In OTS-based systems, often “the COTS capabilities effectively determine the requirements” [Boe00]. Yang et al. go as far as to recommend “don’t start with ‘requirements’” [YBBP05].

Requirements are still valuable as a description of the product vision, providing guidance for developers. However, they should be seen as guiding principles rather than inflexible rules [CW98].

Feature lists have the problem that “the items on the list describe the behavior of the software, not the behavior or goals of user” [Coh04, p. 136]. In contrast, use cases, user stories, scenarios, and such are preferable because they describe a tool user’s goals. Cohn relates the following experience :

“I can’t count the number of times that I saved a whole lot of work by taking a feature list and asking a customer to create scenarios that use those features. Often the customer will realize that the feature really isn’t needed, and that you should spend time building things that add value” (qtd. in [Coh04, p. 136]).

references: IBM OOTC’s process has Use Case Model [Cen97, sec. 10.1] and Analysis/Design Scenarios [Cen97, sec. 11.4] [Cen97, sec. 13.7] work products. RUP has a Use Case Model artifact (which is realized by the Design Model, implemented by the Implementation Model, and verified by the Test Model) [Kru99, ch. 6].

IBM OOTC’s Prioritize Requirements work product [Cen97, sec. 9.4] proposes to use a three-value scale to rate importance (vital, important, would be nice) and urgency (immediate need, pressing, can wait) of each requirement.

7.2.5 Non-functional Requirements

“Although functionality and other qualities are closely related, functionality often takes not only the front set in the development scheme but the only set. Systems are frequently redesigned not because they are functionally deficient—the replacements are often functionally identical—but because they are difficult to maintain, port, or scale or are too slow or have been compromised by network hackers.”

– Bass et al. [BCK98, p. 75]

²⁶⁹Beck defines a user story as follows: “One thing the customer wants the system to do. Stories should be estimable at between one to five ideal programming weeks. Stories should be testable” [Bec00].

description: The Non-functional Requirements work product states requirements not directly related to the behavior of the tool. Examples of non-functional requirements (or quality attributes) are reliability, usability, performance, and portability. These requirements usually take the form of constraints on how the system should operate. Like the Functional Requirements work product, this work product should be expressed clearly so as to be understandable for both tool user and tool developer. Without an understanding of non-functional requirements, the resulting tool may be useful, but not usable.

purpose: Whereas functional requirements drive analysis, non-functional requirements drive design. Whereas non-functional requirements influence the Tool Architecture, “functionality is orthogonal to structure, meaning that it is largely nonarchitectural in nature” [BCK98, p. 81]. However, in practice, not all aspects of a non-functional requirement influence the architecture. For example, usability requirements may mandate a certain font, or the use of radio buttons instead of check boxes; performance considerations may preclude a certain algorithm; et cetera [BCK98, p. 78].

process constraints: This work product need not fulfill any process constraints. However, tools should consider to meet the non-functional requirements identified for the reverse engineering domain in this dissertation’s survey (cf. Sections 3.2.1–3.2.6).

technique: To elicit non-functional requirements, similar techniques can be used as for the Functional Requirements work product. As an aid to check the completeness of quality attributes, the FURPS model, ISO/IEC 9126, or Brown’s classification can be used (cf. Section 3.2). If the functional requirements are expressed as use cases or scenarios, non-functional requirements can be extracted by asking for the constraints imposed by each actor on its associated uses cases [Cen97, p. 107].

Tool developers should especially consider to meet the non-functional requirements identified for the reverse engineering domain (Sections 3.2.1–3.2.6): scalability, interoperability, customizability, usability, and adoptability. There should be a rationalization for each of these quality attributes on how the tool will meet them.

advice and guidance: Non-functional requirements (e.g., usability) are often difficult to formalize and validate. Obtaining user feedback with prototypes can be used in such cases to clarify quality attributes. Often there are tensions between non-functional requirements. These should be explicitly identified and resolved.²⁷⁰

Other work products can support—in an explicit or implicit manner—the adherence to non-functional requirements. Examples of such work products are Technical Prototype (supports scalability), UI Prototype (supports usability), and Tool Architecture (supports interoperability).

²⁷⁰IBM OOTC’s process uses the Issues work product for that [Cen97, sec. 10.12].

For a proof-of-concept tool prototype, quality attributes such as usability, reliability, and security are of lesser concern. However, tool builders should consider that adding a non-functional requirement for a (prototype) tool later on in the development activity might be infeasible or only accomplished with considerable additional effort.

references: The IBM OOTC process captures non-functional requirements in the Non-functional Requirements [Cen97, sec. 9.3] and Target Environment [Cen97, sec. 13.4] work products. RUP has the Supplementary Specification artifact, which captures the quality attributes of the solution that are not present in the Use Case Model [AB02a, p. 42].

7.2.6 Candidate Host Components

“Developers seldom use formal selection procedures. Familiarity with either the product or the generic architecture is the leading factor in selection.”

– Torchiano and Morisio [TM04]

description: The Candidate Host Components work product addresses the identification, assessment, and selection of host products. In order to construct a component-based tool, it is necessary to first identify potential host products. Out of these candidates, the “best” one should be selected based on an assessment. To enable a comparison among candidate host products, assessment criteria have to be defined first.

The proposed Tool Architecture determines the number of the host products. For each (potential) host product, an instance of the Candidate Host Components work product should be created. For instance, a tool that plans to use extractor and visualizer OTS products needs two separate work product instances to document the selection decision for the extractor and visualizer. The decision to use a certain host component can impact the Tool Architecture.

Note that the selection decision may come to the conclusion that there is no suitable host component. In this case, tool developers have to implement the required functionality themselves.

purpose: This work product documents the rationale for selecting a certain host component. Making the selection of the component transparent increases the confidence of the tool developers into the assessment and selection. Conversely, it helps to uncover weaknesses in the assessment and selection approach.

This work product provides a catalog of candidate components along with their characteristics. Researchers can use this catalog as a knowledge base for other tool development projects, thus avoiding repeated effort in the assessment of the same components.

process constraints: This work product meets two constraints. It addresses component assessment and selection for component-based software engineering, and it requires the candidate host products to fall within the target design space.

technique: To identify suitable candidates a number of techniques can be used ranging from Web searches to literature reviews (cf. Section 4.1.6). The most promising approach for tool builders is to use the visualizer and extractor host component catalog introduced in Section 5.2 as a starting point.

The identified candidates are then characterized with this dissertation's component taxonomy (cf. Section 4.2): origin, distribution form, customization mechanism, interoperability mechanisms, and packaging. Additional criteria can be introduced as needed (e.g., execution platform and costs). Component taxonomies (cf. Section 4.2.7), CBSE approaches (cf. Section 4.1.6), and Functional/Non-functional Requirements [LM04] can be used as sources for suitable component criteria. Kell and Tiwana give seven OTS characteristics that IT managers have repeatedly mentioned as being important (i.e., functionality, reliability, cost, ease of use, vendor reputation, ease of customization, and ease of implementation) [KT05]. Golden's Open Source Maturity Model (OSMM) assesses products with six key product elements: software, support, documentation, training, product integrations, and professional services [Gol04]. If applicable, each criteria should have a capability statement (i.e., metrics to assess the criteria) and quantification method (i.e., means for obtaining the metrics) [LM04]. Characteristics of a good set of criteria are assessable, discriminating, non-overlapping, and significant [LM04].

The criteria can be evaluated qualitatively or quantitatively. OSMM assesses each key product element independently with a numerical score between 0 and 10 [Gol04]. Each score is determining with the help of checklists. The individual scores of the product elements are then added (using different weighting factors for each element) to produce a final score. Evaluators can also adjust the weighting factors (e.g., giving higher emphasis to training and lowering support) in order to reflect their specific needs. Kell and Tiwana propose a similar rating scheme [KT05].

advice and guidance: For this work product, one has to strike a balance between efficiency and completeness [AB02a, p. 194]. Yang et al. say that a good approach "balances the risk of erroneous choices from insufficient COTS assessment with the risk of delay from excessive COTS assessment, to find a risk-driven 'sweet spot' for the amount of COTS assessment effort" [YBBP05]. In practice, it is probably sufficient to consider only 2–4 competitive components. If sufficient tool-building experience is available, it may be feasible to focus on a single candidate for an in-depth evaluation right away.

To avoid *analysis paralysis*²⁷¹ [PC04] a host product should be selected relatively quickly based on a few criteria, but then tested with a series of Technical Prototypes.

²⁷¹http://en.wikipedia.org/wiki/Analysis_paralysis

If the prototypes should reveal that building of the tool with the chosen component is infeasible, it is still relatively cheap to repeat the selection process (while benefiting from the lessons already learned).

If the tool developers already have accumulated considerable experience with a certain component, it may make sense to selected it without considering other alternatives, given that “adopting a new product might require a long period of familiarization before effective and efficient use” [TM04].

references: The CMMI’s Supplier Agreement Management process area has a Review COTS Products practice that proposes the following work products: trade studies, price lists, evaluation criteria, supplier performance reports, and reviews of COTS products [Tea02, p. 242].

SEI’s EPIC RUP-based process framework defines three component-related artifacts. Market Segment Information captures the characteristics of competing components within a certain market [AB02a, ch. 7]. The Component Dossier artifact contains information about a certain OTS product [AB02a, ch. 8]. Component Screening Criteria and Rationale Guidelines document the evaluation and selection criteria for components [AB02a, ch. 9].

7.2.7 Host Product Customization

“More than once we had to bend requirements to make designs work.”
– Coppit and Sullivan [CS03]

description: The Host Product Customization work product outlines a strategy how to customize a certain host component in order to realize a tool’s functional and non-functional requirements. The process framework introduces a dedicated work product that explores a component’s customization capabilities, because tool development crucially depends on it.

Whereas the Candidate Host Components work product selects a host component among a number of candidates with several diverse criteria, this work product explores the selected component’s customization capabilities in more detail.

The construction of this work product is a way for the tool developers to familiarize themselves with the selected host components.

purpose: This work product should give tool developers the needed confidence that they can realize the envisioned tool with the selected host components. It also identifies customization strategies that can be reused for other tool-development projects.

Without this work product, there is a risk that missing customization capabilities of the host product, which threaten to compromise the requirements of the tool, are discovered relatively late in the project.

process constraints: The customization of the selected host products has to meet the characteristics outlined in the customization mechanism criteria of the target design space (cf. Section 5.1).

technique: This work product describes the customization mechanisms that a certain component offers. The techniques can be grouped according to the available mechanisms such as configuration file options, command-line options, API programming, and scripting support (cf. Section 4.2.3). This work product can also discuss testing and debugging strategies for the customized component.

If applicable, examples for each customization mechanism should be provided. For example, there could be code snippets that show how to add a new menu bar and menu item; create a new view or (pop-up) window; select and change the properties of a GUI element, and so on. There should be also references to relevant documentation in manuals, user guides, and FAQs; newsgroup discussions and user forums; reported experiences of other tool developers, et cetera. Thus, this work product can take the form of a “cookbook” that developers can use to solve recurring customizations.

Boehm et al. propose a (sub-)process for the customization of a single component [BPY⁺03a] [YBBP05]. The first step is to identify the component’s customization options. Next, a decision needs to be made which customization mechanism is used to implement which capabilities (if there happens to be more than one potential mechanism). When making a decision, one should perform a trade-off analyses between the effort that available customization mechanisms require and the functionality to be achieved via customization. Considerations are automation support (e.g., macro recorder), complexity (e.g., non-programmatic vs. programmatic), need for a detailed design, and development costs. If customization is found to be feasible, it should be planned first (involving a design) for complex systems.

advice and guidance: Especially for components that have many customization mechanisms, it can happen that too much time is spent documenting and exploring the options. To avoid this problem, tool developers should focus first on critical functionality that needs to be realized via component customization. Critical functionality can be identified with the Functional Requirements and User Interface Prototype work products. For instance, functional requirements may suggest that the selection of a GUI element by the tool’s user needs to be detected and requires the creation of a pop-up window. The screen layout of a user interface prototype may suggest that a user has to activate a certain functionality with the menu bar, tool bar, context menu, and a keyboard shortcut.

This work product is most effective if it interacts with other work products. It should be focused based on the Functional Requirements and User Interface Prototype work products. Conversely, the additional knowledge about the component’s customization capabilities may necessitate adjustments in (user interface) requirements. Fur-

thermore, the made experiences can be directly used in the construction of Technical Prototypes. In fact, the customization mechanisms can be described as a set of (rudimentary) Technical Prototypes.

references: One of the purposes of the EPIC Component Dossier artifact is to “record the history, rationale, and specific activities for customization and tailoring of the component” [AB02a, p. 172]. Specifically, the artifact should answer questions such as (1) what effort is involved in performing a customization, (2) must this effort be repeated for new component releases, and (3) can the component be configured to efficiently handle an appropriate range of performance expectations (e.g., transaction rates, number of threads, etc.)?

7.2.8 User Interface Prototype

“This is an example of a chicken-or-egg problem of user-centered design: We cannot discover how users can best work with systems until the systems are built, yet we should build systems based on knowledge of users and how they work. This is a user-centered design paradox. The solution has been to design iteratively, conducting usability studies of prototypes and revising the system over time”

– G. Marchionini, qtd. in [Elv05, p. 27]

description: The User Interface Prototype work product provides a representation of the tool’s user interface. It should not be concerned with other issues (e.g., persistence, data representations, or computational algorithms). It is built early on in development and may be an early version of the actual tool.

The prototype simulates interaction flows as well as the layout of the user interface, capturing the “main line” tool’s functionality. It may be a software simulation or paper-based. Ideally, a user interface prototype is constructed or validated by a human factors or user interface expert.

purpose: A user interface prototype can serve a number of purposes. Since it is easy for the tool user to understand, it can be used to solicit feedback on the functional completeness and usability of the user interface. This work product help to steer the system towards a user-oriented system as opposed to a developer-oriented one.

The prototype can be developed with the tool user and then used as input to create the Functional and Non-functional Requirements work products (“UI-driven requirements gathering”) [Cen97, p. 243]. This approach can speed up and focus requirements elicitation.

A user interface prototype may not be necessary if “the development effort is a re-make of an existing system with a user interface that is already widely accepted” [Cam02].

process constraints: This work product satisfies the requirement of a process that is prototype-based.

technique: Cameron suggests the follows general steps for developing a prototype [Cam02]: (1) define the prototype objectives, (2) choose the prototyping tool, (3) build the prototype, (4) investigate the prototype. A typical objective for a prototype is that it should be good enough to enable user interaction and feedback; and detailed and complete enough to support some kind of evaluation.

The Functional and Non-functional requirements work products can be used to identify the tool's "main line" functionalities that the prototype should focus on. Useful aids for prototype construction are also interaction design's scenarios [CRCK98] [Coh04, p. 141] and Cooper's personae [Coo99, ch. 9] [Coo03].

Before a user interface prototype is constructed, user interface and human-interaction advice, guidelines and standards should be consulted (e.g., [Nor98] [Joh00] [Ras00] [Gra01, sec. 9.10] [KL98]). To simulate the user interface, screen flows and layouts can be used. Screen flows describe the sequence of screens (e.g., windows, dialogs, and prompts) that a tool user will see when interacting with the system. Screen layouts depict the actual appearance of screens (i.e., the arrangement of graphical elements). They can be developed with a GUI builder²⁷² or drawn on paper. Tool users can be walked through the prototypes or interact with the prototype through a "Wizard of Oz" simulation²⁷³ [WL05].

advice and guidance: Tool developers often do not seriously consider the use of paper-based prototypes. However, they can be very effective in the beginning of the project for requirements elicitation and rapid user feedback. For example, Holzinger had positive experiences with this approach:

"Using common office supplies (paper, markers, index cards, scissors, and transparency film), we sketched the main screen and each changeable interactive interface element, including dialog boxes, menus, and error messages. The paper mock-up with its handwritten text, crooked lines, and last-minute corrections wasn't very neat but was good enough to show user-centered design participants and developers what the screens might look like. The mock-up provided a good basis for playing out some workflows" [Hol04].

The prototype should focus on the few screens and workflows that tool users are expected to spend most of their time with ("80/20 rule").

²⁷²Examples of products with GUI-builder functionality are IBM VisualAge, Apple Hypercard, Interface Builder, Visual Basic, and Web browsers (http://en.wikipedia.org/wiki/GUI_Builder).

²⁷³With this kind of simulation the impression of a working system is created with the help of a human acting behind the scenes (the "wizard") to produce output in response to user requests [GCH83].

The user interface of a component-based tool is constrained by its host products. For this reason, it is important to understand the host product's constraints first before certain layouts and interaction patterns are proposed (e.g., drag-and-drop cannot be implemented for Web applications). The Candidate Host Components and Host Product Customization work products can be used as guidance. Furthermore, host products that have scripting support can be used as a rapid prototyping environment. The development of the prototype can give the tool developers valuable insights for the actual tool development.

references: The IBM OOTC process has a User Interface Prototype work product [Cen97, sec. 12.4] as well as Screen Flows [Cen97, sec. 12.2] and Screen Layout [Cen97, sec. 12.3] work products to describe the sequence of screens, and to depict the actual appearance of screens, respectively. The IBM Global Services Method also has a User Interface Prototype work product [Cam02]. RUP defines Use-Case Storyboard and User-Interface Prototype artifacts (part of the requirements set) [Kru99, ch. 9]. SEI's EPIC process framework has a Executable Representation artifact that can be realized by a paper-based prototype early on in the project [AB02a, p. 242].

7.2.9 Technical Prototype

“Prototype early and often.”

– Michael Sparling [Spa00, Lesson 5]

description: The Technical Prototype work product is concerned with the construction of a prototype to explore issues related to the design, implementation, and architecture of the tool under construction. This is in contrast to the User Interface Prototype, which is exclusively concerned with user-interface design.

A prototype can be characterized as a “learning vehicle that allows us to understand some aspect of a solution in the context of the real problem” [Cen97, p. 380]. A technical prototype can be an exploratory, throw-away prototype or an evolutionary prototype, which successively evolves into the final tool (cf. Section 3.3.3).

Prototyping can be used as a risk mitigation technique [Cen97, sec. 17.4]. Often there are uncertainties of tool development that cannot be resolved by theoretical analysis or research alone. In this case, an exploratory prototype is a practical experiment that can provide useful information.

purpose: This work product can serve diverse purposes; for example, it can validate requirements, reduce uncertainty before designing, assess the feasibility of a new or unfamiliar technology, mitigate technical risk, demonstrate critical features, measure performance characteristics of components or features, and aid generation of test cases for product evaluation [AB02a, p. 254] [PMCS05, Key Concept 6.2.2] [Tea02, p. 402] [Kru99, p. 159] [MN98].

In the context of component-based development, exploratory prototypes are especially useful to provide input for the Candidate Host Components and Host Product Customization work products. Public information about components can be inaccurate, misleading, or outdated. For example, Lewis and Morris caution that “one should guard against accepting at face value product feature checklists that appear in trade magazines. These are sometimes produced by vendors specifically to present their product in best light” [LM04].

process constraints: This work product satisfies the requirement of a process that is prototype-based.

technique: The construction of a prototype should be accomplished with little resources and time. Typically a rapid prototyping approach involves a scripting or domain-specific language (cf. Section 3.3.3). Exploratory prototypes that are constructed to resolve technical issues will be typically realized via the customization of host components. Similar to the User Interface Prototype, this work product provides input for component selection and requirements elicitation; and valuable experiences for the actual tool construction. This is consistent with the experiences made by the developers of the Galileo tool, who say that “we had to use a development style based on prototyping and the ongoing exploration of both component properties and user requirements” [CS00b]. Sommerville proposes the following approach:

“Use hands-on product evaluation to further refine your choice of system and your system requirements. By this stage, you effectively have a prototype system for experiment, and you can use this with stakeholders to drive the requirements elicitation process” [Som05a].

A tool-development project typically uses several technical prototypes to mitigate different risks, and to resolve diverse issues. RUP, for instance, “advocates the development of an evolutionary structural prototype throughout the elaboration phase accompanied by any number of exploratory prototypes” [Kru99, p. 161].

advice and guidance: As suggested by RUP, the tool should be developed as an evolutionary prototype. If feasible, the code from exploratory prototypes should be refactored to augment the evolutionary prototype.

Evolutionary prototyping and the use of scripting languages can lead to brittle code that is difficult to enhance, causing maintenance problems in the future [GB95]. These problems can be mitigated by starting out the evolutionary prototype as an architectural prototype that adheres to the Tool Architecture work product. The EPIC process framework describes such an approach [AB02a, p. 223f]:

“The architectural prototype is a partial implementation of a solution, built to demonstrate selected system functions and properties. ... It

should at least address the critical use cases and the non-functional requirements . . . that have an impact on the solution architecture. . . .

This prototype is not a quick-and-dirty throwaway prototype, but it will evolve through the construction phase to become the final solution. The architectural prototype is built with the intention of retaining what is found to work (and satisfies requirements) and making it part of the fielded system.”

references: The IBM Global Services Method has a Technical Prototype work product description [Cam02]. SEI’s EPCI process framework has a Executable Representation artifact that can be realized by an evolutionary, architectural prototype [AB02a, p. 242].

7.2.10 Tool Architecture

“Software architecture is part of product quality and isn’t tied to a particular process, technology, culture, or tool.”

– Nord and Tomayko [NT06]

description: The Tool Architecture work product captures the system architecture of the tool under construction. It can be seen as the set of early, global design decisions.

The architecture places constraints on the tool’s design and implementation. It prescribes the structure of the system in terms of modules and how they are related with each other. In the conceptual architecture, the modules are abstractions of the tool’s main functionalities, and relationship between modules indicate sharing of data or transfer of control (cf. Section 2.3.5).

The architecture is often visualized with diagrams [SG96] [BCK98], but it can also take the form of a textual description of non-functional requirements and derived architectural decisions [Cen97, sec. 13.2].

purpose: An architecture can be used to show the partitioning of the system into components; the communication patterns and mechanisms between components; the nature and services of the used components; et cetera.

Without an architecture, it may be difficult to reason about tool properties, to communicate its design principles to new project members, and to maintain its (conceptual) integrity. The architecture can be used to reason about certain Non-functional Requirements of the system (e.g., performance, modifiability, and reusability).

process constraints: This work products supports component-based software engineering by requiring an architecture that identifies the tool’s functional decomposition in terms of components.

technique: The tool’s architecture can be described as a conceptual architecture diagram that shows the tool’s main components. Often a single customized host component

implements the functionality associated with a certain tool component type (i.e., extractor, analyzer, visualizer, or repository).

In the early project stage, a component in the diagram may indicate its intended functionality without revealing its realization. For example, there may be a component that is meant to implement repository functionality. Only later on in the project a decision will be made about the nature of the repository (e.g., local file system, relational database, object-oriented database, or XML database). Still later on, the actual OTS component may be chosen (e.g., DB2, MySQL, or Postgres for a relational database).

The diagram should identify the individual host components (e.g., OTS products), and their interoperation. The kind of interoperability (i.e., data, control or presentation integration) among components should be indicated as well. Furthermore, the architecture can reveal needed glue code and wrappers.

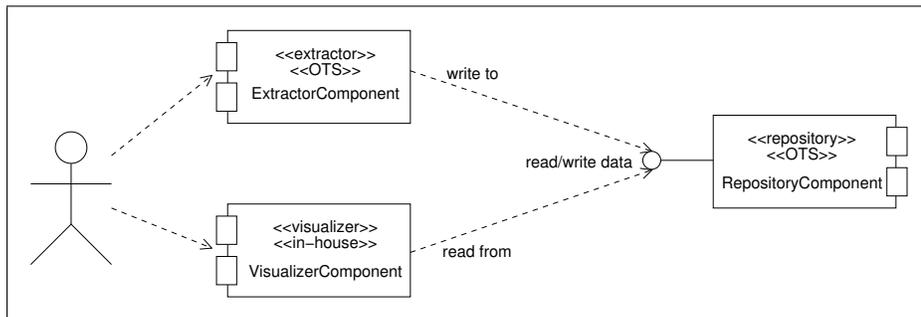


Figure 59: A tool architecture in UML

Architectural diagrams are typically visualized with UML, which is the de facto standard modeling language. UML offers modeling elements to represent components and interfaces [Crn02] [HN01] [Kob00] [Kru98]. UML stereotypes can be used to convey additional meaning. For example, Egyed et al. use stereotypes to distinguish between <<in-house>> and <<COTS>> components [EJB04]. Similarly, stereotypes can be used to identify the tool component type. Figure 59 shows an example of a tool’s architecture modeled in UML. The tool is composed of three components. The repository component exposes an interface (shown in UML’s shorthand (“lollipop”) interface notation) to read and write data. The extractor and visualizer components use this interface to write and read data, respectively. Stereotypes are used to indicate the tool component types, and to distinguish between third-party and custom components.

advice and guidance: The tool architecture can be effective to assess whether the resulting tool meets established design principles. For example, a component-based system will be more maintainable if it minimizes coupling among components and relies on open standards [VD98]. Boehm et al.’s MBASE approach identifies

issues—grouped into *simplifiers* and *complicators*—that make the development of component-based systems easier or harder [Som05a]. Simplifiers that are reflected in the architecture include “clean, well-defined APIs” and “a single COTS package;” complicators include “multiple, incompatible COTS packages.”

To come up with an architecture it is often helpful to reuse existing expertise in the form of reference models and architectures, architectural patterns, or architectural mechanisms [BCK98, ch. 2] [Kru99, ch. 5]. In a reference model or architecture, tool component types can identify a tool’s functionalities and represent its software components. Several researchers have proposed reference models and architectures for reverse engineering tools (e.g., [LA97] [MK96] [HEH⁺95]), which can be used as a starting point to define a tool’s architecture. Figure 59 can be seen as a reference architecture that shows the data flow between the extractor and visualizer tool components.²⁷⁴ An architectural mechanism provides a “common solution to a common problem” [Kru99, p. 90]. For example, a certain communication mechanism (e.g., a file-based data transfer), or a component that realizes standard functionality (e.g., a database system). An architectural pattern is a “solution to a recurring design problem that arises in specific design situations” [Kru99, p. 91]. For example, Figure 59 can be seen as an architectural pattern whose components `ExtractorComponent`, `VisualizerComponent`, and `RepositoryComponent` can be instantiated with the concrete products `vacppparse`, `rigiedit`, and `LocalFileSystem`. The file-based communication (`write to` and `read from`) between extractor and visualizer can be seen as an architectural mechanism.

references: The IBM OOTC process defines System Architecture [Cen97, sec. 13.2] and Subsystem Model [Cen97, sec. 13.5] work products; there is also a Rejected Design Alternatives work product [Cen97, sec. 13.11] that can be used to record why a certain architecture was not chosen. The IBM Architecture Description Standard recommends to produce several architecture-related work product descriptions, among them: Non-functional Requirements, Architecture Overview Diagram, Reference Architecture Fit/Gap Analysis (which documents the selected reference architecture, and identifies how the system deviates from it), and Architectural Decisions (which records and rationalizes architectural decisions) [C⁺98, p. 20]. RUP defines two artifacts related to architecture: Software Architecture Document and Architectural Prototype [Kru99, p. 86]. The EPIC process framework defines an Architecture Document (which contains various high-level architectural views of the system, information about the combination of components, component integration strategy, resolved and unresolved mismatches, key decisions, and lessons learned) [AB02a, p. 239]

²⁷⁴This reference architecture can be easily mapped to the Abstract Data Repository ABAS in order to further reason about its implications [KK99, app. B].

7.2.11 Discussion

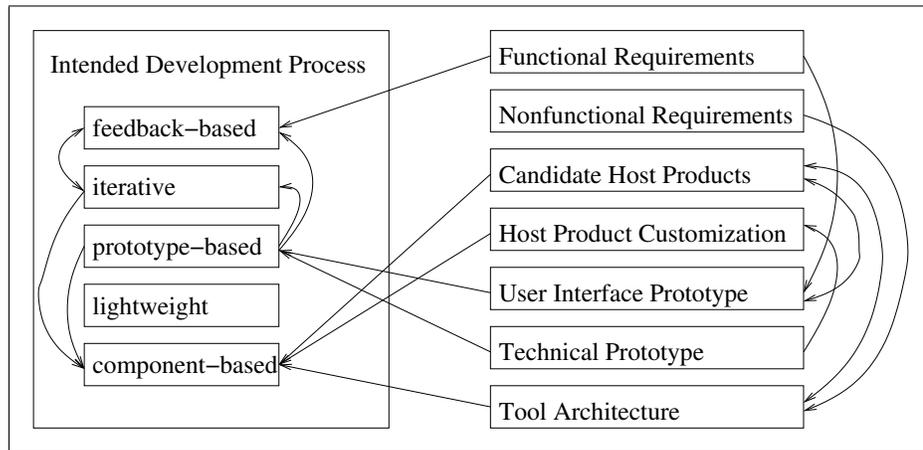


Figure 60: Relationships of the process framework's work products

The previous sections have discussed the work products of my proposed process framework for tool building. These work products support and depend on each other, forming a network of relationships. Figure 60 depicts the important relationships in a diagram. For the Intended Development Process work product, the diagram also shows the required process characteristics. The process characteristics have relationships with each other as well as other work products. Arrows in the diagrams can be read as a “supports” or “enables” relationship.

Process framework
relationships

The relationships among the work products (1) motivate the existence of each work product, and (2) illustrate the implications of omitting a certain work product. In the following, each relationship in Figure 60 is briefly motivated:

iterative ↔ feedback-based: Iterative processes mandate the continuous refinement of work products, which can then be used for continuous feedback from tool users. Since iterative development proceeds without a complete and detailed requirements specification, initial and continued “feedback is used to clarify and improve the evolving specifications” (e.g., via a series of requirements workshops) [Lar04, p. 18]. Kruchten states that “developing software iteratively . . . enables and encourages user feedback so as to elicit the system’s real requirements” [Kru99, p. 8].

There is also an inverse relationship. Bass et al. note that “by getting frequent user feedback on products or prototypes, change is accommodated by accepting the need for rework and minimizing its effect” [BCK98, p. 403].

prototype-based → iterative: In iterative development, an executable system should be produced at the end of each iterative development cycle. An evolutionary prototype is well suited to achieve this goal. A series of related prototypes that converge on a

consensus about the requirements results in an iterative prototyping process [LS92, sec. 1.2].

prototype-based → **feedback-based**: A prototype-based process mandates the creation of prototypes. Since prototypes can be created relatively quickly and cheaply (e.g., user interface prototypes), they enable user feedback early on in the project's life time. Holzinger says that "prototyping offers a quick way to incorporate user feedback into a design" [Hol04]. Furthermore, research by Keil and Carmel shows that prototypes are a very effective customer-developer link [KC95, tab. 4a].

prototype-based → **component-based**: Component-based development has various risks, among them missing functionality of OTS components and developers' lack of experience with a component's technologies. Technical prototypes are important to mitigate uncertainty in a project involving components.

iterative → **component-based**: The use of prepackaged functionality in the form of components typically leads to the situation that not all requirements can be met. Iterative development accommodates changes to requirements because it enables the revisiting of the Functional and Non-functional Requirements work products.

Functional Requirements → **feedback-based**: The Functional Requirements work product supports feed-back based requirements elicitation. It provides a snapshot of the current understanding of the tool's functional requirements (obtained by user feedback and other means).

Candidate Host Products → **component-based**: The Candidate Host Products supports component-based software engineering by explicitly requiring a rationalization for the selection of host components.

Host Product Customization → **component-based**: The Host Product Customization work product supports component-based development by mitigating the risk of basing the tool development on an unsuitable host component.

User Interface and Technical Prototype → **prototype-based**: User Interface and Technical Prototype work products support and enforce a process that leverages prototypes.

Tool Architecture → **component-based**: The Tool Architecture work product identifies the tool's leveraged host products and their interaction mechanisms.

Functional Requirement → **User Interface Prototype**: The Functional Requirement work product provides the necessary information for the construction of a User Interface Prototype.

Non-functional Requirements → **Tool Architecture**: Non-functional Requirements can have an influence on the Tool Architecture. Examples of such requirements are certain interoperability mechanisms (e.g., presentation integration of two components),

or the ability to easily accommodate a new programming language in a tool with multi-language support.

Candidate Host Products ↔ User Interface Prototype: The choice of a certain host component constrains the options of how to realize the tool's user interface. Conversely, certain requirements for the user interface may preclude the use of a component.

Technical Prototype → Host Product Customization: Technical Prototypes play an important role to assure that the envisioned customizations of the host products are indeed feasible.

Tool Architecture ↔ Candidate Host Products: The chosen Tool Architecture can constrain the applicability of certain Candidate Host Products. Conversely, the choice of a certain host product may preclude a certain Tool Architecture.

To better understand the properties and scope of my proposed process framework, I characterize it using the NIMSAD method evaluation framework [Mat04]. This allows researchers who are interested in using my process framework to compare it with other available methods and processes. The NIMSAD framework uses four categories to characterize processes: context (i.e., characterizing the problem situation), user (i.e., characterizing the problem solver or users of the method), contents (i.e., characterizing the problem solving process), and validation (i.e., evaluating of the method itself as well as its outputs).

Process framework
evaluation

NIMSAD has been used to evaluate methods for both product line architecture design (i.e., COPA, FAST, FORM, KobrA, and QADA) [Mat04], and component-based software development (i.e., Catalysis, OMT++, and RUP) [FHA99] [For02, ch. 2]. My evaluation combines elements from the two instantiation of the NIMSAD framework mentioned above. I primarily rely on Martinlassi's instantiation [Mat04], but omit elements specific to product lines. The categories and elements of the framework are given in Table 13.

The evaluation framework allows me to characterize my process framework. It also make it explicit which parts of the process are already provided and which parts still need to be instantiated. The evaluation of my process framework is as follows (cf. Table 13):

process goal: The goal of my process framework is to enable the effective construction of component-based tools that satisfy the requirements of tool users.

application domain: The application domain is tools for reverse engineering in particular and software engineering in general. However, the process framework might be applicable to other domains as well.

process start: The process framework starts out with the assumption that a new tool should be developed, which has to meet certain (perceived) requirements.

process outputs: The output of the process are a set of work products. In terms of the process' goal, the resulting tool is the most important work product.

Category	Element	Question
context	process goal	What is the specific goal of the process or method?
	application domain	What is the application domain the process is focused on?
	process start	Which incidents initiate the use of the process?
	process outputs	What are the results of the process?
user	process stakeholders	Who are the stakeholders addressed by the process?
	stakeholders' motivation	What are the stakeholders' benefits when using the process?
	needed skills	What skills are needed to accomplish the tasks required by the process?
	process guidance	How does the method guide the user while applying the process?
contents	process structure	What are the design steps that are used to accomplish the process' goal?
	artifacts	What are the artifacts created and managed by the process?
	tool support	What are the tools supporting the process?
	language	Does the process define a language or notation to represent the models, diagrams and other artifacts it produces?
validation	method maturity	Has the process been evaluated in practical industrial case studies?
	output quality	How does the process evaluate the quality of the output it produces?

Table 13: Process characterization framework

process stakeholders: The stakeholders addressed by the process are the tool user and tool developer.

stakeholders' motivation: Both tool user and tool developer strive for a tool that satisfies the requirements of the tool user. The tool developer is interested in a process that allows to build tools in an effective, repeatable, and predictable way.

needed skills: The process framework assumes that the tool user understands the (potential) benefits of reverse engineering tools. The tool developer needs knowledge about the reverse engineering domain, and the full life-cycle of software construction. Specifically, the tool developer needs to know or acquire the skills to produce the process framework's work products.

process guidance: The process framework provides a skeleton that needs to be instantiated. There are—deliberately—few constraints on how to accomplish this. Instead,

guidance can be obtained from suitable process frameworks (e.g., RUP [Kru99, ch. 17]).

process structure: The process framework does not define concrete sequences of activities. It does not prescribe when a certain work product needs to be created or modified. However, there are implicit dependencies between work products (e.g., creating a Tool Architecture before any tool requirements have been obtained does not make much sense).

artifacts: The artifacts of the process framework are the defined work products (and others that a process instantiation may add).

tool support: The process framework itself has no tool support,²⁷⁵ and its work products do not mandate any tools. However, some work products may provide suggestions for tools to create and manage the work product.²⁷⁶

language: Similar to the process framework's approach to tool support, it does not mandate any languages for expressing the work products.

method maturity: So far there is no case study of an instantiation of the framework. However, there are tool-building case studies and existing processes that indicate the usefulness of individual work products.

output quality: The process framework does not prescribe or require an evaluation of the produced work products.²⁷⁷

7.3 Summary

This chapter discussed recommendations and lessons learned of my tool-building approach. To structure the discussion, the tool perspective and the process perspective of the approach was addressed separately. For the tool perspective, a number of concrete lessons learned—motivated by tool-building experiences—were provided. For the process perspective, a process framework that supports my tool-building approach was introduced and motivated.

This chapter shows that my tool-building approach is both feasible and effective. It is feasible in the sense that it enables a tool builder to realize tools with typical reverse engineering functionalities. It is effective in the sense that it can produce tools that satisfy the requirements of an idealized tool user as expressed by the survey's tool requirements. It is also effective in the sense that a process can be defined for it that enables the tool developer to realize reverse engineering tools in a repeatable and predictable manner.

²⁷⁵However, tool support for other processes may be used; for instance, IBM offers for RUP the Rational Method Composer.

²⁷⁶Tool suggestions can be obtained from some processes; for instance, the IBM OOTC process makes tool suggestion for each work product [Cen97, app. C].

²⁷⁷Few processes address evaluation [Mat04] [For02, ch. 2]. The IBM Global Services Method (informally) discusses “Validation and Verification” in its work product descriptions [Cam02]. RUP gives evaluation criteria to assess the project after each of its four phases [Kru99, ch. 4].

8 Conclusions

In order to provide an understanding of my approach to component-based tool-building, this dissertation has to cover a number of different topics. Reverse engineering is the domain in which I investigate tool building. Consequently, Chapter 2 provides an introduction into this domain, mainly for readers who are not intimately familiar with it.

Summary of
dissertation

A thorough understanding of the reverse engineering domain is necessary to understand its (unique) requirements, which are addressed in Chapter 3. To construct reverse engineering tools, it is necessary to first understand the tool's requirements. Whereas each reverse engineering tool has different requirements, there are both quality attributes and functional requirements that are common to many tools or kinds of tools. Chapter 3 identifies these requirements as well as the requirements for an appropriate tool development process. The requirements are derived with an in-depth literature survey that identifies and collects diverse requirements. The identified requirements are voiced by a significant number of researchers who are part of the reverse engineering community; thus, these requirements are more trustworthy compared to a single researcher's opinion, and represent a sound body of knowledge for this domain.

In this dissertation, the tool requirements from the literature survey are used as a yardstick for the evaluation of reverse engineering tools and tool-building processes. Specifically, they are used to assess my proposed tool-building approach to construct tools from components. Chapter 4 introduces software components, which are the building blocks of my tool-building approach. There are many different definitions of components. Hence, it is necessary to establish what software component means in the context of this dissertation. The use of components for software construction has its unique challenges, which are addressed by component-based software engineering. Chapter 4 also introduces a taxonomy to classify components; this makes it possible to reason about characteristics of components and how these characteristics affect the construction of tools.

My tool-building approach does not attempt to be equally applicable for all kinds of components. Instead, it is scoped by identifying the characteristics of suitable components in the target design space in Chapter 5. This chapter also provides a catalog of visualizer and extractor components. The catalog exclusively describes components that have been used already as building blocks for tool construction. Unfortunately, most tool builders do not provide details on their approach and made experiences—they focus on the tool as the desired end product, but do not reflect on how it was produced. However, there are examples of researchers who understand that component-based tool-building constitutes a paradigm shift that needs to be better understood, and that it impacts both the tool builder and the tool user. Chapter 5 discusses in depth three outstanding tool-building case studies (VDE, Desert, and Galileo), summarizing interesting insights on how these component-based tools have been constructed, and distilling valuable experiences and lessons learned reported by the tool builders. These case studies also illustrate that publishing about tool building is indeed a valuable research contribution in its own right.

Chapter 6 discusses my own tool-building case studies. The six case studies realize different reverse engineering tools, covering a spectrum of reverse engineering functionalities, host components, and technologies. These case studies along with the ones reported by other researchers constitute a body of valuable tool-building experiences. However, these experiences represent somewhat “raw” knowledge.

Chapter 7 distills the various experiences made by myself and other tool builders into ten lessons learned. The lessons cover important quality attributes for tools as uncovered by my tool survey. Addressing these quality attributes promises to result in better tools that are more likely to meet the requirements of tool users. Tool construction is software development and as such needs a well-defined process. Chapter 7 discusses a suitable process framework for component-based tool development that can be instantiated by tool builders. The process framework encodes desirable properties of a process for tool-building (as identified by my survey), while providing the necessary flexibility to account for the variations of individual tool-building projects. This dissertation argues that a more disciplined approach for component-based tool-building is necessary, and shows that it is indeed feasible.

The following sections summarize the major contributions of this dissertation, propose directions for future work, and close with parting thoughts.

8.1 Contributions

In the following, I identify the major contributions of this dissertation:

reverse-engineering requirements survey: Chapter 3 has introduced a comprehensive survey of requirement in the reverse engineering domain. The survey covers both tool requirements (cf. Section 3.2) and process requirements (cf. Section 3.3). It has been conducted as a systematic review and is grounded in a formal process based on evidence-based software engineering.

Specific contributions of the survey include:

- The survey—to my knowledge—provides the most comprehensive coverage of requirements in the reverse engineering domain to date.
- The survey has enabled me to qualitatively assess my tool-building case studies with objectively derived tool requirements.
- The survey is a vehicle for a requirements-driven approach to tool development, evaluation, and research in the reverse engineering domain (cf. Section 3.2.9).
- The survey provides an example for other researchers on how to employ evidence-based software engineering in their own research.

component taxonomy: Section 4.2 introduces a new taxonomy to characterize components and component-based systems. The taxonomy is grounded in the work of other researchers who have identified characteristics of (off-the-shelf) components, connectors, and system architectures, distilling them into a lightweight taxonomy.

The taxonomy is lightweight in the sense that it consists of only six top-level criteria whose values should be easy to determine given only basic knowledge about the evaluation target (e.g., in the form of developer documentation).

Specific contributions of the taxonomy include:

- The taxonomy has enabled me to characterize and identify the kinds of components and component-based systems that are applicable for my tool-building approach.
- The taxonomy is lightweight, enabling a simple yet effective characterization of components.
- The taxonomy along with the identified related work is comprehensive and well researched; as a result, the provided resources can serve as good starting points for deriving a small and lightweight taxonomy for other purposes.

host components catalog: Section 5.2 provides a comprehensive catalog of host components that can be used for tool-building. To enable research to better judge the applicability of these components to realize specific functionalities, they are grouped by tool component type (i.e., components to realize visualizers as well as extractors). Most importantly, the catalog includes only components that have been already used to realize tools and thus contains only components that have already proven their feasibility and practicality for tool-building.

Specific contributions of the catalog include:

- The large number of tools that have been built leveraging host components attests that component-based tool-building is feasible and indeed practiced.
- The catalog can be used by researchers to browse and identify appropriate components for their own tool-building endeavors.

(own) tool-building case studies: Chapter 6 describes the construction of various reverse engineering tools that have been built using components. These tools have been constructed with my involvement as part of the Rigi group's ACSE research project. Section 5.3 gives detailed accounts of other researchers' tool-building experiences involving components.

It is my believe that descriptions of such experiences are sorely missing in the reverse engineering research domain. Since tool-building is expensive but pervasively used as a means for research validation, published experiences that help researchers build better tools more effectively have the potential to make research as a whole more effective.

Specific contributions of the case studies include:

- The case studies have enabled the distilling of guidelines to improve tool-building.

- The case studies provide valuable descriptions of tool-building experiences and lessons learned that can benefit other researchers.

component-based tool-building approach: Chapter 7 discusses recommendations and lessons learned for a component-based approach to build reverse engineering tools. This approach can be characterized as using customizable host components to graft reverse engineering functionality on top of these components.

While the use of components for tool-building is not new (as exemplified by the host components catalog), I have identified a particular approach to tool-building that is scoped by targeting components with certain well-defined characteristics (as expressed by the target design space discussed in Section 5.1).

I believe that this approach has the potential (1) to deliver tools that satisfy the requirements of its users, and (2) to build tools in a manner that satisfies the requirements of an effective tool-building process.

Based on my tool-building case studies and other researchers' tool-building experiences, I have qualitatively evaluated my tool-building approach by assessing the resulting tools with the tool requirements identified in my survey. My tool-building approach is guided by the process requirements identified by my survey, incorporating the requirements in the form of suitable practices and work products. These work products can be used as building blocks for the instantiation of a dedicated tool-building process.

Specific contributions of this chapter include:

- I provide a vision on how components can be leveraged to advance the current state-of-the-art of tool-building in the reverse engineering domain, potentially leading to better tools.
- Ten lessons learned that are distilled from a number of tool-building experiences of myself and other researchers.
- Definition of a process framework, consisting of a set of core work products, that supports my tool-building approach.

8.2 Future Work

This section identifies possible future research. I first discuss future work surrounding requirements elicitation for the reverse engineering domain, followed by future work concerning my proposed tool-building approach.

8.2.1 Reverse-Engineering Requirements

My reverse-engineering requirements survey (cf. Chapter 3) has laid a solid foundation for further research into this area to build upon. Possible future research directions are as follows:

comparison of other domains' requirements: The survey has identified the requirements for reverse engineering. It would be interesting to conduct similar surveys for other domains (e.g., compiler construction) to see whether there are different emphases on requirements.

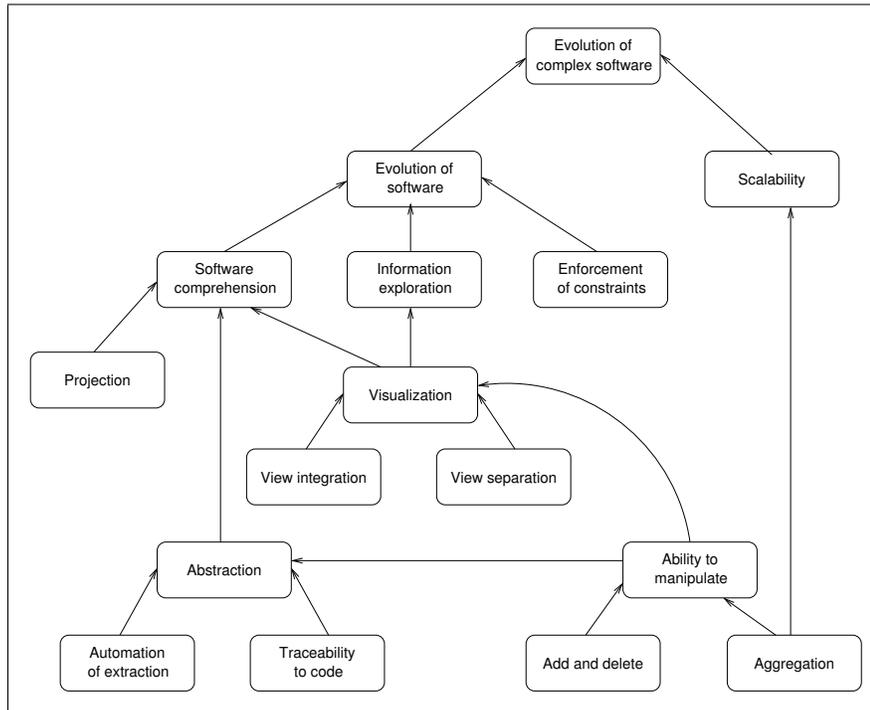


Figure 61: Goal model for software evolution ([AMM03])

interdependencies of requirements: The survey has mostly focused on the requirements in isolation. This is sufficient for tool assessment. However, there are complex interdependencies among requirements in the sense that a certain requirement can positively or negatively influence another one. Future research could explore these interdependencies in a more systematic manner, which might lead to a more predictable tool-building process. A starting point in that direction could be the NFR-Framework, which models dependencies between quality attributes as goal graphs [MCN92] [CNY95]. For instance, Amyot et al. present a goal model for software evolution that also touches on reverse engineering [AMM03] (cf. Figure 61). A more detailed goal model targeted at reverse engineering could expose important requirements and their interdependencies.

evaluation of quality attributes: It is not a trivial task to assess whether a particular piece of software fulfills a certain quality attribute. A suitable evaluation technique depends on the particular quality attribute and how it is defined.

Requirement	Evaluation approaches
scalability	<ul style="list-style-type: none"> • performance benchmarks [Hen00] • performance ATAM [BKM01] [VMB03] • performance ABAS [KK99] [KKB⁺99]
interoperability	<ul style="list-style-type: none"> • SAAM [KABC96]
extensibility	<ul style="list-style-type: none"> • modifiability ATAM [BKM01] [VMB03] • modifiability ABAS [KK99] • ALMA (modifiability) [LBvVB02] [BLBvV04] • SAAM (modifiability and evolution) [KBAW94] [LBKK97] • scenario profiles for software change [BB00]
usability	<ul style="list-style-type: none"> • field studies [LSS05] • user-centered design and user engineering [Vre03] • participatory design [CdB93] • ATAM (usability) [BKM01] • SALUTA [FvGB03] [FB04] [FvGB04]
adoptability	<ul style="list-style-type: none"> • diffusion of innovations [Rog95] • software technology maturation [RR85] • technology acceptance model [ML04] [Dav89] • task technology fit [DS98] [GT95] • Patterson-Conner technology adoption commitment curve [GRE04] • consumer behavior theory [HL02b]

Table 14: Possible evaluation approaches for tool quality attributes

To simplify tool evaluation, there should be a catalog of proven evaluation techniques. Such a catalog should state, for instance, evaluation context and design; how to conduct the evaluation; and analysis and interpretation of results [KPP⁺02]. While the catalog needs to be grown by researchers in the community, reflecting an increasing body of knowledge, there are a number of evaluation approaches that can serve as starting points. Figure 14 states existing evaluation approaches that might be promising candidates to evaluate the survey's tool requirements. Most of these evaluation approaches have not been applied to reverse engineering tools yet.

conducting a requirements questionnaire: Our survey has used existing literature as a lens to identify requirements. One could now conduct a requirements questionnaire with researchers in the reverse engineering field. This approach would provide a complementary lens to show correlations and to expose discrepancies between the identified requirements of the two lenses.

8.2.2 Tool-Building Approach

As with any approach to software development, the assumptions and implications made by my component-based tool-building approach should be continuously questioned and explored. For the future, possible research directions are as follows:

tool development needs a process: My personal experience, which is further backed up by the survey that I conducted, suggests that few academic tool-building efforts make use of an explicit process.²⁷⁸ This is hard to defend, because building of a tool is a particular instance of building a software system—and the need for a process when building any software system is universally acknowledged! How can this lack of process be explained? Is it a conscious decision of researchers to shun development processes; are suitable processes lacking or have not been identified yet by researchers; or are researchers ignorant²⁷⁹ of the need for a process? I believe that this issue should be further pursued and discussed in the research community.

experimenter handbook: There are various case studies of component-based tool-building. However, these studies focus on different aspects of the development effort and its implications. As a result, they are difficult to compare and make it (unnecessary) hard to identify common patterns.

A more structured approach to conducting case studies is needed. For instance, an experimenter handbook could be developed that guides a case study, covering issues such as

- assessment of host component's existing reverse engineering capabilities (e.g., using Tilley's REEF [Ti100] [Ti198a])
- metrics to gather during tool development
- evaluation criteria for tool assessment

knowledge database: Tool-building experiences are valuable assets that should be archived in a central repository for easy access to interested parties. Repository users should be able to query, for instance, for experiences involving certain technologies, architectures, host components, and tool functionalities. The idea of using a repository to collect knowledge about a certain domain is not new. For example, the eCots portal [MLL⁺03] provides a platform for sharing information on software COTS products and producers with the goal to enable more informed components selection. Similarly, a tool-building knowledge database could lead to more informed decisions in the tool-building process.

²⁷⁸Perhaps surprisingly, only about 50% of organizations follow a software engineering method [RHD02]. The percentage of academic researchers who follow a method for reverse engineering tool-building is probably even lower.

²⁷⁹Armour calls this *2nd Order Ignorance—Lack of Awareness*: “I don't know that I don't know something” [Arm00].

additional case studies: There is no such thing as too many case studies. Each additional case study can deepen our understanding of component-based tool-building. However, since a case study draws time and money, it should be well planned and diligently exercised.

An interesting approach might be to conduct a “family of case studies” that strives to implement the same reverse engineering functionality on top of different host components. For instance, Parnas’ tabular representations [SZP96] could be implemented utilizing Excel, Word, SVG, or Dynamic HTML.

application of lessons learned: My proposed recommendations and lessons learned are synthesized—bottom-up—from tool-building experiences. The obtained results should now be applied—top-down—to other tool-building case studies to prove (or disprove) their merit.

8.3 Parting Thoughts

This dissertation provides a new perspective on the construction of reverse engineering tools. It proposes to leverage existing components as building blocks for tool construction. While there are already existing tools that leverage components, their construction is mostly undocumented and pursued in an ad hoc manner.

What is needed is a more predicable approach that provides processes, techniques, and guidance to tool builders. My approach to tool-building is a first step in that direction with the goal to elevate component-based tool-building beyond the craft stage. To achieve this goal, more documented experiences and case studies are needed; more lessons learned have to be distilled; and more disciplined approaches for tool construction have to be proposed, tried out, and evaluated.

I hope that other researchers will pick up where this dissertation left off. Especially, it would be worthwhile to gain experiences with component-based tool building in other domains besides software engineering. At the very least, this dissertation should have been able to establish that this approach to tool-building is worthwhile to explore further.

Bibliography

- [AB02a] Cecilia Albert and Lisa Brownsword. Evolutionary process for integrating COTS-based systems (EPIC). Technical Report CMU/SEI-2002-TR-005, Software Engineering Institute, Carnegie Mellon University, November 2002. <http://www.sei.cmu.edu/pub/documents/02.reports/pdf/02tr005.pdf>.
- [AB02b] Cecilia Albert and Lisa Brownsword. Evolutionary process for integrating COTS-based systems (EPIC): An overview. Technical Report CMU/SEI-2002-TR-009, Software Engineering Institute, Carnegie Mellon University, July 2002. <http://www.sei.cmu.edu/pub/documents/02.reports/pdf/02tr009.pdf>.
- [AB02c] Cecilia Albert and Lisa Brownsword. Meeting the challenges of commercial-off-the-shelf (COTS) products: The information technology solutions evolution process (ITSEP). In J. Dean and A. Gravel, editors, *1st International Conference on COTS-Based Software Systems (ICCBSS'02)*, volume 2255 of *Lecture Notes in Computer Science*, pages 10–20. Springer-Verlag, 2002.
- [ABB⁺01] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, , and J. Zettel. *Component-Based Product Line Engineering with UML*. The Component Software Series. Addison-Wesley, 2001.
- [ABC⁺94] Steven Atkinson, Paul A. Bailes, Murray Chapman, Martin Chilvers, and Ian Peake. A re-engineering evaluation of software refinery: Architecture, process and technology. *3rd IEEE Symposium on Assessment of Quality Software Development Tools*, pages 191–200, June 1994.
- [ABC00] Chris Abts, Barry W. Boehm, and Elisabeth Clark. COCOTS: A COTS software integration lifecycle cost model—model overview and preliminary data collection findings. Technical Report USC-CSE-2000-501, University of Southern California Center for Software Engineering, March 2000. <http://sunset.usc.edu/TechRpts/Papers/usccse2000-501/usccse2000-501.pdf>.
- [ABLZ00] Colin Atkinson, Joachim Bayer, Oliver Laitenberger, and Jörg Zettel. Component-based software engineering: The KobrA approach. *3rd International Workshop on Component-based Software Engineering (CBSE'00)*, June 2000.
- [Abt02] Chris Abts. COTS-based systems (CBS) functional density—a heuristic for better CBS design. In J. Dean and A. Gravel, editors, *1st International Conference on COTS-Based Software Systems (ICCBSS'02)*, volume 2255 of *Lecture Notes in Computer Science*, pages 1–9. Springer-Verlag, 2002.
- [ACD94] M. Ancona, A. Clematis, and G. Dodero. Reusing a compiler. *9th ACM Symposium on Applied Computing (SAC'94)*, pages 82–87, April 1994.
- [ACT99] G. Antonioli, F. Calzolari, and P. Tonella. Impact of function pointers on the call graph. *3rd IEEE European Conference on Software Maintenance and Reengineering (CSMR'99)*, pages 51–59, March 1999.
- [Adl95] Richard M. Adler. Emerging standards for component software. *IEEE Computer*, 28(3):68–77, March 1995.
- [Ado02a] Adobe SVG Zone. Developer track: Chemical markup language (CML) demo, 2002. <http://www.adobe.com/svg/demos/devtrack/chemical.html>.
- [Ado02b] Adobe Systems. *GoLive 6.0 Extend Script SDK Programmer's Guide: SDK Release 6.0r4*, 2002.
- [Ado02c] Adobe Systems. *GoLive 6.0 Extend Script SDK Programmer's Reference: SDK Release 6.0r4*, 2002.
- [ADOV02] Selma Arbaoui, Jean-Claude Derniame, Flavio Oquendo, and Herve Verjus. A comparative review of process-centered software engineering environments. *Annals of Software Engineering*, 14(1–4):311–340, December 2002.

- [AFL⁺97] G. Antoniol, R. Fiutem, G. Lutteri, P. Tonella, S. Zanfei, and E. Merlo. Program understanding and maintenance with the CANTO environment. *13th IEEE International Conference on Software Maintenance (ICSM'97)*, pages 72–81, October 1997.
- [AFMT95] G. Antoniol, R. Fiutem, E. Merlo, and P. Tonella. Application and user interface migration from BASIC to Visual C++. *11th IEEE International Conference on Software Maintenance (ICSM'95)*, pages 76–85, October 1995.
- [AG96] Darren C. Atkinson and William G. Griswold. The design of whole-program analysis tools. *18th ACM/IEEE International Conference on Software Engineering (ICSE'96)*, pages 16–27, May 1996.
- [AG06] Darren C. Atkinson and William G. Griswold. Effective pattern matching of source code using abstract syntax patterns. *Software—Practice and Experience*, 36(4):413–447, April 2006.
- [AH93] Sujai Asur and Steve Hufnagel. Taxonomy of rapid-prototyping methods and tools. *4th IEEE International Workshop on Rapid System Prototyping (RSP'93)*, pages 42–56, June 1993.
- [AHI⁺05] Pierre America, Dieter Hammer, Mugurel T. Ionita, Henk Obbink, and Eelco Rommes. Scenario-based decision making for architectural variability in product families. *Software Process: Improvement and Practice*, 10(2):171–187, April–June 2005.
- [Ale77] Christopher Alexander. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.
- [ALG⁺03] Alexandre Alvaro, Daniel Lucredio, Vinicius Cardoso Garcia, Antonio Francisco do Prado, and Luis Carlos Trevelin. Orion-RE: A component-based software reengineering environment. *10th IEEE Working Conference on Reverse Engineering (WCRE'03)*, pages 248–259, November 2003.
- [AMM03] Daniel Amyot, Nikolai Mansurov, and Gunter Mussbacher. Understanding existing software with use case map scenarios. In E. Sherratt, editor, *SAM 2002*, volume 2599 of *Lecture Notes in Computer Science*, pages 124–140. Springer-Verlag, 2003.
- [Aoy06] Mikio Aoyama. Co-evolutionary service-oriented model of technology transfer in software engineering. *International Workshop on Software Technology Transfer in Software Engineering (TT'06)*, pages 3–8, May 2006.
- [APMV03] G. Antoniol, M. Di Penta, G. Masone, and U. Villano. XOGastan: XML-oriented gcc AST analysis and transformations. *3rd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'03)*, pages 173–182, September 2003.
- [Arm00] Phillip G. Armour. The five orders of ignorance. *Communications of the ACM*, 43(10):17–20, October 2000.
- [Art03] John Arthorne. Project builders and natures. *Eclipse Corner Articles*, January 2003. <http://www.eclipse.org/articles/Article-Builders/builders.html>.
- [ASRW02] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, and Juhani Warsta. Agile software development methods: Review and analysis. Technical Report Publication 478, VTT Electronics, Finland, 2002. <http://www.inf.vtt.fi/pdf>.
- [ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley, 1986.
- [AT98] M. N. Armstrong and C. Trudeau. Evaluating architectural extractors. *5th IEEE Working Conference on Reverse Engineering (WCRE'98)*, pages 30–39, October 1998.
- [ATW00] Kenneth M. Anderson, Richard N. Taylor, and E. James Whitehead. Chimera: Hypermedia for heterogeneous software development environments. *ACM Transactions on Office Information Systems*, 18(3):211–245, July 2000.

- [AWSR03] Pekka Abrahamsson, Juhani Warsta, Mikko T. Siponen, and Jussi Ronkainen. Directions on agile methods: A comparative analysis. *25th ACM/IEEE International Conference on Software Engineering (ICSE'03)*, pages 244–254, May 2003.
- [BA99] Barry Boehm and Chris Abts. COTS integration: Plug and pray? *IEEE Computer*, 32(1):135–138, January 1999.
- [Bac90] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice Hall, 1990.
- [Bak95] Brenda S. Baker. On finding duplication and near-duplication in large software systems. *2nd IEEE Working Conference on Reverse Engineering (WCRE'95)*, pages 86–95, July 1995.
- [Bak02] Thomas G. Baker. Lessons learned integrating COTS into systems. In J. Dean and A. Gravel, editors, *1st International Conference on COTS-Based Software Systems (ICCBSS'02)*, volume 2255 of *Lecture Notes in Computer Science*, pages 21–30. Springer-Verlag, 2002.
- [Bal01] Francoise Balmas. Displaying dependence graphs: a hierarchical approach. *8th IEEE Working Conference on Reverse Engineering (WCRE'01)*, pages 261–270, October 2001.
- [Bal03] Robert Balzer. Evolution stability through COTS integration. *6th IEEE International Workshop on Principles of Software Evolution (IWPSE'03)*, pages 57–58, September 2003.
- [Bal04] F. Balmas. Displaying dependence graphs: a hierarchical approach. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(3):151–185, 2004.
- [Bar91] Timothy J. Barnes. SKILL: A CAD system extension language. *27th ACM/IEEE Design Automation Conference (DAC'91)*, pages 266–271, January 1991.
- [Bar96] John Barnes. *Programming in Ada95*. Addison-Wesley, 1996.
- [Bas96] Victor R. Basili. The role of experimentation in software engineering: Past, current, and future. *18th ACM/IEEE International Conference on Software Engineering (ICSE'96)*, pages 442–449, 1996.
- [BB98] V. Balasubramanian and Alf Bashian. Document management and web technologies: Alice marries the mad hatter. *Communications of the ACM*, 41(7):107–115, July 1998.
- [BB00] PerOlof Bengtsson and Jan Bosch. An experiment on creating scenario profiles for software change. *Annals of Software Engineering*, 9(1–4):59–78, 2000.
- [BB01] Victor R. Basili and Barry Boehm. COTS-based systems top 10 list. *IEEE Computer*, 34(3):91–93, May 2001.
- [BBB⁺00] Felix Bachmann, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau. Volume II: Technical concepts of component-based software engineering. Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie Mellon University, May 2000. <http://www.sei.cmu.edu/pub/documents/00.reports/pdf/00tr008.pdf>.
- [BBB03] Franck Barbier, Nicolas Belloir, and Jean-Michel Bruel. Incorporation of test functionality into software components. In H. Erdogmus and T. Weng, editors, *2nd International Conference on COTS-Based Software Systems (ICCBSS'03)*, volume 2580 of *Lecture Notes in Computer Science*, pages 25–35. Springer-Verlag, 2003.
- [BBDM01] Cornelia Boldyreff, Elizabeth Burd, Joanna Donkin, and Sarah Marshall. The case for the use of plain english to increase web accessibility. *3rd IEEE International Workshop on Web Site Evolution (WSE'01)*, pages 42–48, November 2001.
- [BBH98] Pearl Brereton, David Budgen, and Geoff Hamilton. Hypertext: The next maintenance mountain. *IEEE Computer*, 31(12):49–55, December 1998.
- [BBZ03] Philipp Bouillon, Martin Burger, and Andreas Zeller. Automated debugging in Eclipse (at the touch of not even a button). *2003 OOPSLA workshop on eclipse technology eXchange*, pages 1–5, October 2003.

- [BCH⁺01] Adam Buchsbaum, Yih-Farn Chen, Huale Huang, Eleftherios Koutsofios, John Mocenigo, Anne Rogers, Michael Jankowsky, and Spiros Mancoridis. Visualizing and analyzing software infrastructures. *IEEE Software*, 18(5):62–70, September/October 2001.
- [BCK98] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison-Wesley, 1998.
- [BCO98] Lisa Brownsword, David Carney, and Tricia Oberndorf. The opportunities and complexities of applying commercial-off-the-shelf components. *CrossTalk*, 11(4):25–30, April 1998. <http://www.stsc.hill.af.mil/crosstalk/frames.asp?uri=1998/04/applying.asp>.
- [BCV00] Luca Bompani, Paolo Ciancarini, and Fabio Vitali. Software engineering and the internet: a roadmap. *Conference on The Future of Software Engineering*, pages 305–315, June 2000.
- [BDDD89] A. Bedetti, R. Daverio, B. DiStefano, and E. Distefano. The GSI road management integrated software package. *IEEE Vehicle Navigation and Information Systems Conference*, pages A22–A27, September 1989.
- [BDG⁺04] Ken Brodli, David Duce, Julian Gallop, Musbah Sagar, Jeremy Walton, and Jason Wood. Visualization in grid computing environments. *IEEE Visualization 2004*, pages 155–162, October 2004.
- [BDH⁺98] Manfred Broy, Anton Deimel, Juergen Henn, Kai Koskimies, Frantisek Plasil, Gustav Pomberger, Wolfgang Pree, Michael Stal, and Clemens Szyperski. What characterizes a (software) component? *Software – Concepts & Tools*, 19(1):49–56, June 1998.
- [BdM04a] Brian Barry and Oege de Moor. Preface. In B. Barry and O. de Moor, editors, *2nd Eclipse Technology Exchange: eTX and the Eclipse Phenomenon (eTX'04)*, volume 107 of *Electronic Notes in Theoretical Computer Science*, pages 1–5. Elsevier, December 2004.
- [BdM04b] Brian Barry and Oege de Moor, editors. *A satellite event of ETAPS 2004: 2nd eclipse Technology Exchange (eTX'04)*, March 2004. <http://web.comlab.ox.ac.uk/oucl/work/oege.de.moor/etxpage/eclipse.html>.
- [BEA01] Jongmoon Baik, Nancy Eickelmann, and Chris Abts. Empirical software simulation for COTS glue code development and integration. *25th IEEE International Computer Software and Applications Conference (COMPSAC'01)*, pages 8–17, September 2001.
- [Bec00] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.
- [Bec02] David Becker. Microsoft gives up some ground. *CNET News.com*, August 2002. <http://news.com.com/2100-1001-955383.html>.
- [BEL⁺03] Mario R. Barbacci, Robert Ellison, Anthony J. Lattanze, Judith A. Stafford, Charles B. Weinstock, and William G. Wood. Quality attribute workshops (QAWs), third edition. Technical Report CMU/SEI-2003-TR-016, Software Engineering Institute, Carnegie Mellon University, August 2003. <http://www.sei.cmu.edu/pub/documents/03.reports/03tr016.pdf>.
- [Ber04] Brian Berenbach. Towards a unified model for requirements engineering. *4th International Workshop on Adoption-Centric Software Engineering (ACSE'04)*, pages 26–29, May 2004.
- [Bes02] Casey Best. Designing a component-based framework for a domain independent visualization tool. Master's thesis, Department of Computer Science, University of Victoria, 2002.
- [Bev99] Nigel Bevan. Quality in use: Meeting user needs for quality. *Journal of Systems and Software*, 49(1):89–96, December 1999.

- [BFBS04] Michael Burke, Bjorn Freeman-Benson, and Margaret Storey, editors. *2004 OOPSLA workshop on eclipse technology eXchange*, October 2004. <http://www.oopsla.org/2004/ShowEvent.do?id=207>.
- [BFK⁺99] Joachim Bayer, Oliver Flege, Peter Knauber, Roland Laqua, Dirk Muthig, Klaus Schmid, Tanya Widen, and Jean-Marc DeBaud. PuLSE: A methodology to develop software product lines. *5th ACM Symposium on Software Reusability (SSR'99)*, pages 122–131, May 1999.
- [BFW92] Alan W. Brown, Peter H. Feiler, and Kurt C. Wallnau. Past and future models of CASE integration. *4th IEEE International Workshop on Computer-Aided Software Engineering (CASE'92)*, pages 36–45, July 1992.
- [BG88] Nathaniel S. Borenstein and James Gosling. Unix emacs: A retrospective—lessons for flexible system design. *1st ACM SIGGRAPH Symposium on User Interface Software*, pages 95–101, October 1988.
- [BG97] Berndt Bellay and Harald Gall. A comparison of four reverse engineering tools. *4th IEEE Working Conference on Reverse Engineering (WCRE'97)*, pages 2–11, October 1997.
- [BG98] Berndt Bellay and Harald Gall. An evaluation of reverse engineering tool capabilities. *Journal of Software Maintenance: Research and Practice*, 10(5):305–331, September/October 1998.
- [BG99] Robert M. Balzer and Neil M. Goldman. Mediating connectors. *International Conference on Distributed Computing Systems Workshops on Electronic Commerce and Web-based Applications/Middleware*, pages 73–77, May 1999.
- [BG00a] Robert M. Balzer and Neil M. Goldman. A COTS based product line architecture for generating design editors. *IFIPS World Computer Conference 2000*, August 2000.
- [BG00b] Robert M. Balzer and Neil M. Goldman. Mediating connectors: a non-bypassable process wrapping technology. *DARPA Information Survivability Conference and Exposition (DISCEX'00)*, pages 361–368, January 2000.
- [BGH99] Ivan T. Bowman, M. W. Godfrey, and Richard C. Holt. Connecting software architecture recovery frameworks. *1st International Symposium on Constructing Software Engineering Tools (CoSET'99)*, May 1999.
- [BGSS92] Kathleen Brade, Mark Guzdial, Mark Steckel, and Elliot Soloway. Whorf: A visualization tool for software maintenance. *IEEE Workshop on Visual Languages*, pages 148–154, September 1992.
- [BGSS03] Michael Burke, Erich Gamma, Gabby Silberman, and Mary Lou Soffa, editors. *2003 OOPSLA workshop on eclipse technology eXchange*, October 2003. <http://oopsla.acm.org/oopsla2003/files/ws-19.html>.
- [BGV90] Robert A. Ballance, Susan L. Graham, and Michael L. Van De Vanter. The Pan language-based editing system for integrated development environments. *4th ACM SIGSOFT Symposium on Software Development Environments (SDE 4)*, pages 77–93, December 1990.
- [BH91] Erich Buss and John Henshaw. A software reverse engineering experience. *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'91)*, pages 55–73, October 1991.
- [BH92] Erich Buss and John Henshaw. Experiences in program understanding. *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'92)*, pages 55–73, November 1992.
- [BH96] Kimin Bao and Ellis Horowitz. A new approach to software tool interoperability. *11th ACM Symposium on Applied Computing (SAC'96)*, pages 500–509, February 1996.
- [BH99] Jules P. Bergmann and Mark A. Horowitz. Vex—a CAD toolbox. *36th ACM/IEEE Conference on Design Automation (DAC'99)*, pages 523–528, June 1999.

- [BHB99] Ivan T. Bowman, Richard C. Holt, and Neil V. Brewster. Linux as a case study: Its extracted software architecture. *21st ACM/IEEE International Conference on Software Engineering (ICSE'99)*, pages 555–563, May 1999.
- [BHJM04] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. An Eclipse plug-in for model checking. *12th IEEE International Workshop on Program Comprehension (IWPC'04)*, pages 251–255, June 2004.
- [Big98] Ted J. Biggerstaff. A perspective of generative reuse. *Annals of Software Engineering*, 5:169–226, 1998.
- [Bir05] Dorian Birsan. On plug-ins and extensible architectures. *ACM Queue*, 3(2):40–46, March 2005.
- [Bis92] Walter R. Bischofberger. Sniff—pragmatic approach to a C++ programming environment. *USENIX C++ conference*, August 1992. <http://www.ubilab.org/publications/bis92a.html>.
- [BJ97] Suresh K. Bhavnani and Bonnie E. John. From sufficient to efficient usage: An analysis of strategic knowledge. *ACM Conference on Human Factors in Computing Systems (CHI'96)*, pages 91–98, March 1997.
- [BJK01] Len Bass, Bonnie E. John, and Jesse Kates. Achieving usability through software architecture. Technical Report CMU/SEI-2001-TR-005, Software Engineering Institute, Carnegie Mellon University, March 2001. <http://www.sei.cmu.edu/pub/documents/01.reports/pdf/01tr005.pdf>.
- [BK01] Sarita Bassil and Rudolf K. Keller. Software visualization tools: Survey and analysis. *9th IEEE International Workshop on Program Comprehension (IWPC'01)*, pages 7–17, May 2001.
- [BK02] David M. Brann and Beth C. Kulick. Simulation of restaurant operations using the restaurant modeling studio. *IEEE Winter Simulation Conference (WSC'02)*, pages 1448–1453, December 2002.
- [BKB02] Len Bass, Mark Klein, and Felix Bachmann. Quality attribute design primitives and the attribute driven design method. In F. van der Linden, editor, *PFE-4 2001*, volume 2290 of *Lecture Notes in Computer Science*, pages 169–186. Springer-Verlag, 2002.
- [BKLW95] Mario Barbacci, Mark. H. Klein, Thomas A. Longstaff, and Charles B. Weinstock. Quality attributes. Technical Report CMU/SEI-95-TR-021, Software Engineering Institute, Carnegie Mellon University, December 1995. <http://www.sei.cmu.edu/pub/documents/95.reports/pdf/95.tr.021.pdf>.
- [BKM01] Len Bass, Mark Klein, and Gabriel Moreno. Applicability of general scenarios to the architecture tradeoff analysis method. Technical Report CMU/SEI-2001-TR-014, Software Engineering Institute, Carnegie Mellon University, October 2001. <http://www.sei.cmu.edu/pub/documents/01.reports/01tr014.pdf>.
- [BL96] Mary Beth Butler and Ericca Lathi. Lotus Notes database support for usability testing. *ACM SIGCHI Conference on Human Factors in Computing Systems (CHI'96)*, pages 23–24, April 1996.
- [BL98] Brian C. Behrens and Reuven R. Levary. Practical legal aspects of software reverse engineering. *Communications of the ACM*, 41(2):27–29, February 1998.
- [BL02] Nelson Baloian and Wolfram Luther. Visualization for the mind's eye. In S. Diehl, editor, *Software Visualization*, volume 2269 of *Lecture Notes in Computer Science*, pages 354–367. Springer-Verlag, 2002.
- [BLBvV04] PerOlof Bengtsson, Nico Lassing, Jan Bosch, and Hans van Vliet. Architecture-level modifiability analysis (ALMA). *Journal of Systems and Software*, 69(1–2):129–147, January 2004.
- [BLM98] Michael Blaha, David LaPlant, and Erica Marvak. Requirements for repository software. *5th IEEE Working Conference on Reverse Engineering (WCRE'98)*, pages 164–173, October 1998.

- [BLM⁺04] Bob Balzer, Marin Litoiu, Hausi Müller, Dennis Smith, Margaret-Anne Storey, Scott Tilley, and Kenny Wong. 4th international workshop on adoption-centric software engineering. *4th International Workshop on Adoption-Centric Software Engineering (ACSE'04)*, pages 1–2, May 2004.
- [BM98] Elisa L. A. Baniassad and Gail C. Murphy. Conceptual module querying for software reengineering. *20th ACM/IEEE International Conference on Software Engineering (ICSE'98)*, pages 64–73, April 1998.
- [BMD⁺00] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lagüe, and Kostas Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. *7th IEEE Working Conference on Reverse Engineering (WCRE'00)*, pages 98–107, November 2000.
- [BML97] Naser S. Barghouti, John Mocenigo, and Wenke Lee. Grappa: A graph package in java. In *5th International Symposium on Graph Drawing*, volume 1353 of *Lecture Notes in Computer Science*, pages 336–343. Springer-Verlag, 1997.
- [BMMM98] William J. Brown, Raphael C. Malveau, Hays W. McCormick, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, 1998.
- [BMN04] Robert Biddle, Angela Martin, and James Noble. No name: Just notes on software reuse. *ACM SIGPLAN Notices*, 38(2):76–96, February 2004.
- [BMS05] Michael Burke, Cheryl Morris, and Margaret-Anne Storey, editors. *Research-Industry Technology Exchange at EclipseCon 2005*, 2005. <http://www.cs.uvic.ca/~mstorey/eclipsecon2005-exchange.html>.
- [BMT04] Carlo Bellettini, Alessandro Marchetto, and Andrea Trentini. WebUml: Reverse engineering of Web applications. *19th ACM Symposium on Applied Computing (SAC'04)*, pages 1662–1669, March 2004.
- [BMW93] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster. The concept assignment problem in program understanding. *15th ACM/IEEE International Conference on Software Engineering (ICSE'93)*, pages 482–498, May 1993.
- [Boe88] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, May 1988.
- [Boe99] Barry Boehm. Making RAD work for your project. *IEEE Computer*, 32(3):113–115, March 1999.
- [Boe00] Barry Boehm. Requirements that handle IKIWISI, COTS, and rapid change. *IEEE Computer*, 33(7):99–102, July 2000.
- [Boo87] Grady Booch. *Software Components with Ada: Structures, Tools, and Subsystems*. Benjamin/Cummings Publishing Company, 1987.
- [Bos99] Jan Bosch. Product-line architectures in industry: A case study. *21st ACM/IEEE International Conference on Software Engineering (ICSE'99)*, pages 544–554, May 1999.
- [BOS00] Lisa Brownsword, Tricia Oberndorf, and Carol A. Sledge. Developing new processes for COTS-based systems. *IEEE Software*, 17(4):48–55, July/August 2000.
- [BOS01] John Bergey, Liam O'Brien, and Dennis Smith. Options analysis for reengineering (oar): A method for mining legacy assets. Technical Note CMU/SEI-2001-TN-013, Software Engineering Institute, Carnegie Mellon University, September 2001. <http://www.sei.cmu.edu/pub/documents/01.reports/pdf/01tn013.pdf>.
- [BP84] Ted J. Biggerstaff and Alan J. Perlis. Foreword. *IEEE Transactions on Software Engineering*, SE-10(5):474–477, September 1984.
- [BP92] Alan W. Brown and Maria H. Penedo. An annotated bibliography on integration in software engineering environments. Special Report CMU/SEI-92-SR-8, Software Engineering Institute, Carnegie Mellon University, May 1992. <http://www.sei.cmu.edu/pub/documents/92.reports/pdf/sr08.92.pdf>.

- [BPM04] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. DMS: Program transformations for practical scalable software evolution. *26th ACM/IEEE International Conference on Software Engineering (ICSE'04)*, pages 625–634, May 2004.
- [BPY⁺03a] Barry Boehm, Dan Port, Ye Yang, Jesal Bhuta, and Chris Abts. Composable process elements for developing COTS-based applications. *International Symposium on Empirical Software Engineering (ISESE'03)*, pages 2–5, September 2003.
- [BPY⁺03b] Barry W. Boehm, Dan Port, Ye Yang, , and Jesal Bhuta. Not all CBS are created equally: COTS-intensive project types. In H. Erdogmus and T. Weng, editors, *2nd International Conference on COTS-Based Software Systems (ICCBSS'03)*, volume 2580 of *Lecture Notes in Computer Science*, pages 36–50. Springer-Verlag, 2003.
- [Bre98] Neil V. Brewster. Creation of a software bookshelf for the Linux operating system. Master's thesis, University of Toronto, April 1998.
- [Bre99] Ansgar Bredenfeld. Co-design tool construction using APICES. *7th ACM International Workshop on Hardware/Software Codesign (CODES'99)*, pages 126–130, 1999.
- [Bre04] Pearl Brereton. The software customer/supplier relationship. *Communications of the ACM*, 47(2):77–81, February 2004.
- [Bri04] David Brickner. 40 million users who don't buy books, November 2004. http://www.oreillynet.com/onlamp/blog/2004/11/40_million_users_who_dont_buy.html.
- [Bro77] Ruven Brooks. Towards a theory of the cognition processes in computer programming. *International Journal of Man-Machine Studies*, 9:737–751, 1977.
- [Bro83] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:543–554, 1983.
- [Bro91] P. Brown. Integrated hypertext and program understanding tools. *IBM Systems Journal*, 30(3):363–392, March 1991.
- [Bro93] Alan W. Brown. Control integration through message-passing in a software development environment. *Software Engineering Journal*, 8(3):121–131, May 1993.
- [Bro94] Alan W. Brown. Why evaluating CASE environments is different from evaluating CASE tools. *3rd IEEE Symposium on Assessment of Quality Software Development Tools*, pages 4–13, June 1994.
- [Bro95] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, Anniversary edition, 1995.
- [Bro04] Andrew Brooks. Results of rapid bottom-up software process modeling. *Software Process: Improvement and Practice*, 9(4):265–278, October–December 2004.
- [BSM02] Casey Best, Margaret-Anne Storey, and Jeff Michaud. Designing a component-based framework for visualization in software engineering and knowledge engineering. *14th ACM/IEEE International Conference on Software Engineering and Knowledge Engineering (SEKE'02)*, pages 323–326, July 2002.
- [BSM⁺03] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework*. Addison-Wesley, December 2003.
- [BT03] Barry Boehm and Richard Turner. Observations on balancing discipline and agility. *Agile Development Conference (ADC'03)*, pages 32–39, 2003.
- [BTMG02] R. Ian Bull, Andrew Trevors, Andrew J. Malton, and Michael W. Godfrey. Semantic grep: Regular expressions + relational abstraction. *9th IEEE Working Conference on Reverse Engineering (WCRE'02)*, pages 267–276, October 2002.
- [Buc00] Jürgen Buchner. Hotdoc: A framework for compound documents. *ACM Computing Surveys*, 32(1es):33–38, March 2000.

- [BW96] Alan W. Brown and Kurt C. Wallnau. Engineering of component-based systems. *2nd IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'96)*, pages 414–422, October 1996.
- [BW98] Alan W. Brown and Kurt C. Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37–46, September/October 1998.
- [BYM⁺98] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant Anna, and Lorraine Bier. Clone detection using abstract syntax trees. *14th IEEE International Conference on Software Maintenance (ICSM'98)*, pages 368–377, November 1998.
- [C⁺98] John Cameron et al. *Architecture Description Standard: Overview*. IBM, document version 1.0 edition, August 1998.
- [Cal81] Italo Calvino. *If on a winter's night a traveler*. Harcourt Brace & Company, 1981.
- [Cam02] John Cameron. Configurable development processes. *Communications of the ACM*, 45(3):72–77, March 2002.
- [Car97] David Carney. Assembling large systems from COTS components: Opportunities, cautions, and complexities. In *SEI Monographs on the Use of Commercial Software in Government Systems*. Software Engineering Institute, Carnegie Mellon University, June 1997.
- [CB99] Patrick P. Cao and Frada V. Burnstein. An asynchronous group decision support system study for intelligent multicriteria decision making. *32nd IEEE Hawaii International Conference on System Sciences (HICSS'99)*, January 1999.
- [CC90] Elliot J. Chikofsky and James H. Cross, II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, January 1990.
- [CC00] Anthony Cox and Charles Clarke. A comparative evaluation of techniques for syntactic level source code analysis. *7th IEEE Asia-Pacific Software Engineering Conference (APSEC'00)*, pages 282–289, December 2000.
- [CC01] Anthony Cox and Charles Clarke. Representing and accessing extracted information. *17th IEEE International Conference on Software Maintenance (ICSM'01)*, pages 12–21, November 2001.
- [CC03] Anthony Cox and Charles Clarke. Syntactic approximation using iterative lexical analysis. *11th IEEE International Workshop on Program Comprehension (IWPC'03)*, pages 154–163, May 2003.
- [CCM05] Olivier Chirouze, David Cleary, and George G. Mitchell. A software methodology for applied research: eXtreme Researching. *Software—Practice and Experience*, 35(15):1441–1454, December 2005.
- [CdB93] Andrew Clement and Peter Van den Besselaar. A retrospective look at PD projects. *Communications of the ACM*, 36(4):29–37, April 1993.
- [CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
- [CEK⁺00a] Jörg Czeransk, Thomas Eisenbarth, Holger M. Kienle, Rainer Koschke, Erhard Plödereeder, Daniel Simon, Yan Zhang, Jean-François Girard, and Martin Würthner. Data exchange in Bauhaus. *7th IEEE Working Conference on Reverse Engineering (WCRE'00)*, pages 293–295, November 2000.
- [CEK⁺00b] Jörg Czeransk, Thomas Eisenbarth, Holger M. Kienle, Rainer Koschke, and Daniel Simon. Analyzing `xfiig` using the Bauhaus tool. *7th IEEE Working Conference on Reverse Engineering (WCRE'00)*, pages 197–199, November 2000.
- [Cen97] IBM Object-Oriented Technology Center. *Developing Object-Oriented Software: An Experience-Based Approach*. Prentice Hall, 1997.
- [Cez95] Ruknet Cezzar. *A Guide to Programming Languages: Overview and Comparison*. Artech House, 1995.

- [CF03] Jeff Carlson and Glenn Fleishman. *Real World Adobe GoLive 6*. Peachpit Press, 2003.
- [CFKW95] Yih-Farn R. Chen, Glenn S. Fowler, Eleftherios Koutsofios, and Ryan S. Wallach. Ciao: A graphical navigator for software and document repositories. *11th IEEE International Conference on Software Maintenance (ICSM'95)*, October 1995.
- [CG95] C. Cifuentes and K. J. Gough. Decompilation of binary programs. *Software—Practice and Experience*, 25(7):811–829, July 1995.
- [CHBM98] James H. Cross, T. Dean Hendrix, Larry A. Barowski, and Karl S. Mathias. Scalable visualizations to support reverse engineering: A framework for evaluation. *5th IEEE Working Conference on Reverse Engineering (WCRE'98)*, pages 201–209, October 1998.
- [Che00] Robert David Cherinka. *Extending Traditional Static Analysis Techniques to Support Development, Testing and Maintenance of Component-Based Solutions*. PhD thesis, Old Dominion University, December 2000.
- [Che03] Li-Te Chen. Breaking out of Eclipse: Developing an ActiveX host for SWT. *2003 OOPSLA workshop on eclipse technology eXchange*, pages 15–19, October 2003.
- [Che06] Yu Chen. Building a software metrics visualization tool using the Visio COTS product. Master's thesis, Department of Computer Science, University of Victoria, 2006.
- [CHK⁺01] Ned Chapin, Joanne E. Hale, Khaled Md. Khan, Juan F. Ramil, and Wui-Gee Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(1):3–30, January/February 2001.
- [CHP00] David Carney, Scott A. Hissam, and Daniel Plakosh. Complex COTS-based software systems: practical steps for their maintenance. *Journal of Software Maintenance: Research and Practice*, 12(6):357–376, November/December 2000.
- [CI02] Pilu Crescenzi and Gaia Innocenti. Towards a taxonomy of network protocol visualization tools. In S. Diehl, editor, *Software Visualization*, volume 2269 of *Lecture Notes in Computer Science*, pages 241–255. Springer-Verlag, 2002.
- [Cif99] Cristina Cifuentes. The impact of copyright on the development of cutting edge binary reverse engineering technology. *6th IEEE Working Conference on Reverse Engineering (WCRE'99)*, pages 66–76, October 1999.
- [CK91] S. R. Chidamber and C. F. Kemerer. Towards a metrics suite for object oriented design. *7th ACM Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'91)*, pages 197–211, October 1991.
- [CK97] Yin-Farn Chen and Eleftherios Koutsofios. WebCiao: A website visualization and tracking system. WebNet, 1997. <http://www.research.att.com:9000/~chen/webciao/>.
- [CK98] S. Jeromy Carriere and Rick Kazman. The perils of reconstructing architectures. *3rd International Software Architecture Workshop (ISAW-3)*, November 1998.
- [CL00a] David Carney and Fred Long. What do you mean by COTS? Finally, a useful answer. *IEEE Software*, 17(2):83–86, March/April 2000.
- [CL00b] Ivica Crnkovic and Magnus Larsson. A case study: Demands on component-based development. *22nd ACM/IEEE International Conference on Software Engineering (ICSE'00)*, pages 22–30, June 2000.
- [CM93] Mariano P. Consens and Alberto O. Mendelzon. Hy+: A hygraph-based query visualization system. *ACM SIGMOD International Conference on Management of Data*, pages 511–516, May 1993.
- [CMR92] Mariano Consens, Alberto Mendelzon, and Arthur Ryman. Visualizing and querying software structures. *14th ACM/IEEE International Conference on Software Engineering (ICSE'92)*, pages 138–156, May 1992.

- [CMW02] Katja Cremer, Andre Marburger, and Bernhard Westfechtel. Graph-based tools for re-engineering. *Journal of Software Maintenance and Evolution: Research and Practice*, 14(4):257–292, July/August 2002.
- [CN01] Paul Clements and Linda M. Northrop. *Software Product Lines: Practices and Patterns*. The SEI Series in Software Engineering. Addison-Wesley, August 2001.
- [CNR90] Yih-Farn Chen, Michael Y. Nishimoto, and C. V. Ramamoorthy. The C information abstraction system. *IEEE Transactions on Software Engineering*, 16(3):325–334, March 1990.
- [CNY95] Lawrence Chung, Brian A. Nixon, and Eric Yu. An approach to building quality into software architecture. *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'95)*, November 1995.
- [Coc00] Alistair Cockburn. Selecting a project's methodology. *IEEE Software*, 17(4):64–71, July/August 2000.
- [Coc03] Alistair Cockburn. *People and Methodologies in Software Development*. PhD thesis, University of Oslo, Norway, February 2003.
- [Coh02] Sholom Cohen. Product line state of the practice report. Technical Note CMU/SEI-2002-TN-017, Software Engineering Institute, Carnegie Mellon University, September 2002. <http://www.sei.cmu.edu/pub/documents/02.reports/pdf/02tn017.pdf>.
- [Coh04] Mike Cohn. *User Stories Applied: For Agile Software Development*. Addison-Wesley, 2004.
- [Con99] Jim Conallen. Modeling Web application architectures with UML. *Communications of the ACM*, 42(10):63–70, October 1999.
- [Coo97] C. Daniel Cooper. ASIS-based code analysis automation. *Ada Letters*, 17(6):65–69, November/December 1997.
- [Coo99] Alan Cooper. *The Inmates Are Running the Asylum*. SAMS, 1999.
- [Coo03] Alan Cooper. The origin of personas. *Cooper Newsletters*, August 2003. http://www.cooper.com/content/insights/newsletters/2003_08/Origin_of_Personas.asp.
- [COP03] David J. Carney, Patricia A. Oberndorf, and Patrick R.H. Place. A basis for an assembly process for COTS-based systems (APCS). Technical Report CMU/SEI-2003-TR-010, Software Engineering Institute, Carnegie Mellon University, May 2003. <http://www.sei.cmu.edu/pub/documents/03.reports/pdf/03tr010.pdf>.
- [Cor89] T. A. Corbi. Program understanding: Challenges for the 1990s. *IBM Systems Journal*, 28(2):294–306, February 1989.
- [COR98] R. Cherinka, C. M. Overstreet, and J. Ricci. Maintaining a COTS integrated solution—are traditional static analysis techniques sufficient for this new programming methodology? *14th IEEE International Conference on Software Maintenance (ICSM'98)*, pages 160–169, November 1998.
- [Cor01] Microsoft Corporation. *Developing Microsoft Visio Solutions*, , 2001. Microsoft Press, 2001.
- [Cor03] James R. Cordy. Comprehending reality—practical barriers to industrial adoption of software maintenance automation. *11th IEEE International Workshop on Program Comprehension (IWPC'03)*, pages 196–206, May 2003.
- [Cox89] Brad J. Cox. Planning for software manufacturing. *13th IEEE Computer Software and Applications Conference (COMPSAC'89)*, pages 331–332, September 1989.
- [Cox90] Brad J. Cox. Planning the software industrial revolution. *IEEE Software*, 7(6):25–33, November 1990.

- [CRCK98] John M. Carroll, Mary Beth Rosson, George Chin, and Jürgen Koenemann. Requirements development in scenario-based design. *IEEE Transactions on Software Engineering*, 24(12):1156–1170, December 1998.
- [Crn02] Ivica Crnkovic. Component-based software engineering—new challenges in software development. *Software Focus*, 2(4):127–133, Winter 2002.
- [Crn04] Ivica Crnkovic. Component-based approach for embedded systems. *9th International Workshop on Component-Oriented Programming (WCOP'04)*, June 2004. <http://research.microsoft.com/~cszypers/events/WCOP2004/18-Crnkovic.pdf>.
- [CRW98] R. Clayton, S. Rugaber, and L. Wills. On the knowledge required to understand programs. *5th IEEE Working Conference on Reverse Engineering (WCRE'98)*, pages 69–78, October 1998.
- [CS98] David W. Coppit and Kevin J. Sullivan. Unwrapping. Technical Report CS-98-08, University of Virginia, May 1998. <http://www.cs.virginia.edu/~techrep/CS-98-08.ps.Z>.
- [CS99] Michael A. Copenhafer and Kevin J. Sullivan. Exploration harnesses: Tool-supported interactive discovery of commercial component properties. *14th IEEE Conference on Automated Software Engineering (ASE'99)*, pages 7–14, October 1999.
- [CS00a] David Coppit and Kevin J. Sullivan. Galileo: A tool built from mass-market applications. *22nd ACM/IEEE International Conference on Software Engineering (ICSE'00)*, pages 750–753, June 2000.
- [CS00b] David Coppit and Kevin J. Sullivan. Multiple mass-market applications as components. *22nd ACM/IEEE International Conference on Software Engineering (ICSE'00)*, pages 273–282, June 2000.
- [CS03] David Coppit and Kevin J. Sullivan. Sound methods and effective tools for engineering modeling and analysis. *25th ACM/IEEE International Conference on Software Engineering (ICSE'03)*, pages 198–207, May 2003.
- [CST05] Richard Cardone, Danny Soroker, and Alpna Tiwari. Using XForms to simplify Web programming. *14th ACM International Conference on World Wide Web (WWW'05)*, pages 215–224, May 2005.
- [CT01] Philippe Chevalley and Pascale Thevenod-Fosse. Automated generation of statistical test cases from UML state diagrams. *25th IEEE International Computer Software and Applications Conference (COMPSAC'01)*, pages 205–214, September 2001.
- [Cur02] Charles Curley. Emacs: the free software IDE. *Linux Journal*, June 2002. <http://www.linuxjournal.com/article/5765>.
- [CW98] David J. Carney and Kurt C. Wallnau. A basis for evaluation of commercial software. *Information and Software Technology*, 40(14):851–860, December 1998.
- [DA00] Kathi Hogshead Davis and Peter H. Aiken. Data reverse engineering: A historical survey. *7th IEEE Working Conference on Reverse Engineering (WCRE'00)*, pages 70–78, November 2000.
- [Dav89] D. Davis. Perceived usefulness, perceived ease of use and user acceptance of information technology. *MIS Quarterly*, 13(3):319–342, September 1989.
- [Dav01] Kathi Hogshead Davis. Lessons learned in data reverse engineering. *8th IEEE Working Conference on Reverse Engineering (WCRE'01)*, pages 323–327, October 2001.
- [dCBB⁺93] Dennis de Champeaux, Andrew J. Baer, Brian Bernsen, Alan R. Kroncoff, Tim Korson, and Daniel S. Tkach. Strategies for object-oriented technology transfer (panel). *8th ACM Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'93)*, pages 437–447, September 1993.
- [DDG⁺96] Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers. Vortex: An optimizing compiler for object-oriented languages. *11th ACM Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'96)*, pages 83–100, October 1996.

- [Dei98] Anton Deimel. The SAP R/3 business framework. *Software – Concepts & Tools*, 19(1):29–36, June 1998.
- [DeI97] Chrysanthos Dellarocas. Toward a design handbook for integrating software components. *5th International Symposium on Assessment of Software Tools and Technologies*, pages 3–13, June 1997.
- [DeL99] Robert DeLine. A catalog of techniques for resolving packaging mismatch. *5th ACM Symposium on Software Reusability (SSR'99)*, pages 44–53, May 1999.
- [DeL01] Robert DeLine. Avoiding packaging mismatch with flexible packaging. *IEEE Transactions on Software Engineering*, 27(2):124–143, February 2001.
- [Der03] Charles Derby. Knowledge management for engineers. *IEEE KIMAS'03*, pages 760–765, October 2003.
- [Dev99a] Premkumar T. Devanbu. GENOA—a customizable, front-end-retargetable source code analysis framework. *ACM Transactions on Software Engineering and Methodology*, 8(2):177–212, April 1999.
- [Dev99b] Premkumar T. Devanbu. Re-targetability in software tools. *ACM SIGAPP Applied Computing Review*, 7(3):19–26, Fall 1999.
- [DFGK03] L. Davis, D. Flagg, R. Gamble, and C. Karatas. Classifying interoperability conflicts. In H. Erdogmus and T. Weng, editors, *2nd International Conference on COTS-Based Software Systems (ICCBSS'03)*, volume 2580 of *Lecture Notes in Computer Science*, pages 62–71. Springer-Verlag, 2003.
- [DG04] L. Davis and R. Gamble. Understanding services for integration management. In R. Kazman and D. Ports, editors, *3rd International Conference on COTS-Based Software Systems (ICCBSS'04)*, volume 2959 of *Lecture Notes in Computer Science*, pages 84–93. Springer-Verlag, 2004.
- [DGP⁺01] L. Davis, R. Gamble, J. Payton, G. Jonsdottir, and D. Underwood. A notation for problematic architecture interactions. *8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-9)*, pages 132–141, September 2001.
- [DGP02] L. Davis, R. F. Gamble, and J. Payton. The impact of component architecture on interoperability. *Journal of Systems and Software*, 61(1):31–45, March 2002.
- [dJ04] Merijn de Jonge. Decoupling source trees into build-level components. In Jan Bosch and Charles Krueger, editors, *Software Reuse: Methods, Techniques, and Tools (ICSR'04)*, volume 3107 of *Lecture Notes in Computer Science*, pages 215–231. Springer-Verlag, 2004.
- [dJK03] Hayco A. de Jong and Paul Klint. ToolBus: The next generation. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Formal Methods for Components and Objects*, *Lecture Notes in Computer Science*, pages 220–241. Springer-Verlag, 2852 edition, 2003.
- [DKJ05] Tore Dyba, Barbara A. Kitchenham, and Magne Jorgensen. Evidence-based software engineering for practitioners. *IEEE Software*, 22(1):58–65, January/February 2005.
- [DLDPAC01] G. A. Di Lucca, M. Di Penta, G. Antoniol, and G. Casazza. An approach for reverse engineering of web-based applications. *8th IEEE Working Conference on Reverse Engineering (WCRE'01)*, pages 231–240, October 2001.
- [DLFP⁺02a] G. A. Di Lucca, A. R. Fasolino, F. Pace, P. Tramontana, and U. De Carlini. Comprehending web applications by a clustering based approach. *10th IEEE International Workshop on Program Comprehension (IWPC'02)*, pages 261–270, June 2002.
- [DLFP⁺02b] G. A. Di Lucca, A. R. Fasolino, F. Pace, P. Tramontana, and U. De Carlini. WARE: a tool for the reverse engineering of Web applications. *6th IEEE European Conference on Software Maintenance and Reengineering (CSMR'02)*, pages 241–250, March 2002.

- [DLFT02] G. A. Di Lucca, A. R. Fasolino, and P. Tramontana. Towards a better comprehensibility of Web applications: Lessons learned from reverse engineering experiments. *4th IEEE International Workshop on Web Site Evolution (WSE'02)*, pages 33–42, October 2002.
- [DLT00] Stephane Ducasse, Michele Lanza, and Sander Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. *International Symposium on Constructing Software Engineering Tools (COSET'00)*, June 2000.
- [DLT01] Stephane Ducasse, Michele Lanza, and Sander Tichelaar. The moose reengineering environment. *Smalltalk Chronicles*, 3(2), August 2001.
- [DMH01] Thomas R. Dean, Andrew J. Malton, and Ric Holt. Union schemas as a basis for a C++ extractor. *8th IEEE Working Conference on Reverse Engineering (WCRE'01)*, pages 59–67, October 2001.
- [DRD99] Stephane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. *15th IEEE International Conference on Software Maintenance (ICSM'99)*, pages 109–118, August 1999.
- [DS98] M. Dishaw and D. Strong. Supporting software maintenance with software engineering tools: A computed task-technology fit analysis. *Journal of Systems and Software*, 44(2):107–120, December 1998.
- [DS05] D. A. Duce and M. Sagar. skML: A markup language for distributed collaborative visualization. *3rd Theory and Practice of Computer Graphics (TPCG'05)*, pages 171–178, June 2005.
- [dSRC⁺04] Cleidson R. B. de Souza, David Redmiles, Li-Te Cheng, David Millen, and John Patterson. How a good software practice thwarts collaboration—the multiple roles of APIs in software development. *12th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-12)*, pages 221–230, October 2004.
- [DT03] S. Ducasse and S. Tichelaar. Dimensions of reengineering environment infrastructures. *Journal of Software Maintenance and Evolution: Research and Practice*, 15(5):345–373, September/October 2003.
- [DvK87] E. M. Dusink and J. van Katwijk. Reflections on reusable software and software components. *ACM Ada-Europe International Conference*, pages 113–126, 1987.
- [DZS97] Robert DeLine, Gregory Zelesnik, and Mary Shaw. Lessons on converting batch systems to support interaction. *19th ACM/IEEE International Conference on Software Engineering (ICSE'97)*, pages 195–204, May 1997.
- [EB01] Alexander Egyed and Robert Balzer. Unfriendly COTS integration—instrumentation and interfaces for improved plugability. *16th International Conference of Automated Software Engineering (ASE'01)*, pages 223–231, November 2001.
- [Eds00] D. A. Edson. *Professional Development with Visio 2000*. SAMS, 2000.
- [Edw99] Stephen H. Edwards. The state of reuse: Perceptions of the reuse community. *ACM SIGSOFT Software Engineering Notes*, 24(3):32–36, May 1999.
- [EGH⁺05] D. Edelsohn, W. Gellerich, M. Hagog, D. Naishlos, M. Namolaru, E. Pasch, H. Penner, U. Weigand, and A. Zaks. Contributions to the gnu compiler collection. *IBM Systems Journal*, 44(2):259–278, February 2005.
- [Egy00] Alexander Franz Egyed. *Heterogeneous View Integration and its Automation*. PhD thesis, University of Southern California, August 2000.
- [Egy02] Alexander Egyed. Automated abstraction of class diagrams. *ACM Transactions on Software Engineering and Methodology*, 11(4):449–491, October 2002.
- [EHMS05] Michael Eichberg, Michael Haupt, Mira Mezini, and Thorsten Schäfer. Comprehensive software understanding with Sextant. *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 315–324, September 2005.

- [EJB04] Alexander Egyed, Sven Johann, and Robert Balzer. Data and state synchronicity problems while integrating COTS software into systems. *4th International Workshop on Adoption-Centric Software Engineering (ACSE'04)*, pages 69–74, May 2004.
- [EK99] Alexander Egyed and Philippe B. Kruchten. Rose/Architect: A tool to visualize architecture. *32rd IEEE Hawaii International Conference on System Sciences (HICSS'99)*, January 1999.
- [EKW99] Jürgen Ebert, Bernt Kullbach, and Andreas Winter. GraX—an interchange format for reengineering tools. *6th IEEE Working Conference on Reverse Engineering (WCRE'99)*, pages 89–98, 1999.
- [Elv05] Robert Douglas Elves. Navtracks—helping developers navigate source code. Master's thesis, Department of Computer Science, University of Victoria, 2005.
- [EMG00] A. Egyed, N. Modvidovic, and C. Gacek. Component-based perspective on software mismatch detection and resolution. *IEE Proceedings – Software*, 147(6):225–236, December 2000.
- [EMP05] Alexander Egyed, Hausi A. Müller, and Dwayne E. Perry. Integrating COTS into the development process. *IEEE Software*, 22(4):16–18, July/August 2005.
- [ES98] K. Erdos and Harry M. Sneed. Partial comprehension of complex programs (enough to perform maintenance). *6th IEEE International Workshop on Program Comprehension (IWPC'98)*, pages 98–105, June 1998.
- [ESS92] Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner. Seesoft—a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, November 1992.
- [ET94] Graham Ewart and Marijana Tomic. Experiences using reverse engineering techniques to analyse documentation. *3rd IEEE Workshop on Program Comprehension (WPC'94)*, pages 54–61, November 1994.
- [EW01] Alexander Egyed and Dave Wile. Statechart simulator for modeling architectural dynamics. *2nd IEEE/IFIP Working Conference on Software Architecture (WICSA'01)*, pages 87–96, August 2001.
- [Fal97] Mike Falkner. *Using Lotus Notes as an Intranet*. John Wiley & Sons, 1997.
- [FATM99] Roberto Fiutem, Giulio Antoniol, Paolo Tonella, and Ettore Merlo. ART: An architectural reuse engineering environment. *Journal of Software Maintenance: Research and Practice*, 11(5):339–364, September/October 1999.
- [Fav01] Jean-Marie Favre. G^{SEE} : a generic software exploration environment. *9th IEEE International Workshop on Program Comprehension (IWPC'01)*, pages 233–244, May 2001.
- [Fav02] Jean-Marie Favre. A new approach to software exploration: Back-packing with G^{SEE} . *6th IEEE European Conference on Software Maintenance and Reengineering (CSMR'02)*, pages 251–262, March 2002.
- [FB04] Eelke Folmer and Jan Bosch. Architecting for usability: a survey. *Journal of Systems and Software*, 70(1-2):61–78, February 2004.
- [FBG02] Rudolf Ferenc, Arpad Beszedes, and Tibor Gyimothy. Fact extraction and code auditing with Columbus and SourceAudit. *20th IEEE International Conference on Software Maintenance (ICSM'04)*, page 513, September 2002.
- [FBLL05] Rudolf Ferenc, Arpad Beszedes, Lajos Lülöp, and Janos Lele. Design pattern mining enhanced by machine learning. *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 295–304, September 2005.
- [FBTG02] Rudolf Ferenc, Arpad Beszedes, Mikko Tarkiainen, and Tibor Gyimothy. Columbus—reverse engineering tool an schema for C++. *18th IEEE International Conference on Software Maintenance (ICSM'02)*, pages 172–181, October 2002.

- [FD04] J. Froehlich and P. Dourish. Unifying artifacts and activities in a visual tool for distributed software development. *26th ACM/IEEE International Conference on Software Engineering (ICSE'04)*, pages 387–396, May 2004.
- [Fer01] Jon Ferraiolo. *Scalable Vector Graphics (SVG) 1.0 Specification*. W3C, September 2001. <http://www.w3.org/TR/2001/REC-SVG-20010904/>.
- [Fer04] Rudolf Ferenc. *Modelling and Reverse Engineering C++ Source Code*. PhD thesis, Department of Software Engineering, University of Szeged, November 2004.
- [Fer05] Rudolf Ferenc. Transformations among the three levels of schemas for reverse engineering. *Dagstuhl Seminar 05161, Schloss Dagstuhl*, April 2005. <http://www.dagstuhl.de/05161/Materials/>.
- [FES03] Jean-Marie Favre, Jacky Estublier, and Remy Sanlaville. Tool adoption issues in a very large software company. *3rd International Workshop on Adoption-Centric Software Engineering (ACSE'03)*, pages 81–89, May 2003.
- [FG90] Gerhard Fischer and Andreas Girgensohn. End-user modifiability in design environments. *ACM SIGCHI Conference on Human Factors in Computing Systems (CHI'90)*, pages 183–191, April 1990.
- [FGBS04] D. Flagg, R. Gamble, R. Baird, and W. Stewart. Migrating application integrations. In R. Kazman and D. Ports, editors, *3rd International Conference on COTS-Based Software Systems (ICCBSS'04)*, volume 2959 of *Lecture Notes in Computer Science*, pages 94–103. Springer-Verlag, 2004.
- [FHA99] M. Forsell, V. Halttunen, and J. Ahonen. Evaluation of component-based software development methodologies. *Fenno-Ugric Symposium on Software Technology (FUSST'99)*, pages 53–63, August 1999.
- [FHK⁺97] P. J. Finnigan, R. C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. A. Müller, J. Mylopoulos, S. G. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, April 1997.
- [FK00] Anthony Finkelstein and Jeff Kramer. Software engineering: a roadmap. *Conference on The Future of Software Engineering*, pages 5–22, June 2000.
- [FM03] Xavier Franch and N. A. M. Maiden. Modelling component dependencies to inform their selection. In H. Erdogmus and T. Weng, editors, *2nd International Conference on COTS-Based Software Systems (ICCBSS'03)*, volume 2580 of *Lecture Notes in Computer Science*, pages 81–91. Springer-Verlag, 2003.
- [FN00] Norman Fenton and Martin Neil. Software metrics: Roadmap. *Conference on The Future of Software Engineering*, pages 359–370, June 2000.
- [For02] Marko Forsell. *Improving Component Reuse in Software Development*. PhD thesis, University of Jyväskylä, March 2002.
- [Fow03] Martin Fowler. The new methodology. *MartinFowler.com*, April 2003. <http://www.martinfowler.com/articles/newMethodology.html>.
- [FS97] Mahamed E. Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, October 1997.
- [FSH⁺01] Rudolf Ferenc, Susan Elliott Sim, Richard C. Holt, Rainer Koschke, and Tibor Gyimóthy. Towards a standard schema for C/C++. *8th IEEE Working Conference on Reverse Engineering (WCRE'01)*, pages 49–58, October 2001.
- [FvGB03] Eelke Folmer, Jilles van Gurp, and Jan Bosch. Scenario-based assessment of software architecture usability. *25th ACM/IEEE International Conference on Software Engineering (ICSE'03)*, pages 61–68, May 2003.
- [FvGB04] Eelke Folmer, Jilles van Gurp, and Jan Bosch. Software architecture analysis of usability. In R. Bastide, P. Palanque, and J. Roth, editors, *EHCI-DSVIS 2004*, volume 3425 of *Lecture Notes in Computer Science*, pages 38–58. Springer-Verlag, 2004.

- [FW94] Michael Fröhlich and Mattias Werner. The graph visualization system daVinci—a user interface for applications. Technical Report 5/94, Department of Computer Science, University of Bremen, 1994. <ftp://ftp.tzi.de/tzi/biss/daVinci/papers/techrep0594.ps.gz>.
- [GA95] William G. Griswold and Darren C. Atkinson. Managing design trade-offs for a program understanding and transformation tool. *Journal of Systems and Software*, 30(1–2):99–116, July–August 1995.
- [GAB⁺03] Michele Galic, Jonathan Adams, Jon A. Bell, Richard Disney, Ville-Mikko Kanerva, Steve Matulevich, Kent Rebman, and Philippe Spaas. *Patterns: Applying Pattern Approaches*. Patterns for e-business Series. IBM Redbooks, July 2003.
- [GAM96] William G. Griswold, Darren C. Atkinson, and Collin McCurdy. Fast, flexible syntactic pattern matching and processing. *4th IEEE Workshop on Program Comprehension (WPC'96)*, pages 144–153, March 1996.
- [GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. *17th ACM/IEEE International Conference on Software Engineering (ICSE'95)*, pages 179–185, April 1995.
- [Gar05] Jesse James Garrett. Ajax: A new approach to Web applications. *Adaptive Path*, February 2005. <http://www.adaptivepath.com/publications/essays/archives/000385.php>.
- [GB95] V. Scott Gordon and James M. Bieman. Rapid prototyping: Lessons learned. *IEEE Software*, 12(1):85–95, January 1995.
- [GB99] Neil M. Goldman and Robert M. Balzer. The ISI visual design editor generator. *IEEE Symposium on Visual Languages (VL'99)*, pages 20–27, September 1999.
- [GB04] Erich Gamma and Kent Beck. *Contributing to eclipse*. Addison-Wesley, 2004.
- [GBN98] Robert C. Martin Grady Booch and James Newkirk. *Object Oriented Analysis and Design with Applications*. Addison-Wesley, second edition, September 1998.
- [GBP02] Mark Grechanik, Don Batory, and Dewayne E. Perry. Integrating and reusing GUI-driven application. *7th International Conference on Software Reuse (ICSR-7)*, pages 1–16, April 2002. <http://www.cs.utexas.edu/users/gmark/Publications.html>.
- [GBP04] Mark Grechanik, Don Batory, and Dewayne E. Perry. Design of large-scale polylingual systems. *26th ACM/IEEE International Conference on Software Engineering (ICSE'04)*, pages 357–366, May 2004.
- [GCH83] John D. Gould, John Conti, and Todd Hovanyecz. Composing letters with a simulated listening typewriter. *Communications of the ACM*, 26(4):295–308, April 1983.
- [GG05] Alan Grosskurth and Michael W. Godfrey. A reference architecture for web browsers. *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 661–664, September 2005.
- [GGB03] Bengt Göransson, Jan Gulliksen, and Inger Boivie. The usability design process—integrating user-centered systems design in the software development process. *Software Process: Improvement and Practice*, 8(2):111–131, April–June 2003.
- [GGS⁺02] Caroline P. Graettinger, Suzanne Garcia, Jeannine Sivi, Robert J. Schenk, and Peter J. Van Syckle. Using the technology readiness levels scale to support technology management in the DoD's ATD/STO environments. Special Report CMU/SEI-2002-SR-027, Software Engineering Institute, Carnegie Mellon University, September 2002. <http://www.sei.cmu.edu/pub/documents/02.reports/pdf/02sr027.pdf>.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

- [GJK⁺03] Carsten Gutwenger, Michael Jünger, Karsten Klein, Joachim Kupke, Sebastian Leipert, and Petra Mutzel. A new approach for visualizing UML class diagrams. *ACM Symposium on Software Visualization (SoftVis'03)*, pages 179–188, June 2003.
- [GKM05a] Grace Gui, Holger M. Kienle, and Hausi A. Müller. REGoLive: Adding web site comprehension to adobe GoLive. *ICSM 2005 Poster Proceedings*, pages 24–26, September 2005.
- [GKM05b] Grace Gui, Holger M. Kienle, and Hausi A. Müller. REGoLive: Building a web site comprehension tool by extending GoLive. *7th IEEE International Symposium on Web Site Evolution (WSE'05)*, pages 46–53, September 2005.
- [GKM05c] Grace Gui, Holger M. Kienle, and Hausi A. Müller. REGoLive: Web site comprehension with viewpoints. *13th IEEE International Workshop on Program Comprehension (IWPC'05)*, pages 161–164, May 2005.
- [GKS97] Jean-François Girard, Rainer Koschke, and Georg Schied. Comparison of abstract data type and abstract state encapsulation detection techniques for architectural understanding. *4th IEEE Working Conference on Reverse Engineering (WCRE'97)*, pages 66–75, October 1997.
- [Gla04] Robert L. Glass. Matching methodology to problem domain. *Communications of the ACM*, 47(5):19–21, May 2004.
- [GLG03] B. Göransson, M. Lif, and J. Gulliksen. Usability design—extending rational unified process with a new discipline. *10th International Workshop on Interactive Systems: Design, Specification and Verification (DSV-IS'03)*, pages 316–330, June 2003.
- [GLGB05] Jiang Guo, Yuehong Liao, Jeff Gray, and Barrett Bryant. Using connectors to integrate software components. *12th International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05)*, pages 11–18, April 2005.
- [GMH00] J. Gundy, W. Mugridge, and J. Hosking. Constructing component-based software engineering environments: issues and experiences. *Information and Software Technology*, 42(2):103–114, January 2000.
- [GN92] Michelle Gantt and Bonnie A. Nardi. Gardeners and gurus: Patterns of cooperation among CAD users. *ACM SIGCHI Conference on Human Factors in Computing Systems (CHI'92)*, pages 107–117, May 1992.
- [GN00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software—Practice and Experience*, 30(11):1203–1233, September 2000.
- [God01] Michael W. Godfrey. Practical data exchange for reverse engineering frameworks: Some requirements, some experience, some headaches. *ACM SIGSOFT Software Engineering Notes*, 26(1):50–52, January 2001.
- [Gol00] Neil M. Goldman. Smiley—an interactive tool for monitoring inter-module function calls. *8th IEEE International Workshop on Program Comprehension (IWPC'00)*, pages 109–118, June 2000.
- [Gol04] Bernard Golden. *Succeeding with Open Source*. Addison-Wesley, 2004.
- [GP96] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131–174, June 1996.
- [GPB04] Mark Grechanik, Dewayne E. Perry, and Don Batory. Reengineering large-scale polylingual systems. *International Workshop on Incorporating COTS-Software into Software Systems: Tools and Techniques (IWICSS'04)*, pages 22–32, February 2004. <http://www.tuisr.utulsa.edu/iwicss/>.
- [GPG04] Thomas Gschwind, Martin Pinzger, and Harald Gall. TUNalyzer—analyzing templates in C++ code. *11th IEEE Working Conference on Reverse Engineering (WCRE'04)*, pages 48–57, November 2004.

- [Gra87] David F. Gray. A microprocessor development environment based on the Amsterdam Compiler Kit, Emacs and Unix. *ACM SIGCSE Bulletin*, 19(4):30–35, December 1987.
- [Gra95] Ian Graham. *Migrating to Object Technology*. Addison-Wesley, 1995.
- [Gra01] Ian Graham. *Object-Oriented Methods: Principles and Practice*. Object Technology Series. Addison-Wesley, 3rd edition, 2001.
- [Gre87] Williams Gregg. Hypercard: Hypercard extends the Macintosh user interface and makes everybody a programmer. *Byte*, pages 109–117, December 1987.
- [GRE04] Suzanne Garcia, John Robert, and Len Estrin. Managed technology adoption risk: A way to realize better return from COTS investments. In R. Kazman and D. Ports, editors, *3rd International Conference on COTS-Based Software Systems (ICCBSS'04)*, volume 2959 of *Lecture Notes in Computer Science*, pages 74–83. Springer-Verlag, 2004.
- [Gri98] Martin L. Griss. Architecting for large-scale systematic component reuse. *26th IEEE International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 26)*, pages 8–16, August 1998.
- [Gri01] Martin L. Griss. Product-line architectures. In George T. Heineman and William T. Councill, editors, *Component-Based Software Engineering: Putting the Pieces Together*, chapter 22, pages 405–419. Addison-Wesley, 2001.
- [Gru91] Jonathan Grudin. Interactive systems: Bridging the gaps between developers and users. *IEEE Computer*, 24(4):59–69, April 1991.
- [Gru94] Jonathan Grudin. Groupware and social dynamics: Eight challenges for developers. *Communications of the ACM*, 37(1):92–105, January 1994.
- [GT95] D. Goodhue and R. Thompson. Task-technology fit and individual performance. *MIS Quarterly*, 19(2):213–236, June 1995.
- [Gue04] Yann-Gael Gueheneuc. A reverse engineering tool for precise class diagrams. *Conference of The Centre for Advanced Studies On Collaborative Research (CASCON'04)*, pages 28–41, October 2004.
- [Gui05] Grace Gui. Extending a Web authoring tool for Web site reverse engineering. Master's thesis, Department of Computer Science, University of Victoria, 2005.
- [GV95] Robert L. Glass and Iris Vessey. Contemporary application-domain taxonomies. *IEEE Software*, 12(4):63–76, July 1995.
- [GV98] Robert L. Glass and Iris Vessey. Focusing on the application domain: Everyone agrees it's vital, but who's doing anything about it? *31st IEEE Hawaii International Conference on System Sciences (HICSS'98)*, pages 187–196, January 1998.
- [GVR02] R.L. Glass, I. Vessey, and V. Ramesh. Research in software engineering: an analysis of the literature. *Information and Software Technology*, 44(8):491–506, June 2002.
- [GW99] Philip Gray and Ray Welland. Increasing the flexibility of modelling tools via constraint-based specification. *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'99)*, November 1999.
- [GZ05] Ian Gorton and Iming Zhu. Tool support for just-in-time architecture reconstruction and evaluation: An experience report. *27th ACM/IEEE International Conference on Software Engineering (ICSE'05)*, pages 514–523, May 2005.
- [Hal99] Joseph N. Hall. Perl: Internet duct tape. *IEEE Internet Computing*, 3(4):95–96, July-August 1999.
- [Ham97] Jennifer Hamilton. Montana smart pointers: They're smart, and they're pointers. *3rd USENIX Conference on Object-Oriented Technologies and Systems (COOTS'97)*, pages 21–39, June 1997.

- [Hau05] Peter Haumer. Ibm rational method composer: Part 1: Key concepts. *IBM developerWorks*, December 2005. www.ibm.com/developerworks/rational/library/dec05/haumer.
- [HBCC99] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, July 1999.
- [HC99] Scott A. Hissam and David Carney. Isolating faults in complex COTS-based systems. *Journal of Software Maintenance: Research and Practice*, 11(3):183–199, May/June 1999.
- [HC01] George T. Heineman and William T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.
- [HD02] Christopher Hundhausen and Sarah Douglas. A language and system for constructing and presenting low fidelity algorithm visualizations. In S. Diehl, editor, *Software Visualization*, volume 2269 of *Lecture Notes in Computer Science*, pages 227–240. Springer-Verlag, 2002.
- [HEH⁺95] J-L. Hainaut, V. Englebert, J. Henrard, J-M. Hick, and D. Roland. Requirements for information system reverse engineering support. *2nd IEEE Working Conference on Reverse Engineering (WCRE'95)*, pages 136–145, July 1995.
- [HEH⁺96] Jean-Luc Hainaut, Vincent Englebert, Jean Henrard, Jean-Marc Hick, and Didier Roland. Database reverse engineering: From requirements to care tools. *Journal of Automated Software Engineering*, 3(1/2):9–45, July 1996.
- [Hei01] Lauren Heinz. TransPlant: Helping organizations to make the transition. *news@sei*, 4(4), 2001. <http://sei.cmu.edu/news-at-sei/features/2001/4q01/feature-4-4q01.html>.
- [Hen98] Juergen Henn. IBM San Francisco: Object-oriented infrastructure and reusable business components for distributed, multi-platform business applications—implemented entirely in Java. *Software – Concepts & Tools*, 19(1):37–48, June 1998.
- [Hen00] John L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.
- [Hen01] Brian Henderson-Sellers. An OPEN process for component-based development. In George T. Heineman and William T. Councill, editors, *Component-Based Software Engineering: Putting the Pieces Together*, chapter 18, pages 321–340. Addison-Wesley, 2001.
- [HF02] John F. Hopkins and Paul A. Fishwick. The *rube* framework for personalized 3-d software visualization. In S. Diehl, editor, *Software Visualization*, volume 2269 of *Lecture Notes in Computer Science*, pages 368–380. Springer-Verlag, 2002.
- [HGK⁺06] M. Hepner, R. Gamble, M. Kelkar, L. Davis, and D. Flagg. Patterns of conflict among software components. *Journal of Systems and Software*, 79(4):537–551, April 2006.
- [HH00] Ahmed E. Hassan and Richard C. Holt. A reference architecture for Web servers. *7th IEEE Working Conference on Reverse Engineering (WCRE'00)*, pages 150–159, November 2000.
- [HH01] Mark Harman and Robert M. Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, Autumn 2001.
- [HH02] Ahmed E. Hassan and Richard C. Holt. Architecture recovery of Web applications. *24th ACM/IEEE International Conference on Software Engineering (ICSE'02)*, pages 349–359, May 2002.
- [HHBM97] T. Dean Hendrix, James H. Cross, Larry A. Barowski, and Karl S. Mathias. Tool support for reverse engineering multi-lingual software. *4th IEEE Working Conference on Reverse Engineering (WCRE'97)*, pages 136–143, October 1997.

- [HHN03] Minmin Han, Christine Hofmeister, and Robert L. Nord. Reconstructing software architecture for J2EE web applications. *10th IEEE Working Conference on Reverse Engineering (WCRE'03)*, pages 67–78, November 2003.
- [HHT01] Jochen Hartmann, Shihong Huang, and Scott Tilley. Documenting software systems with views II: An integrated approach based on XML. *19th ACM International Conference on Computer Documentation (SIGDOC'01)*, pages 237–246, October 2001.
- [Him96] Michael Himsolt. GML: Graph modelling language. University of Passau, Germany, Unpublished, December 1996.
- [Hin01] Michael Hind. Pointer analysis: Haven't we solved this problem yet? *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, pages 54–61, June 2001.
- [HK05] Jan Hannemann and Gail C. Murphy Gregor Kiczales. Role-based refactoring of crosscutting concerns. *4th ACM International Conference on Aspect-Oriented Software Development (AOSD'05)*, pages 135–146, March 2005.
- [HL02a] Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge. Compression techniques to simplify the analysis of large execution traces. *10th IEEE International Workshop on Program Comprehension (IWPC'02)*, pages 159–168, June 2002.
- [HL02b] Se-Joon Hong and F. Javier Lerch. A laboratory study of consumers' preferences and purchasing behavior with regards to software components. *ACM SIGMIS Database*, 33(3):23–37, Summer 2002.
- [HL04] Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge. A survey of trace exploration tools. *Conference of The Centre for Advanced Studies On Collaborative Research (CASCON'04)*, pages 42–55, October 2004.
- [HLF04] Abdelwahab Hamou-Lhadj, Timothy C. Lethbridge, and Lianjiang Fu. Challenges and requirements for an effective trace exploration tool. *12th IEEE International Workshop on Program Comprehension (IWPC'04)*, pages 70–78, June 2004.
- [HMP03] Mark Hennessy, Brian A. Malloy, and James F. Power. gccXfront: Exploiting gcc as a front end for program comprehension tools via XML/XSLT. *11th IEEE International Workshop on Program Comprehension (IWPC'03)*, pages 298–299, May 2003.
- [HN01] Kelli Houston and Davyd Norris. Software components and the UML. In George T. Heineman and William T. Councill, editors, *Component-Based Software Engineering: Putting the Pieces Together*, chapter 14, pages 243–262. Addison-Wesley, 2001.
- [HNHR00] M. Elizabeth C. Hull, Peter N. Nicholl, Philip Houston, and Niall Rooney. Towards a visual approach for component-based software development. *Software – Concepts & Tools*, 19(4):154–160, August 2000.
- [Hol96] R. C. Holt. Binary relational algebra applied to software architecture. Technical Report 345, CSRI, University of Toronto, 1996. <http://plg.uwaterloo.ca/~holt/papers/report345.ps>.
- [Hol97] Ric Holt. TA: The tuple-attribute language. <http://plg2.math.uwaterloo.ca/~holt/papers/ta-intro.html>, 1997.
- [Hol00] Ric Holt. Wosef introduction. *Workshop on Standard Exchange Format (WoSEF)*, June 2000.
- [Hol01] Ric Holt. Software architecture as a shared mental model. *1st ASERC Workshop on Software Architecture*, August 2001.
- [Hol04] Andreas Holzinger. Rapid prototyping for a virtual medical campus interface. *IEEE Software*, 21(1):92–99, January/February 2004.
- [Hol05] Andreas Holzinger. Usability engineering methods for software developers. *Communications of the ACM*, 48(1):71–74, January 2005.
- [Hop00] Jon Hopkins. Component primer. *Communications of the ACM*, 42(10):27–30, October 2000.

- [HOT00] William Harrison, Harold Ossher, and Peri Tarr. Software engineering tools and environments: A roadmap. *Conference on The Future of Software Engineering*, pages 263–277, June 2000.
- [HPM⁺05] P. R. Henriques, M. J. V. Pereira, M. Mernik, M. Lenic, J. Gray, and H. Wu. Automatic generation of language-based tools using the LISA system. *IEE Proceedings – Software*, 152(2):54–69, April 2005.
- [HT02] Andy Hunt and Dave Thomas. Software archaeology. *IEEE Software*, 19(2):20–22, March/April 2002.
- [HTT97] Duane W. Hybertson, Anh D. Ta, and William M. Thomas. Maintenance of COTS-intensive software systems. *Journal of Software Maintenance: Research and Practice*, 9(4):203–216, July/August 1997.
- [HTZ03] Shihong Huang, Scott Tilley, and Zhou Zhiying. On the yin and yang of academic research and industrial practice. *3rd International Workshop on Adoption-Centric Software Engineering (ACSE'03)*, pages 19–22, May 2003.
- [Hua04] Shihong Huang. *An Integrated Approach to Program Redocumentation*. PhD thesis, University of California Riverside, June 2004.
- [HWS00] Richard C. Holt, Andreas Winter, and Andy Schürr. GXL: Towards a standard exchange format. *7th IEEE Working Conference on Reverse Engineering (WCRE'00)*, pages 162–171, November 2000.
- [Iiv96] Juhani Iivari. Why are CASE tools not used? *Communications of the ACM*, 39(10):94–103, October 1996.
- [Ins90] The Institute of Electrical and Electronics Engineers. *IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990)*, 1990.
- [Jah99] Jens Jahnke. *Managing Uncertainty and Inconsistency in Database Reengineering Processes*. PhD thesis, University of Paderborn, Germany, August 1999.
- [Jaz04] Mehdi Jazayeri. The education of a software engineer. *19th IEEE Conference on Automated Software Engineering (ASE'04)*, pages xviii–xxvii, September 2004.
- [JBC⁺02] Jon (Sean) Jaspersen, Brian S. Butler, Traci A. Carte, Henry J. P. Croes, Carol S. Saunders, and Weijun Zheng. Power and information technology research: A meta-triangulation review. *MIS Quarterly*, 26(4):397–459, December 2002.
- [JCD02] Dean Jin, James R. Cordy, and Thomas R. Dean. Where's the schema? A taxonomy of patterns for software exchange. *10th IEEE International Workshop on Program Comprehension (IWPC'02)*, pages 65–74, June 2002.
- [JCD03] Dean Jin, James R. Cordy, and Thomas R. Dean. Transparent reverse engineering tool integration using a conceptual transaction adapter. *7th IEEE European Conference on Software Maintenance and Reengineering (CSMR'03)*, pages 399–408, March 2003.
- [JCH01] Istvan Jonyer, Diane J. Cook, and Lawrence B. Holder. Graph-based hierarchical conceptual clustering. *Journal of Machine Learning Research*, 2(1):19–43, Winter 2001.
- [JE04] Sven Johann and Alexander Egyed. State consistency strategies for COTS integration. *International Workshop on Incorporating COTS-Software into Software Systems: Tools and Techniques (IWICSS'04)*, pages 33–38, February 2004. <http://www.tuisr.utulsa.edu/iwicss/>.
- [JGJ97] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software Reuse: Architecture Process and Organization for Business Success*. ACM Press, 1997.
- [JH98] Stan Jarzabek and Riri Huang. The case for user-centered CASE tools. *Communications of the ACM*, 81(8):93–99, August 1998.
- [Jin01] Dean Jin. Exchange of software representations among reverse engineering tools. External Technical Report ISSN-0836-0227-2001-454, Department of Computing and Information Science, Queen's University, December 2001.

- [Jin04] Dean Jin. *Ontological Adaptive Integration of Reverse Engineering Tools*. PhD thesis, Queen's University, Canada, August 2004.
- [JMW⁺02] Jens H. Jahnke, Hausi A. Müller, Andrew Walenstein, Nikolai Mansurov, and Kenny Wong. Fused data-centric visualizations for software evolution environments. *10th IEEE International Workshop on Program Comprehension (IWPC'02)*, pages 187–196, June 2002.
- [Joh94] J. Howard Johnson. Substring matching for clone detection and change tracking. *10th IEEE International Conference on Software Maintenance (ICSM'94)*, pages 120–126, September 1994.
- [Joh97] Ralph E. Johnson. Frameworks = (components+patterns). *Communications of the ACM*, 40(10):39–42, October 1997.
- [Joh00] Jeff Johnson. *GUI Bloopers: Don'ts and Do's for Software Developers and Web Designers*. Morgan Kaufmann Publishers, March 2000.
- [Jon98] Capers Jones. *The Year 2000 Software Problem: Quantifying the Costs and Assessing the Consequences*. ACM Press, 1998.
- [JQB⁺99] Matthias Jarke, Christoph Quix, Guido Blees, Dirk Lehmann, Gunter Michalk, and Stefan Stierl. Improving OLTP data quality using data warehouse mechanisms. *ACM SIGMOD International Conference on Management of Data*, pages 536–537, June 1999.
- [JR97] Dean Jerding and Spencer Rugaber. Using visualization for architectural localization and extraction. *4th IEEE Working Conference on Reverse Engineering (WCRE'97)*, pages 56–65, October 1997.
- [JR00] Daniel Jackson and Martin Rinard. Software analysis: A roadmap. *Conference on The Future of Software Engineering*, pages 135–145, June 2000.
- [JS03] Juanjuan Jiang and Tarja Systä. Exploring differences in exchange formats—tool support and case studies. *7th IEEE European Conference on Software Maintenance and Reengineering (CSMR'03)*, pages 389–398, March 2003.
- [JWZ02] Jens H. Jahnke, Jörg P. Wadsack, and Albert Zündorf. A history concept for design recovery tools. *6th IEEE European Conference on Software Maintenance and Reengineering (CSMR'02)*, pages 37–46, March 2002.
- [KABC96] Rick Kazman, Gregory Abowd, Len Bass, and Paul Clements. Scenario-based analysis of software architecture. *IEEE Software*, 13(6):47–55, November 1996.
- [Kai96] J. Bradford Kain. Enabling an application or system to be the sum of its parts. *Object Magazin*, 6:64–69, April 1996.
- [Kam94] J.F.Th. Kamperman. GEL, a graph exchange language. Technical report, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, Netherlands, 1994. <http://ftp.cwi.nl/gipe/reports/Kam94.ps.Z>.
- [Kam98] Manfred Kamp. Managing a multi-file, multi-language software repository for program comprehension tools—a generic approach. Fachberichte Informatik 1/98, Universität Koblenz-Landau, Institut für Informatik, Koblenz, Germany, 1998.
- [Kar98] Michael Karasick. The architecture of Montana: An open and extensible programming environment with an incremental C++ compiler. *6th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE-6)*, pages 131–142, November 1998.
- [Kar99] Helena Karsten. Collaboration and collaborative information technologies: A review of the evidence. *The DATA BASE for Advances in Information Systems*, 30(2):44–65, Spring 1999.
- [Kas03] Antonia Kastendiek. *Computer und Ethik statt Computerethik*. Technikphilosophie. Lit Verlag, 2003.

- [Kaz96] Rick Kazman. Tool support for architecture analysis and design. *2nd International Software Architecture Workshop (ISAW-2)*, pages 94–97, October 1996.
- [KB98] Wojtek Kozaczynski and Grady Booch. Component-based software engineering. *IEEE Software*, 15(5):34–36, September/October 1998.
- [KBAW94] Rick Kazman, Len Bass, Gregory Abowd, and Mike Webb. SAAM: A method for analyzing the properties of software architectures. *16th ACM/IEEE International Conference on Software Engineering (ICSE'94)*, pages 81–90, May 1994.
- [KC92] Eduardo Korthright and David Cordes. Cnest and Cscope: Tools for the literate programming environment. *IEEE Southeastcon '92*, 2:604–609, April 1992.
- [KC95] Mark Keil and Erran Carmel. Customer-developer links in software development. *Communications of the ACM*, 38(5):33–44, May 1995.
- [KC96] Rick Kazman and Jeromy Carriere. Rapid prototyping of information visualizations using VANISH. *InfoVis'96*, pages 21–28, October 1996.
- [KC98] Rick Kazman and S. Jeromy Carriere. View extraction and view fusion in architectural understanding. *5th International Conference on Software Reuse (ICSR-5)*, pages 290–299, June 1998.
- [KC99] Rick Kazman and S. Jeromy Carriere. Playing detective: Reconstructing software architecture from available evidence. *Journal of Automated Software Engineering*, 6(2):107–138, April 1999.
- [KCBA97] Rick Kazman, Paul Clements, Len Bass, and Gregory Abowd. Classifying architectural elements as foundation for mechanism mismatching. *21st IEEE International Computer Software and Applications Conference (COMPSAC'97)*, pages 14–17, August 1997.
- [KCE00] Holger M. Kienle, Jörg Czeranski, and Thomas Eisenbarth. The API perspective of exchange formats. *Workshop on Standard Exchange Format (WoSEF)*, June 2000. <http://www.ics.uci.edu/~ses/wosef/papers/Kienle-api.ps>.
- [KCK99] Jeffrey Korn, Yih-Farn Chen, and Eleftherios Koutsofios. Chava: Reverse engineering and tracking of Java applets. *6th IEEE Working Conference on Reverse Engineering (WCRE'99)*, pages 314–325, October 1999.
- [KDJ04] Barbara A. Kitchenham, Tore Dyba, and Magne Jorgensen. Evidence-based software engineering. *26th ACM/IEEE International Conference on Software Engineering (ICSE'04)*, pages 273–281, May 2004.
- [Kee04] Frank Keenan. Agile process tailoring and problem analysis (APPLY). *26th ACM/IEEE International Conference on Software Engineering (ICSE'04)*, pages 45–47, May 2004.
- [Kem92] Chris F. Kemerer. How the learning curve affects CASE adoption. *IEEE Software*, 9(3):23–28, May 1992.
- [KFY⁺06] Pranam Kolari, Tim Finin, Yelena Yesha, Kelly Lyons, Jen Hawkins, and Stephen Perelgut. Policy management of enterprise systems: A requirements study. *7th International Workshop on Policies for Distributed Systems and Networks (POLICY'06)*, pages 231–234, June 2006.
- [KG05] Cory Kasper and Michael W. Godfrey. Improved tool support for the investigation of duplication in software. *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 305–314, September 2005.
- [KGW98] Rainer Koschke, Jean-François Girard, and Martin Würthner. An intermediate representation for integrating reverse engineering analyses. *5th IEEE Working Conference on Reverse Engineering (WCRE'98)*, pages 241–250, October 1998.
- [KH05] Gerald Kotonya and John Hutchinson. Managing change in COTS-based systems. *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 69–78, September 2005.

- [Kie01] Holger M. Kienle. Exchange format bibliography. *ACM SIGSOFT Software Engineering Notes*, 26(1):56–60, January 2001.
- [Kir02] Michele Kirchner. Evaluation, repair, and transformation of Web pages for Web content accessibility. review and some available tools. *4th IEEE International Workshop on Web Site Evolution (WSE'02)*, pages 65–72, October 2002.
- [Kit04] Barbara Kitchenham. Procedures for performing systematic reviews. Technical Report TR/SE-0401, Department of Computer Science, University of Keele, July 2004.
- [KJ03] Holger M. Kienle and Jens H. Jahnke. A visual language in Visio: First experiences. *3rd International Workshop on Adoption-Centric Software Engineering (ACSE'03)*, pages 90–93, May 2003.
- [KK99] Mark Klein and Rick Kazman. Attribute-based architectural styles. Technical Report CMU/SEI-99-TR-022, Software Engineering Institute, Carnegie Mellon University, October 1999. http://www.sei.cmu.edu/pub/documents/99_reports/pdf/99tr022.pdf.
- [KKB⁺99] Mark H. Klein, Rick Kazman, Len Bass, Jeromy Carriere, Mario Barbacci, and Howard Lipson. Attribute-based architectural styles. *1st IFIP Working Conference on Software Architecture (WICSA'99)*, pages 225–243, February 1999.
- [KL98] Paul Kahn and Krzysztof Lenk. Principles of typography for user interface design. *interactions*, 5(6):15–29, 1998.
- [KL03] S. Klusener and R. Lämmel. Deriving tolerant grammars from a base-line grammar and constructs of interest, April 8 2003. Draft, 10 pages.
- [KLC05] Axel Anders Kvale, Jingyue Li, and Reidar Conradi. A case study on building COTS-based system using aspect-oriented programming. *20th ACM Symposium on Applied Computing (SAC'05)*, pages 1491–1498, March 2005.
- [Kli03] Paul Klint. How understanding and restructuring differ from compiling: a rewriting perspective. *11th IEEE International Workshop on Program Comprehension (IWPC'03)*, pages 1–9, May 2003.
- [KLM04] Holger M. Kienle, Marin Litoiu, and Hausi A. Müller. Using components to build software engineering tools. *4th International Workshop on Adoption-Centric Software Engineering (ACSE'04)*, pages 36–42, May 2004.
- [KLV05] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering and Methodology*, 14(3):331–380, July 2005.
- [KM97] Minna Koskinen and Pentti Marttiin. Process support in MetaCASE: Implementing the conceptual basis for enactable process models in MetaEdit+. *8th IEEE Conference on Software Engineering Environments*, pages 110–122, April 1997.
- [KM01] Holger M. Kienle and Hausi A. Müller. Leveraging program analysis for web site reverse engineering. *3rd IEEE International Workshop on Web Site Evolution (WSE'01)*, pages 117–125, November 2001.
- [KM02a] Holger M. Kienle and David L. Moore. smgn: Rapid prototyping of small domain-specific languages. *Journal of Computing and Information Technology (CIT)*, 10(1):37–52, 2002.
- [KM02b] Claire Knight and Malcolm Munro. Program comprehension experiences with GXL; comprehension for comprehension. *10th IEEE International Workshop on Program Comprehension (IWPC'02)*, pages 147–156, June 2002.
- [KM06] Holger M. Kienle and Hausi A. Müller. A WSAD-based fact extractor for J2EE web projects. *Technical Report, University of Victoria*, June 2006.
- [KMP05a] Nicholas A. Kraft, Brian A. Malloy, and James F. Power. g⁴re: Harnessing GCC to reverse engineer C++ applications. *Dagstuhl Seminar 05161, Schloss Dagstuhl*, April 2005. <http://www.cs.clemson.edu/~nkraft/research/dagstuhl05/dagstuhl05.pdf>.

- [KMP05b] Nicholas A. Kraft, Brian A. Malloy, and James F. Power. Toward an infrastructure to support interoperability in reverse engineering. *12th IEEE Working Conference on Reverse Engineering (WCRE'05)*, pages 196–205, November 2005.
- [Knu84] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, May 1984.
- [Knu96] Donald Knuth. An interview with Donald Knuth. *Dr. Dobbs's Journal*, pages 16–22, April 1996.
- [Kob00] Chris Kobryn. Modeling components and frameworks with uml. *Communications of the ACM*, 43(10):31–38, October 2000.
- [Köl00] Ulrike Kölsch. *Methodische Integration und Migration von Informationssystemen in object-orientierte Umgebungen*. PhD thesis, University of Karlsruhe, Germany, 2000.
- [Kop97] Rainer Koppler. A systematic approach to fuzzy parsing. *Software—Practice and Experience*, 27(6):637–649, June 1997.
- [Kos00] Rainer Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, University of Stuttgart, Germany, 2000.
- [Kos02] Rainer Koschke. Software visualization for reverse engineering. In S. Diehl, editor, *Software Visualization*, volume 2269 of *Lecture Notes in Computer Science*, pages 138–150. Springer-Verlag, 2002.
- [Kos03] Rainer Koschke. Software visualization in software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 15(2):87–109, March 2003.
- [KOV03] Rick Kazman, Liam O'Brien, and Chris Verhoef. Architecture reconstruction guidelines, third edition. Technical Report CMU/SEI-2002-TR-034, Software Engineering Institute, Carnegie Mellon University, November 2003. <http://www.sei.cmu.edu/pub/documents/03.reports/03tr034.pdf>.
- [KP88] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming (JOOP)*, 1(3):26–49, August/September 1988.
- [KP03] Jens Knodel and Martin Pinzger. Improving fact extraction of framework-based software systems. *10th IEEE Working Conference on Reverse Engineering (WCRE'03)*, pages 186–195, November 2003.
- [KPP⁺02] Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El-Emam, and Jarett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, August 2002.
- [Kra98] George Kraft. CDE plug-and-play. *Linux Journal*, May 1998. <http://www.linuxjournal.com/article/2362>.
- [Kri97] Rene L. Krikhaar. Reverse architecting approach for complex systems. *13th IEEE International Conference on Software Maintenance (ICSM'97)*, pages 4–11, October 1997.
- [Kri01] Jens Krinke. Identifying similar code with program dependence graphs. *8th IEEE Working Conference on Reverse Engineering (WCRE'01)*, pages 301–309, October 2001.
- [Kru92] Charles W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.
- [Kru95] Phillippe B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, November 1995.
- [Kru98] Phillippe Kruchten. Modeling component systems with the unified modeling language. *International Workshop on Component-Based Software Engineering (CBSE'98)*, April 1998. <http://www.sei.cmu.edu/pacc/icse98/papers/p1.html>.

- [Kru99] Philippe Kruchten. *The Rational Unified Process: an introduction*. Object Technology Series. Addison-Wesley, 1999.
- [Kru02] Charles Krueger. Eliminating the adoption barrier. *IEEE Software*, 19(4):29–31, July/August 2002.
- [KS90] Anthony Karrer and Walt Scacchi. Requirements for an extensible object-oriented tree/graph editor. *3rd ACM SIGGRAPH Symposium on User Interface Software and Technology (UIST'90)*, pages 84–91, October 1990.
- [KS94] S. H. Kan and V. R. Basili L. N. Shapiro. Software quality: An overview from the perspective of total quality management. *IBM Systems Journal*, 33(1):4–19, January 1994.
- [KS97] Peter Klein and Andy Schürr. Constructing SDEs with the IPSEN meta environment. *8th IEEE Conference on Software Engineering Environments*, pages 2–10, April 1997.
- [KS01] Peter Knauber and Giancarlo Succi. Perspectives on software product lines. *ACM SIGSOFT Software Engineering Notes*, 26(2):29–33, March 2001.
- [KS03a] Holger M. Kienle and Susan Elliott Sim. Towards a benchmark for web site extractors: A call for community participation. *7th IEEE European Conference on Software Maintenance and Reengineering (CSMR'03)*, pages 82–87, March 2003.
- [KS03b] Rainer Koschke and Daniel Simon. Hierarchical reflexion models. *10th IEEE Working Conference on Reverse Engineering (WCRE'03)*, pages 36–45, November 2003.
- [KSO02] Kostas Kontogiannis, Dennis Smith, and Liam O'Brien. On the role of services in enterprise application integration. *10th International Workshop on Software Technology and Engineering Practice (STEP'02)*, October 2002.
- [KSRP99] Rudolf K. Keller, Reinhard Schauer, Sebastien Robitaille, and Patrick Page. Pattern-based reverse-engineering of design components. *21st ACM/IEEE International Conference on Software Engineering (ICSE'99)*, pages 226–235, May 1999.
- [KSS⁺02] Ralf Kollmann, Petri Selonen, Eleni Stroulia, Tarja Systä, and Albert Zündorf. A study on the current state of the art in tool-supported UML-based static reverse engineering. *9th IEEE Working Conference on Reverse Engineering (WCRE'02)*, pages 22–32, October 2002.
- [KSSH03] Jens Krinke, Mirko Streckenbach, Maximilian Störzer, and Christian Hammer. Using program analysis infrastructure for software maintenance. <http://www.fernuni-hagen.de/ST/publications/pai.pdf>, March 2003.
- [KT05] Mark Keil and Amrit Tiwana. Beyond cost: The drivers of COTS application value. *IEEE Software*, 22(3):64–69, May/June 2005.
- [Kuc05] Susan Kuchinskas. All microsoft or none? Jupitermedia, March 2005. <http://www.internetnews.com/ent-news/article.php/3493416>.
- [KV92] Timothy D. Korson and Vijay K. Vaishnavi. Managing emerging software technologies: A technology transfer framework. *Communications of the ACM*, 35(9):101–111, September 1992.
- [KW99] Bernt Kullbach and Andreas Winter. Querying as an enabling technology in software reengineering. *3rd IEEE European Conference on Software Maintenance and Reengineering (CSMR'99)*, pages 42–50, March 1999.
- [KWC98] Rick Kazman, Steven G. Woods, and S. Jeromy Carriere. Requirements for integrating software architecture and reengineering models: CORUM II. *5th IEEE Working Conference on Reverse Engineering (WCRE'98)*, pages 154–163, October 1998.
- [KWDE98] Bernt Kullbach, Andreas Winter, Peter Dahm, and Jürgen Ebert. Program comprehension in multi-language systems. *5th IEEE Working Conference on Reverse Engineering (WCRE'98)*, pages 135–143, October 1998.

- [KWJM02] Holger M. Kienle, Anke Weber, Jens Jahnke, and Hausi A. Müller. Tackling the adoption problem of domain-specific visual languages. *2nd Domain-Specific Modeling Languages Workshop at OOPSLA 2002*, pages 77–88, October 2002.
- [KWM02] Holger M. Kienle, Anke Weber, and Hausi A. Müller. Leveraging SVG in the Rigi reverse engineering tool. *SVG Open / Carto.net Developers Conference*, July 2002.
- [KWMM03] Holger M. Kienle, Anke Weber, Johannes Martin, and Hausi A. Müller. Development and maintenance of a web site for a bachelor program. *5th IEEE International Workshop on Web Site Evolution (WSE'03)*, pages 20–29, September 2003.
- [LA97] Timothy C. Lethbridge and Nicolas Anquetil. Architecture of a source code exploration tool: A software engineering case study. Technical Report TR-97-07, University of Ottawa, Computer Science, 1997.
- [Lak96] Arun Lakhota. A unified framework for software subsystem classification techniques. *Journal of Systems and Software*, 36(3):211–231, March 1996.
- [Lan03] Michele Lanza. Codecrawler—lessons learned in building a software visualization tool. *7th IEEE European Conference on Software Maintenance and Reengineering (CSMR'03)*, pages 1–10, March 2003.
- [Lar04] Craig Larman. *Applying UML and Patterns*. Prentice Hall, third edition, 2004.
- [LB03] Craig Larman and Victor R. Basili. Iterative and incremental development: A brief history. *IEEE Computer*, 36(6):47–56, June 2003.
- [LBKK97] Chung-Horng Lung, Sonia Bot, Kalai Kalaichelvan, and Rick Kazman. An approach to software architecture analysis for evolution and reusability. *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'97)*, November 1997.
- [LBvVB02] Nico Lassing, PerOlof Bengtsson, Hans van Vliet, and Jan Bosch. Experiences with ALMA: Architecture-level modifiability analysis. *Journal of Systems and Software*, 61(1):47–57, March 2002.
- [LCBO03] Panagiotis K. Linos, Zhi-hong Chen, Seth Berrier, and Brian O'Rourke. A tool for understanding multi-language program dependencies. *11th IEEE International Workshop on Program Comprehension (IWPC'03)*, pages 64–72, May 2003.
- [LD03] Michele Lanza and Stephane Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, September 2003.
- [Let98] Timothy C. Lethbridge. Requirements and proposal for a software information exchange format (SIEF) standard. *Unpublished*, November 1998. <http://www.site.uottawa.ca/~tcl/papers/sief/standardProposalv1.html>.
- [Let00] Timothy C. Lethbridge. Integrated personal work management in the tksee software exploration tool. *International Symposium on Constructing Software Engineering Tools (COSET'00)*, June 2000.
- [Let01] Timothy C. Lethbridge. Report from the dagstuhl seminar on interoperability of reengineering tools. *9th IEEE International Workshop on Program Comprehension (IWPC'01)*, page 119, May 2001.
- [Let04] Timothy C. Lethbridge. Value assessment by potential tool adopters: Towards a model that considers costs, benefits and risks of adoption. *4th International Workshop on Adoption-Centric Software Engineering (ACSE'04)*, pages 46–50, May 2004.
- [Lew98] Scott M. Lewandowski. Frameworks for component-based client/server computing. *ACM Computing Surveys*, 30(1):3–27, March 1998.
- [LH01] Timothy C. Lethbridge and Francisco Herrera. Assessing the usefulness of the tksee software exploration tool. In Hakan Erdogmus and Oryal Tanir, editors, *Advances in Software Engineering: Topics in Comprehension, Evolution, and Evaluation*, chapter 11, pages 73–93. Springer-Verlag, December 2001.

- [LHM03] Yuan Lin, Richard C. Holt, and Andrew J. Malton. Completeness of a fact extractor. *10th IEEE Working Conference on Reverse Engineering (WCRE'03)*, pages 196–205, November 2003.
- [LIGA96] Ramiro Liscano, Roger Impey, Paul Gordon, and Suhayya Abu-Hakima. A system for the seamless integration of personal messages using agents developed on a Lotus Notes platform. *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'96)*, November 1996.
- [Lin95] Panagiotis K. Linos. PolyCARE: A tool for re-engineering multi-language program integrations. *1st IEEE International Conference on Engineering of Complex Computer Systems*, pages 338–341, November 1995.
- [Liu04] Tao Liu. A JavaScript layer for the Rigi reverse engineering tool. Project report, University of Victoria, Department of Computer Science, August 2004.
- [LKSP02] Christopher T. Lewis, Steve Karcz, Andrew Sharpe, and Isobel A. P. Parkin. BioViz: Genome viewer. *SVG Open / Carto.net Developers Conference*, July 2002. http://www.svgopen.org/2002/papers/lewis_et_al_bioviz_genome_viewer/.
- [LL01] Huixiang Liu and Timothy C. Lethbridge. Intelligent search techniques for large software systems. *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'01)*, November 2001.
- [LL04] Björn Lundell and Brian Lings. Changing perceptions of CASE technology. *Journal of Systems and Software*, 72(2):271–280, July 2004.
- [LLHX01] Terence C. Lau, Jianguo Lu, Erik Hedges, and Emily Xing. Migrating e-commerce database applications to an enterprise Java environment. *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'01)*, November 2001.
- [LLL00] Sebastien Lapierre, Bruno Lague, and Charles Leduc. Datrix source model and its interchange format: Lessons learned and considerations for future work. *Workshop on Standard Exchange Format (Wosef)*, June 2000. <http://www.ics.uci.edu/~ses/wosef/papers/Lapierre.pdf>.
- [LLWY03] Arun Lakhotia, Junwei Li, Andrew Walenstein, and Yun Yang. Towards a clone detection benchmark suite and results archive. *11th IEEE International Workshop on Program Comprehension (IWPC'03)*, pages 285–286, May 2003.
- [LM04] Grace A. Lewis and Edwin J. Morris. From system requirements to COTS evaluation criteria. In R. Kazman and D. Ports, editors, *3rd International Conference on COTS-Based Software Systems (ICCBSS'04)*, volume 2959 of *Lecture Notes in Computer Science*, pages 159–168. Springer-Verlag, 2004.
- [LMB⁺01] Akos Ledecz, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. The generic modeling environment. *International Workshop on Intelligent Signal Processing (WISP'01)*, May 2001.
- [LMOS05] Grace Lewis, Edwin Morris, Liam O'Brien, and Dennis Smith. Analyzing the reuse potential of migrating legacy components to a service-oriented architecture. *ICSM 2005 Poster Proceedings*, pages 39–42, September 2005.
- [LMS06] Grace Lewis, Edwin Morris, and Dennis Smith. Analyzing the reuse potential of migrating legacy components to a service-oriented architecture. *10th IEEE Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 15–23, March 2006.
- [LMSW03] Rob Lintern, Jeff Michaud, Margaret-Anne Storey, and Xiaomin Wu. Plugging-in visualization: experiences integrating a visualization tool with Eclipse. *ACM Symposium on Software Visualization (SoftVis'03)*, pages 47–56, June 2003.
- [LN04] Phillip A. Laplante and Colin J. Neill. Opinion: The demise of the waterfall model is imminent. *ACM Queue*, 1(10):10–15, February 2004.

- [LON01a] Panagiotis K. Linos, Esther T. Ososanya, and Harri Natarajan. Maintenance support for Web sites: A case study. *3rd IEEE International Workshop on Web Site Evolution (WSE'01)*, pages 70–76, November 2001.
- [Lon01b] Andy Longshaw. Choosing between COM+, EJB, and CCM. In George T. Heineman and William T. Councill, editors, *Component-Based Software Engineering: Putting the Pieces Together*, chapter 35, pages 621–640. Addison-Wesley, 2001.
- [LR95] David A. Ladd and Christopher Ramming. A*: A language for implementing language processors. *IEEE Transactions on Software Engineering*, 21(11):894–901, November 1995.
- [LS92] Luqi and R. Steigerwald. Rapid software prototyping. *25th IEEE Hawaii International Conference on System Sciences (HICSS'92)*, ii:470–479, January 1992.
- [LS96] Timothy C. Lethbridge and Janice Singer. Strategies for studying maintenance. *2nd Workshop on Empirical Studies of Software Maintenance (WESS'96)*, pages 79–83, November 1996.
- [LS97] Timothy C. Lethbridge and Janice Singer. Understanding software maintenance tools: Some empirical research. *3rd Workshop on Empirical Studies of Software Maintenance (WESS'97)*, pages 157–162, October 1997.
- [LS02] Claus Lewerentz and Frank Simon. Metrics-based 3D visualization of large object-oriented programs. *1st International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'02)*, pages 70–77, June 2002.
- [LSP⁺03] Luqi, M. Shing, J. Puett, V. Berzins, Z. Guan, Y. Qiao, L. Zhang, N. Chaki, X. Liang, W. Ray, M. Brown, and D. Floodeen. Comparative rapid prototyping, a case study. *14th IEEE International Workshop on Rapid Systems Prototyping (RSP'03)*, pages 210–217, June 2003.
- [LSS05] Timothy C. Lethbridge, Susan Elliott Sim, and Janice Singer. Studying software engineers: Data collection techniques for software field studies. *Journal on Empirical Software Engineering*, 10(3):311–341, July 2005.
- [LSW01] Carola Lange, Harry M. Sneed, and Andreas Winter. Comparing graph-based program comprehension tools to relational database-based tools. *9th IEEE International Workshop on Program Comprehension (IWPC'01)*, pages 209–218, May 2001.
- [LT97] Tung Lai Lai and Efraim Turban. One organization's use of Lotus Notes. *Communications of the ACM*, 40(10):19–21, October 1997.
- [LTP04] Timothy C. Lethbridge, Sander Tichelaar, and Erhard Ploedereder. The dagstuhl middle metamodel: A schema for reverse engineering. In J.-M. Favre, M. Godfrey, and A. Winter, editors, *International Workshop on Meta-Models and Schemas for Reverse Engineering (ateM'03)*, volume 94, pages 7–18. Elsevier, May 2004.
- [Lüe03] Chris Lüer. Evaluating the Eclipse platform as a composition environment. *3rd International Workshop on Adoption-Centric Software Engineering (ACSE'03)*, pages 59–61, May 2003.
- [LV01] R. Lämmel and C. Verhoef. Cracking the 500-language problem. *IEEE Software*, 18(6):78–88, November/December 2001.
- [LVC89] Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing user interfaces with InterViews. *IEEE Computer*, 22(2):8–22, February 1989.
- [LvdH02] Chris Lüer and Andre van der Hoek. Composition environments for deployable software components. Technical Report UCI-ICS-02-18, Department of Information and Computer Science, University of California, Irvine, August 2002.
- [LW04] Grace Lewis and Lutz Wrangé. Approaches to constructive interoperability. Technical Report CMU/SEI-2004-TR-020, Software Engineering Institute, Carnegie Mellon University, December 2004. <http://www.sei.cmu.edu/pub/documents/04.reports/pdf/04tr020.pdf>.

- [LY01] Hareton K. N. Leung and Terence C. F. Yuen. A process framework for small projects. *Software Process: Improvement and Practice*, 6(2):67–83, June 2001.
- [MA01] Bjorn Erik Munkvold and Robert Anson. Organizational adoption and diffusion of electronic meeting systems: A case study. *ACM 2001 International Conference on Supporting Group Work (GROUP '01)*, pages 279–287, September 2001.
- [Ma04] Jun Ma. Building reverse engineering tools using Lotus Notes. Master's thesis, University of Victoria, Department of Computer Science, October 2004.
- [MAB03] Ed Morris, Cecilia Albert, and Lisa Brownsword. COTS-based development: Taking the pulse of a project. In H. Erdogmus and T. Weng, editors, *2nd International Conference on COTS-Based Software Systems (ICCBSS'03)*, volume 2580 of *Lecture Notes in Computer Science*, pages 168–177. Springer-Verlag, 2003.
- [Mac91] Wendy E. Mackay. Triggers and barriers to customizing software. *ACM SIGCHI Conference on Human Factors in Computing Systems (CHI'91)*, pages 153–160, April 1991.
- [MAFN04] Anders Möller, Mikael Akerholm, Johan Fredriksson, and Mikael Nolin. Evaluation of component technologies with respect to industrial requirements. *30th IEEE EUROMICRO Conference (EUROMICRO'04)*, pages 56–63, August 2004.
- [Mal01] David Malone. Developing Lotus Notes applications. *CASCON 2001 Workshop on Adoption-Centric Tool Development (ACTD'01)*, November 2001. <https://www-927.ibm.com/ibm/cas/toronto/publications/TR-74.182/37/dave-ACTD.pdf>.
- [Man94] Spiros Mancoridis. Loosely integrating tools using the star system. *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'94)*, October 1994.
- [Mar99] Johannes Martin. Leveraging IBM VisualAge for C++ for reverse engineering tasks. *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'99)*, pages 83–95, November 1999.
- [Mar01] Robert C. Martin. RUP vs. XP. *objectmentor.com*, 2001. <http://www.objectmentor.com/resources/articles/RUPvsXP.pdf>.
- [Mar02] Johannes Martin. *Ephedra: A C to Java Migration Environment*. PhD thesis, University of Victoria, 2002.
- [Mar03] Johannes Martin. Tool adoption: A software developer's perspective. *3rd International Workshop on Adoption-Centric Software Engineering (ACSE'03)*, pages 7–9, May 2003.
- [Mat04] Mari Matinlassi. Comparison of software product line architecture design methods: COPA, FAST, FORM, KobrA and QADA. *26th ACM/IEEE International Conference on Software Engineering (ICSE'04)*, pages 127–136, May 2004.
- [Mau05] Peter M. Maurer. Converting command-line applications into binary components. *Software—Practice and Experience*, 35(8):787–797, July 2005.
- [MBJK90] John Mylopoulos, Alex Borgida, Matthias Jarke, and Manolis Koubarakis. Telos: Representing knowledge about information systems. *ACM Transactions on Office Information Systems*, 8(4):325–362, October 1990.
- [McC97] Steve McConnell. Software's ten essentials. *IEEE Software*, 14(2):143–144, March/April 1997.
- [MCF04] Stephen J. Mellor, Anthony N. Clark, and Takao Futagami. Model-driven development. *IEEE Software*, 21(5):14–18, September/October 2004.
- [MCG⁺02] Sarah Matzko, Peter J. Clarke, Tanton H. Gibbs, Brian A. Malloy, James F. Power, and Rosemary Monahan. Reveal: a tool to reverse engineer class diagrams. *40th IEEE International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific'02)*, pages 13–21, February 2002.

- [McG04] Tanya McGill. The effect of end user development on end user success. *Journal of Organizational and End User Computing*, 16(1):41–58, January-March 2004.
- [McI76] M. D. McIlroy. Mass-produced software components. *1968 NATO Conference on Software Engineering*, pages 88–98, 1976.
- [McK99] Dorothy McKinney. Impact of commercial off-the-shelf (COTS) software on the interface between systems and software engineering. *21st ACM/IEEE International Conference on Software Engineering (ICSE'99)*, pages 627–628, May 1999.
- [MCLM90] Allan MacLean, Kathleen Carter, Lennard Lövstrand, and Thomas Moran. User-tailorable systems: Pressing the issues with buttons. *ACM SIGCHI Conference on Human Factors in Computing Systems (CHI'90)*, pages 175–182, April 1990.
- [MCN92] John Mylopoulos, Lawrence Chung, and Brian Nixon. Representing and using non-functional requirements: A process-oriented approach. *IEEE Transactions on Software Engineering*, 18(6):483–497, June 1992.
- [MCS05] Andrian Marcus, Denise Comorski, and Andrey Sergeev. Supporting the evolution of a software visualization tool through usability studies. *13th IEEE International Workshop on Program Comprehension (IWPC'05)*, pages 307–316, May 2005.
- [MD97] Michael G. Morris and Andrew Dillon. How user perceptions influence software use. *IEEE Software*, 14(4):58–65, July/August 1997.
- [MDW05] Aaron McCoy, Declan Delaney, and Tomas Ward. Game-state fidelity across distributed interactive games. *Crossroads*, 12(1):10–15, Fall 2005.
- [ME02] Petra Mutzel and Peter Eades. Graphs in software visualization: Introduction. In S. Diehl, editor, *Software Visualization*, volume 2269 of *Lecture Notes in Computer Science*, pages 285–294. Springer-Verlag, 2002.
- [Mee01] Michael Meeks. Bonobo and free software GNOME components. In George T. Heineman and William T. Councill, editors, *Component-Based Software Engineering: Putting the Pieces Together*, chapter 34, pages 607–619. Addison-Wesley, 2001.
- [Meh02] Katharina Mehner. Javis: A UML-based visualization and debugging environment for concurrent Java programs. In S. Diehl, editor, *Software Visualization*, volume 2269 of *Lecture Notes in Computer Science*, pages 163–175. Springer-Verlag, 2002.
- [Mey91] Scott Meyers. Difficulties in integrating multiple development systems. *IEEE Software*, 8(1):49–57, January 1991.
- [Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.
- [Mey99] Bertrand Meyer. On to components. *IEEE Computer*, 32(1):139–140, January 1999.
- [Mey03] Bertrand Meyer. The grand challenge of trusted components. *25th ACM/IEEE International Conference on Software Engineering (ICSE'03)*, pages 660–667, May 2003.
- [MFM03] Andrian Marcus, Louis Feng, and Jonathan I. Maletic. Comprehension of software analysis data using 3D visualization. *11th IEEE International Workshop on Program Comprehension (IWPC'03)*, pages 105–114, May 2003.
- [MGK⁺93] E. Merlo, J. F. Girard, K. Kontogiannis, P. Panangaden, and R. De Mori. Reverse engineering of user interfaces. *1st IEEE Working Conference on Reverse Engineering (WCRE'93)*, pages 171–178, May 1993.
- [MH95] Spiros Mancoridis and Richard C. Holt. Extending programming environments to support architectural design. *7th IEEE International Workshop on Computer-Aided Software Engineering (CASE'95)*, pages 110–119, July 1995.
- [MHG94a] Spiros Mancoridis, Richard C. Holt, and Michael W. Godfrey. A program understanding environment based on the “star” approach to tool integration. *ACM Conference on Computer Science*, pages 60–65, March 1994.

- [MHG94b] Spiros Mancoridis, Richard C. Holt, and Michael W. Godfrey. Tool support for software engineering education. *ICSE-16 Workshop on Software Engineering Education*, May 1994. <http://plg.uwaterloo.ca/~holt/papers/SEedu.ps>.
- [Mit98] David Mitchell. A component approach to embedding awareness and conversation. *7th IEEE Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETIC'98)*, pages 82–89, June 1998.
- [MJS⁺00] Hausi Müller, Jens Jahnke, Dennis Smith, Margaret-Anne Storey, Scott Tilley, and Kenny Wong. Reverse engineering: A roadmap. *Conference on The Future of Software Engineering*, pages 49–60, June 2000.
- [MK88] Hausi Müller and Karl Klashinsky. Rigi: A system for programming-in-the-large. *10th ACM/IEEE International Conference on Software Engineering (ICSE'88)*, pages 80–86, April 1988.
- [MK96] Nabor C. Mendoca and Jeff Kramer. Requirements for an effective architecture recovery framework. *2nd International Software Architecture Workshop (ISAW-2)*, pages 101–105, October 1996.
- [MK00] Evan Mamas and Kostas Kontogiannis. Towards portable source code representations using XML. *7th IEEE Working Conference on Reverse Engineering (WCRE'00)*, pages 172–182, November 2000.
- [MK01] Johannes Martin and Will Kastelic. Tool integration with StarOffice. *CASCON 2001 Workshop on Adoption-Centric Tool Development (ACTD'01)*, November 2001. <https://www-927.ibm.com/ibm/cas/toronto/publications/TR-74.182/37/johannes-ACTD.pdf>.
- [MKES98] Gisela Menger, James Leslie Keedy, Mark Evered, and Axel Schmolinsky. Collection types and implementations in object-oriented software libraries. *26th IEEE International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 26)*, pages 97–109, August 1998.
- [MKK⁺03] Jun Ma, Holger M. Kienle, Piotr Kaminski, Anke Weber, and Marin Litoiu. Customizing Lotus Notes to build software engineering tools. *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'03)*, pages 276–287, October 2003.
- [ML04] Qingxiong Ma and Liping Liu. The technology acceptance model: A meta-analysis of empirical findings. *Journal of Organizational and End User Computing*, 16(1):59–72, January-March 2004.
- [MLL⁺03] Jean-Christophe Mielnik, Bernard Lang, Stephane Lauriere, Jean-Georges Schlosser, and Vincent Bouthors. eCots platform: An inter-industrial initiative for COTS-related information sharing. In H. Erdogmus and T. Weng, editors, *2nd International Conference on COTS-Based Software Systems (ICCBSS'03)*, volume 2580 of *Lecture Notes in Computer Science*, pages 157–167. Springer-Verlag, 2003.
- [MLM95] Jean Mayrand, Claude Leblanc, and Ettore M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. *11th IEEE International Conference on Software Maintenance (ICSM'95)*, pages 244–254, November 1995.
- [MLMD01] Jonathan I. Maletic, Jason Leigh, Andrian Marcus, and Greg Dunlap. Visualizing object-oriented software in virtual reality. *9th IEEE International Workshop on Program Comprehension (IWPC'01)*, pages 26–35, May 2001.
- [MM00] Johannes Martin and Hausi Müller. Considerations on the syntax of a standard exchange format. *Workshop on Standard Exchange Format (WoSEF)*, June 2000.
- [MM01a] Johannes Martin and Ludger Martin. Web site maintenance with software-engineering tools. *3rd IEEE International Workshop on Web Site Evolution (WSE'01)*, pages 126–131, November 2001.

- [MM01b] Johannes Martin and Hausi A. Müller. Discovering implicit inheritance relations in non object-oriented code. In Hakan Erdogmus and Oryal Tanir, editors, *Advances in Software Engineering: Topics in Comprehension, Evolution, and Evaluation*, chapter 11, pages 177–193. Springer-Verlag, December 2001.
- [MMC02] Jonathan I. Maletic, Andrian Marcus, and Michael L. Collard. A task oriented view of software visualization. *1st IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'02)*, pages 32–42, June 2002.
- [MMM95] Hafedh Mili, Fatma Mili, and Ali Mili. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering*, 21(6):528–662, June 1995.
- [MMM⁺05] C. Marinescu, R. Marinescu, P. F. Mihancea, D. Ratiu, and R. Wettel. iPlasma: An integrated platform for quality assessment of object-oriented design. *ICSM 2005 Industrial & Tool Proceedings*, pages 77–80, September 2005.
- [MMP00] Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a taxonomy of software connectors. *22nd ACM/IEEE International Conference on Software Engineering (ICSE'00)*, pages 178–187, June 2000.
- [MN96] Gail C. Murphy and David Notkin. Lightweight lexical source model extraction. *ACM Transactions on Software Engineering and Methodology*, 5(3):262–292, July 1996.
- [MN97] Gail C. Murphy and David Notkin. Reengineering with reflexion models: A case study. *IEEE Computer*, 30(8):29–36, August 1997.
- [MN98] Neil A. Maiden and Cornelius Ncube. Acquiring COTS software selection requirements. *IEEE Software*, 15(2):46–56, March/April 1998.
- [MNB⁺94] Lawrence Markosian, Philip Newcomb, Russell Brand, Scott Burson, and Ted Kitzmiller. Using and enabling technology to reengineer legacy systems. *Communications of the ACM*, 37(5):58–70, May 1994.
- [MNG198] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S. Lan. An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology*, 7(2):158–191, April 1998.
- [MNL96] Gail C. Murphy, David Notkin, and Erica S.-C. Lan. An empirical study of static call graph extractors. *18th ACM/IEEE International Conference on Software Engineering (ICSE'96)*, pages 90–99, May 1996.
- [MNS01] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380, April 2001.
- [Moo01] Leon Moonen. Generating robust parsers using island grammars. *8th IEEE Working Conference on Reverse Engineering (WCRE'01)*, pages 13–22, October 2001.
- [Moo02a] Leon Moonen. *Exploring Software Systems*. PhD thesis, University of Amsterdam, December 2002.
- [Moo02b] Leon Moonen. Lightweight impact analysis using island grammars. *10th IEEE International Workshop on Program Comprehension (IWPC'02)*, pages 219–228, June 2002.
- [Mor04] Andrew Moran. Report on the first commercial users of functional programming workshop. *ACM SIGPLAN Notices*, 39(12):17–20, December 2004.
- [MP05] Brian A. Malloy and James F. Power. Exploiting uml dynamic object modeling for the visualization of C++ programs. *ACM Symposium on Software visualization (SoftVis'05)*, pages 105–114, May 2005.
- [MR00] Alessandro Maccari and Claudio Riva. Empirical evaluation of CASE tools usage at Nokia. *Journal on Empirical Software Engineering*, 5(3):287–299, November 2000.
- [MRR02] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Precise call graph construction in the presence of function pointers. *2nd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'02)*, pages 155–162, October 2002.

- [MS96] Edwin Morris and Dennis Smith, editors. *IEEE Recommended Practice for the Adoption of Computer-Aided Software Engineering (CASE) Tools (IEEE Std 1348-1995)*. IEEE Computer Society, December 1996.
- [MS99] Mark Marchukov and Kevin J. Sullivan. Reconciling behavioral mismatch through component restriction. Technical Report CS-99-22, Department of Computer Science, University of Virginia, July 1999.
- [MS02] Rym Mili and Renee Steiner. Software engineering: Introduction. In S. Diehl, editor, *Software Visualization*, volume 2269 of *Lecture Notes in Computer Science*, pages 129–137. Springer-Verlag, 2002.
- [MSB⁺02] M. Morisio, C. B. Seaman, V. R. Basili, A. T. Parra, S. E. Kraft, and S. E. Condon. COTS-based software development: Processes and open issues. *Journal of Systems and Software*, 61(3):189–199, April 2002.
- [MSC⁺01] Spiros Manchoridis, Timothy S. Sounder, Yih-Farn Chen, Demden R. Gansner, and Jeffrey L. Korn. REportal: A web-based portal site for reverse engineering. *8th IEEE Working Conference on Reverse Engineering (WCRE'01)*, pages 221–230, October 2001.
- [MSM01] Jeff Michaud, Margaret-Anne Storey, and Hausi Müller. Integrating information sources for visualizing java programs. *17th IEEE International Conference on Software Maintenance (ICSM'01)*, pages 250–258, November 2001.
- [MSP⁺00] M. Morisio, C. B. Seaman, A. T. Parra, V. R. Basili, S. E. Kraft, and S. E. Condon. Investigating and improving a COTS-based software development process. *22nd ACM/IEEE International Conference on Software Engineering (ICSE'00)*, pages 32–41, June 2000.
- [MSW⁺94] John Mylopoulos, Martin Stanley, Kenny Wong, Morris Bernstein, Renato De Mori, Graham Ewart, Kostas Kontogiannis, Ettore Merlo, Hausu Muller, Scott Tilley, and Marijana Tomic. Towards an integrated toolset for program understanding. *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'94)*, pages 19–31, October 1994.
- [MSW⁺02] Hausi A. Müller, Margaret-Anne Storey, Anke Weber, Will Kastelic, Holger Kienle, Qin Zhu, Jun Ma, Fang Yang, David Zwiers, Ken Wong, and Jon Pipitone. Adoption-centric software engineering. *CASCON 2002 Technology Exhibition Handout*, 2002. http://www.acse.cs.uvic.ca/downloads/info/flyer_screen.pdf.
- [MT02] Maurizio Morisio and Marco Torchiano. Definition and classification of COTS: a proposal. In J. Dean and A. Gravel, editors, *1st International Conference on COTS-Based Software Systems (ICCBSS'02)*, volume 2255 of *Lecture Notes in Computer Science*, pages 165–175. Springer-Verlag, 2002.
- [MT04] Tom Mens and Tom Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, February 2004.
- [MTO⁺92] H. A. Müller, S. R. Tilley, M. A. Orgun, B. D. Corrie, and N. H. Madhavji. A reverse engineering environment based on spatial and visual software interconnection models. *5th ACM SIGSOFT Symposium on Software Development Environments (SDE 5)*, pages 88–98, December 1992.
- [Mül86] Hausi A. Müller. *Rigi - A Model for Software System Construction, Integration, and Evolution based on Module Interfaces Specifications*. PhD thesis, Rice University, Houston, Texas, August 1986.
- [Mül98] Hausi A. Müller. Criteria for success of an exchange format. CASCON Workshop Meeting, November 1998. http://plg.uwaterloo.ca/~holt/sw.eng/exch.format/criteria_muller.html.
- [Mül01] Hausi Müller. Leveraging cognitive support and modern platforms for adoption-centric reverse engineering (ACRE). Grant application, University of Victoria, November 2001.

- [Mül05] Martin Müller. Implementierung einer Anfrageschnittstelle für ASIS zur Generierung von IML. Studienarbeit, Universität Stuttgart, April 2005. <http://elib.uni-stuttgart.de/opus/volltexte/2005/2269/>.
- [MW03] Daniel L. Moise and Kenny Wong. An industrial experience in reverse engineering. *10th IEEE Working Conference on Reverse Engineering (WCRE'03)*, pages 275–284, November 2003.
- [MW04a] Chunyan Meng and Kenny Wong. A GXL schema for story diagrams. In J.-M. Favre, M. Godfrey, and A. Winter, editors, *International Workshop on Meta-Models and Schemas for Reverse Engineering (ateM'03)*, volume 94, pages 29–38. Elsevier, May 2004.
- [MW04b] Daniel L. Moise and Kenny Wong. Issues in integrating schemas for reverse engineering. In J.-M. Favre, M. Godfrey, and A. Winter, editors, *International Workshop on Meta-Models and Schemas for Reverse Engineering (ateM'03)*, volume 94, pages 81–91. Elsevier, May 2004.
- [MW05] Daniel L. Moise and Kenny Wong. Extracting and representing cross-language dependencies in diverse software systems. *12th IEEE Working Conference on Reverse Engineering (WCRE'05)*, pages 209–218, November 2005.
- [MWHH06] Daniel L. Moise, Kenny Wong, H. James Hoover, and Daqing Hou. Reverse engineering scripting language extensions. *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 295–306, June 2006.
- [MWS98] Hausi A. Müller, Kenny Wong, and Margaret-Anne Storey. Reverse engineering research should target cooperative information system requirements. *5th IEEE Working Conference on Reverse Engineering (WCRE'98)*, page 255, October 1998.
- [MWS04] Daniel L. Moise, Kenny Wong, and Dabo Sun. Integrating a reverse engineering tool with Microsoft Visual Studio .NET. *8th IEEE European Conference on Software Maintenance and Reengineering (CSMR'04)*, pages 85–92, March 2004.
- [MWW03] Hausi A. Müller, Anke Weber, and Ken Wong. Leveraging cognitive support and modern platforms for adoption-centric reverse engineering (ACRE). *3rd International Workshop on Adoption-Centric Software Engineering (ACSE'03)*, pages 30–35, May 2003.
- [MXBK05] David Mandelin, Lin Xu, Rastislav Bodik, and Doug Kimelman. Jungloid mining: Helping to navigate the api jungle. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, pages 48–51, June 2005.
- [MZG03] Hong Mei, Wei Zhang, and Fang Gu. A feature oriented approach to modeling and reusing requirements of software product lines. *27th IEEE International Computer Software and Applications Conference (COMPSAC'03)*, pages 250–256, November 2003.
- [Nac97] Lee R. Nackman. Codestore and incremental C++. *Dr. Dobbs' Journal*, pages 92–96, December 1997. <http://www.ddj.com/documents/s=934/ddj9712k/>.
- [NDG05] Oscar Nierstrasz, Stephane Ducasse, and Tudor Girba. The story of Moose: an agile reengineering environment. *10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*, pages 1–10, September 2005.
- [NEFZ00] Christian Nentwich, Wolfgang Emmerich, Anthony Finkelstein, and Andrea Zisman. BOX: Browsing objects in XML. *Software—Practice and Experience*, 30(15):1661–1676, December 2000.
- [Nei04] Sascha Neinert. Extraktion statischer Abhängigkeiten aus Ada95-Programmen mittels ASIS. Diplomarbeit, Universität Stuttgart, December 2004. http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2200.

- [New88] Frances J. Newbery. An interface description language for graph editors. *IEEE Symposium on Visual Languages (VL'88)*, pages 144–149, October 1988.
- [NH00] Penelope A. Noe and Thomas C. Hartrum. Extending the notation of Rational Rose 98 for use with formal methods. *National Aerospace and Electronics Conference (NAECON'00)*, pages 43–50, October 2000.
- [NK94] Stephen C. North and Eleftherios Koutsoufios. Applications of graph visualization. *Graphics Interface'94*, pages 235–245, 1994.
- [NM93] Philip Newcomb and Lawrence Markosian. Automating the modularization of large cobol programs: Application of an enabling technology for reengineering. *1st IEEE Working Conference on Reverse Engineering (WCRE'93)*, pages 222–230, May 1993.
- [Nor98] Donald A. Norman. *The Invisible Computer*. The MIT Press, 1998.
- [Nør00a] Kurt Nørmark. Requirements for an elucidative programming environment. *8th IEEE International Workshop on Program Comprehension (IWPC'00)*, pages 119–128, June 2000.
- [Nor00b] Christopher Loy North. *A User Interface for Coordinating Visualizations Based on Relational Schemata: Snap-Together Visualization*. PhD thesis, University of Maryland College Park, 2000.
- [Nor02] Linda M. Northrop. Sei's software product line tenets. *IEEE Software*, 19(4):32–40, July/August 2002.
- [NP04] Stig Nordheim and Tero Päivärinta. Customization of enterprise content management systems: An exploratory case study. *37th IEEE Hawaii International Conference on System Sciences (HICSS'04)*, January 2004.
- [NS00] Chris North and Ben Shneiderman. Snap-together visualization: can users construct and operate coordinated visualizations? *International Journal of Human-Computer Studies*, 53(5):715–739, November 2000.
- [NT06] Robert L. Nord and James E. Tomayko. Software architecture-centric methods and agile development. *IEEE Software*, 23(2):47–53, March/April 2006.
- [NW02] E. W. T. Ngai and F. K. T. Wat. A literature review and classification of electronic commerce research. *Information & Management*, 39(5):415–429, March 2002.
- [O'B01] Liam O'Brien. Architecture reconstruction to support a product line effort: Case study. Technical Note CMU/SEI-2001-TN-015, Software Engineering Institute, Carnegie Mellon University, July 2001. <http://www.sei.cmu.edu/pub/documents/01.reports/pdf/01tn015.pdf>.
- [OBB06] Stephen Owens, David Budgen, and Pearl Brereton. Protocol analysis: A neglected practice. *Communications of the ACM*, 49(2):117–122, February 2006.
- [Off02] Jeff Offutt. Quality attributes of Web software applications. *IEEE Software*, 19(2):25–32, March/April 2002.
- [OHSS02] Liam O'Brien, Fred Hansen, Robert Seacord, and Dennis Smith. Mining and managing software assets. *10th International Workshop on Software Technology and Engineering Practice (STEP'02)*, pages 82–90, October 2002.
- [OJ97] Rober O'Callahan and Daniel Jackson. Lackwit: A program understanding tool based on type inference. *19th ACM/IEEE International Conference on Software Engineering (ICSE'97)*, pages 338–348, May 1997.
- [Ols06] Greg Olsen. From COM to common. *ACM Queue*, 4(5):20–26, June 2006.
- [Ope96] The Open Group. *Preliminary Specification: Architecture Neutral Distribution Format (XANDF)*, January 1996. <http://www.opengroup.org/publications/catalog/p527.htm>.

- [OT03] Liam O'Brien and Vorachat Tamarree. Architecture reconstruction of J2EE applications: Generating views from the module viewtype. Technical Note CMU/SEI-2003-TN-028, Software Engineering Institute, Carnegie Mellon University, November 2003. <http://www.sei.cmu.edu/pub/documents/03.reports/pdf/03tn028.pdf>.
- [Ous98] John K. Ousterhout. Scripting: Higher-level programming for the 21st century. *IEEE Computer*, 31(3):23–70, March 1998.
- [Par72] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.
- [Par78] David Lorge Parnas. Designing software for ease of extension and contraction. *3rd ACM/IEEE International Conference on Software Engineering (ICSE'78)*, pages 264–277, May 1978.
- [Par79] David Lorge Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128–137, March 1979.
- [Par92] Burt Parker. Introducing EIA-CDIF: The case data interchange format standard. *2nd IEEE Symposium on Assessment of Quality Software Development Tools*, pages 74–82, May 1992.
- [Par99] Sang Bong Park. A study on the CAD/CAM system of transfer die for deep drawing process. *IEEE International Symposium on Assembly and Task Planning (ISATP'99)*, pages 21–24, July 1999.
- [Pat05] David A. Patterson. 20th century vs. 21st century c&c: The SPUR manifesto. *Communications of the ACM*, 48(3):15–16, March 2005.
- [PBS93] Blaine A. Price, Ronald M Baecker, and Ian S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, September 1993.
- [PC86] David L. Parnas and Paul C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12(2):251–257, February 1986.
- [PC04] Dan Port and Scott Chen. Assessing COTS assessment: How much is enough? In R. Kazman and D. Ports, editors, *3rd International Conference on COTS-Based Software Systems (ICCBSS'04)*, volume 2959 of *Lecture Notes in Computer Science*, pages 183–198. Springer-Verlag, 2004.
- [PCH93] J. S. Poulin, J. M. Caruso, and D. R. Hancock. The business case for software reuse. *IBM Systems Journal*, 32(4):567–594, April 1993.
- [Per00] Stephen Perelgut. The case for a single data exchange format. *7th IEEE Working Conference on Reverse Engineering (WCRE'00)*, pages 281–283, November 2000.
- [Pet95] M. Petre. Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44, June 1995.
- [PFGJ02] Martin Pinzger, Michael Fischer, Harald Gall, and Mehdi Jazayeri. Revealer: A lexical pattern matcher for architecture recovery. *9th IEEE Working Conference on Reverse Engineering (WCRE'02)*, pages 170–178, November 2002.
- [PFK96] Allison L. Powell, James C. French, and John C. Knight. A systematic approach to creating and maintaining software documentation. *11th ACM Symposium on Applied Computing (SAC'96)*, pages 201–208, 1996.
- [Pfl99] Shari Lawrence Pfleeger. Understanding and improving technology transfer in software engineering. *Journal of Systems and Software*, 47(2-3):111–124, July 1999.
- [PG05] J. Ponzio and O. Gruber. Integrating Web technologies in Eclipse. *IBM Systems Journal*, 44(2):279–288, February 2005.
- [Pie94] Kurt Piersol. A close-up of OpenDoc. *Byte*, pages 183–188, March 1994. <http://www.byte.com/art/9403/sec9/art1.htm>.

- [Pin97] Steven Pinker. *How the Mind Works*. W. W. Norton, 1997.
- [PJAA96] Stanley R. Page, Todd J. Johnsgard, Uhl Albert, and C. Dennis Allen. User customization of a word processor. *ACM SIGCHI Conference on Human Factors in Computing Systems (CHI'96)*, pages 340–346, April 1996.
- [PK02] Jon Pipitone and Holger M. Kienle. The SVG visualization engine. *Presentation at CASCON 2002 SVG Workshop*, October 2002. <https://www.ibm.com/ibm/cas/archives/2002/workshops/svg.shtml>.
- [PLL04] Thomas Panas, Jonas Lundberg, and Welf Löwe. Reuse in reverse engineering. *12th IEEE International Workshop on Program Comprehension (IWPC'04)*, pages 52–61, June 2004.
- [PM02] James F. Power and Brian A. Malloy. Program annotation in xml: a parse-tree based approach. *9th IEEE Working Conference on Reverse Engineering (WCRE'02)*, pages 190–198, November 2002.
- [PMCS05] Marsha Pomeroy-Huff, Julia Mullaney, Robert Cannon, and Mark Sebern. The personal software process(osp) body of knowledge. Special Report CMU/SEI-2005-SR-003, Software Engineering Institute, Carnegie Mellon University, August 2005. <http://www.sei.cmu.edu/pub/documents/05.reports/pdf/05sr003.pdf>.
- [PMDS05] Denys Poshyvanyk, Andrian Marcus, Yubo Dong, and Andrey Sergeev. IRiSS—a source code exploration tool. *ICSM 2005 Industrial & Tool Proceedings*, pages 69–72, September 2005.
- [POG03] Martin Pinzger, Johann Oberleitner, and Harald Gall. Analyzing and understanding architectural characteristics of COM+ components. *11th IEEE International Workshop on Program Comprehension (IWPC'03)*, pages 54–63, May 2003.
- [PP94] Santanu Paul and Atul Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, June 1994.
- [PP96] Santanu Paul and Atul Prakash. A query algebra for program databases. *IEEE Transactions on Software Engineering*, 22(3):202–217, March 1996.
- [PR02] Thomas Pfarr and James E. Reis. The integration of COTS/GOTS within NASA's HST command and control system. In J. Dean and A. Gravel, editors, *1st International Conference on COTS-Based Software Systems (ICBSS'02)*, volume 2255 of *Lecture Notes in Computer Science*, pages 209–221. Springer-Verlag, 2002.
- [PR04] Jinsoo Park and Sudha Ram. Information systems interoperability: What lies beneath? *ACM Transactions on Office Information Systems*, 22(4):595–632, October 2004.
- [PRBH91] Santanu Paul, Atul Prakash, Erich Buss, and John Henshaw. Theories and techniques of program understanding. *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'91)*, pages 37–53, October 1991.
- [Pre00] Roger S. Pressman. What a tangled Web we weave. *IEEE Software*, 17(1):18–21, January/February 2000.
- [Pri93] Ruben Prieto-Diaz. Status report: Software reusability. *IEEE Software*, 10(3):61–66, May 1993.
- [PS98] Frantisek Plasil and Michael Stal. An architectural view of distributed objects and components in CORBA, JavaRMI and COM/DCOM. *Software – Concepts & Tools*, 19(1):14–28, June 1998.
- [PS05] Thomas Panas and Miroslaw Staron. Evaluation of a framework for reverse engineering tool construction. *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 145–154, September 2005.
- [PSK⁺04] P. Popov, L. Strigini, A. Kostov, V. Mollov, and D. Selenskyo. Software fault-tolerance with off-the-shelf SQL servers. In R. Kazman and D. Ports, editors, *3rd International Conference on COTS-Based Software Systems (ICBSS'04)*, volume 2959 of *Lecture Notes in Computer Science*, pages 117–126. Springer-Verlag, 2004.

- [PT90] Frances Newbery Paulish and Walter F. Tichy. Edge: An extensible graph editor. *Software—Practice and Experience*, 20(S1):S1/63–S1/88, June 1990.
- [PT02] Robert Pierce and Scott Tilley. Automatically connecting documentation to code with rose. *20th ACM International Conference on Computer Documentation (SIG-DOC'02)*, pages 157–163, October 2002.
- [Put04] Erik Putrycz. Using trace analysis for improving performance in COTS systems. *Conference of The Centre for Advanced Studies On Collaborative Research (CAS-CON'04)*, pages 68–80, October 2004.
- [PY04] Dan Port and Ye Yang. Empirical analysis of COTS activity effort sequences. In R. Kazman and D. Ports, editors, *3rd International Conference on COTS-Based Software Systems (ICBSS'04)*, volume 2959 of *Lecture Notes in Computer Science*, pages 169–182. Springer-Verlag, 2004.
- [Rai97] Vaclav Rajlich. Incremental redocumentation with hypertext. *23rd IEEE EURO-MICRO Conference (EUROMICRO'97)*, pages 68–72, March 1997.
- [Ras00] Jef Raskin, editor. *The Humane Interface: New Directions for Designing Interactive Systems*. Addison-Wesley, April 2000.
- [Rat01] Rational. Using the Rose extensibility interface, 2001.
- [RBBC03] Donald J. Reifer, Victor R. Basili, Barry W. Boehm, and Betsy Clark. Eight lessons learned during COTS-based systems maintenance. *IEEE Software*, 20(5):94–96, September/October 2003.
- [RBBC04] Donald J. Reifer, Victor R. Basili, Barry W. Boehm, and Betsy Clark. COTS-based systems—twelve lessons learned about maintenance. In R. Kazman and D. Ports, editors, *3rd International Conference on COTS-Based Software Systems (ICBSS'04)*, volume 2959 of *Lecture Notes in Computer Science*, pages 137–145. Springer-Verlag, 2004.
- [RC05] Rui Rodrigues and Joao M. P. Cardoso. An infrastructure to functionally test designs generated by compilers targeting FPGAs. *Design, Automation and Test in Europe Conference and Exhibition (DATE'05)*, pages 30–31, March 2005.
- [RCdBL03] Pablo Romero, Richard Cox, Benedict du Boulay, and Rudi Lutz. A survey of external representations employed in object-oriented programming environments. *Journal of Visual Languages and Computing*, 14(5):387–419, October 2003.
- [RDL04] Matthias Rieger, Stephane Ducasse, and Michele Lanza. Insights into system-wide code duplication. *11th IEEE Working Conference on Reverse Engineering (WCRE'04)*, pages 100–109, November 2004.
- [RE04] Michel Ruffin and Christof Ebert. Using open source software in product development: A primer. *IEEE Software*, 21(1):82–86, January/February 2004.
- [Rei90] Steven P. Reiss. Connecting tools using message passing in the Field environment. *IEEE Software*, 7(4):57–66, July 1990.
- [Rei93] Steven P. Reiss. A framework for abstract 3D visualization. *IEEE Symposium on Visual Languages (VL'93)*, pages 108–115, August 1993.
- [Rei95a] Steven P. Reiss. An engine for the 3D visualization of program information. *Journal of Visual Languages and Computing*, 6(3):299–323, September 1995.
- [Rei95b] Steven P. Reiss. Program editing in a software development environment (draft). <http://www.cs.brown.edu/~spr/research/desert/fredpaper.pdf>, 1995.
- [Rei96] Steven P. Reiss. Simplifying data integration: The design of the desert software development environment. *18th ACM/IEEE International Conference on Software Engineering (ICSE'96)*, pages 398–407, May 1996.
- [Rei99] Steven P. Reiss. The Desert environment. *ACM Transactions on Software Engineering and Methodology*, 8(4):297–342, October 1999.

- [Rei01] Steven P. Reiss. An overview of BLOOM. *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, pages 38–45, June 2001.
- [Rei02] Steven P. Reiss. Constraining software evolution. *18th IEEE International Conference on Software Maintenance (ICSM'02)*, pages 162–171, October 2002.
- [Rei05] Steven P. Reiss. Incremental maintenance of software artifacts. *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 113–122, September 2005.
- [RGH05] Gregorio Robles, Jesus M. Gonzalez-Barahona, and Israel Herraiz. An empirical approach to software archaeology. *ICSM 2005 Poster Proceedings*, pages 47–50, September 2005.
- [RHD02] Cynthia K. Riemenschneider, Bill C. Hardgrave, and Fred D. Davis. Explaining software developer acceptance of methodologies: A comparison of five theoretical models. *IEEE Transactions on Software Engineering*, 28(12):1135–1145, December 2002.
- [Ric97] Robert Richardson. Components battling components. *Byte*, November 1997. <http://www.byte.com/art/9711/sec6/art6.htm>.
- [Ric03] Filippo Ricca. *Analysis, Testing, and Re-structuring of Web Applications*. PhD thesis, Universita di Genova, Italy, September 2003.
- [Rif03] Stan Rifkin. Two good reasons why new software processes are not adopted. *3rd International Workshop on Adoption-Centric Software Engineering (ACSE'03)*, pages 23–29, May 2003.
- [Riv00a] Claudio Riva. Reverse architecting: an industrial experience report. *7th IEEE Working Conference on Reverse Engineering (WCRE'00)*, pages 42–50, November 2000.
- [Riv00b] Claudio Riva. Reverse architecting: Suggestions for an exchange format. *Workshop on Standard Exchange Format (WoSEF)*, June 2000.
- [Riv04] Claudio Riva. *View-based Software Architecture Reconstruction*. PhD thesis, Vienna University of Technology, October 2004.
- [RLS⁺03] Derek Rayside, Marin Litoiu, Margaret-Anne Storey, Casey Best, and Robert Lintern. Visualizing flow diagrams in WebSphere Studio using SHriMP views. *Information Systems Frontiers*, 5(2):161–174, April 2003.
- [RLSB01] Derek Rayside, Marin Litoiu, Margaret-Anne Storey, and Casey Best. Integrating SHriMP with the IBM WebSphere Studio workbench. *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'01)*, pages 79–93, November 2001.
- [RLSB02] Ioana Rus, Mikael Lindvall, Carolyn Seaman, and Victor Basili. Packaging and disseminating lessons learned from COTS-based software development. *27th NASA Goddard/IEEE Software Engineering Workshop (SEW-27'02)*, pages 131–138, December 2002.
- [Rog95] Everett M. Rogers. *Diffusion of Innovations*. The Free Press, fourth edition, 1995.
- [RPR93] Howard Reubenstein, Richard Piazza, and Susan Roberts. Separating parsing and analysis in reverse engineering. *1st IEEE Working Conference on Reverse Engineering (WCRE'93)*, pages 117–125, May 1993.
- [RR85] Samuel T. Redwine and William E. Riddle. Software technology maturation. *8th ACM/IEEE International Conference on Software Engineering (ICSE'85)*, pages 189–200, August 1985.
- [RR02] Claudio Riva and Jordi Vidal Rodriguez. Combining static and dynamic views for architecture reconstruction. *6th IEEE European Conference on Software Maintenance and Reengineering (CSMR'02)*, pages 47–55, March 2002.

- [RR03] T. Ravichandran and Marcus A. Rothenberger. Software reuse strategies and component markets. *Communications of the ACM*, 46(8):109–114, August 2003.
- [RS04] Spencer Rugaber and Kurt Stirewalt. Model-driven reverse engineering. *IEEE Software*, 21(4):45–53, July/August 2004.
- [RT00] Filippo Ricca and Paolo Tonella. Web site analysis: Structure and evolution. *16th IEEE International Conference on Software Maintenance (ICSM'00)*, pages 76–86, October 2000.
- [RT01a] Filippo Ricca and Paolo Tonella. Analysis and testing of Web applications. *23rd ACM/IEEE International Conference on Software Engineering (ICSE'01)*, pages 25–34, May 2001.
- [RT01b] Filippo Ricca and Paolo Tonella. Understanding and restructuring Web sites with ReWeb. *IEEE MultiMedia*, 8(2):40–51, April–June 2001.
- [RTB02] Filippo Ricca, Paolo Tonella, and Ira D. Baxter. Web application transformations based on rewrite rules. *Information and Software Technology*, 44(13):811–825, October 2002.
- [RTR⁺00] Matti Rossi, Juha-Pekka Tovanen, Balasubramaniam Rameshn, Kalle Lyytinen, and Janne Kaipala. Method rationale in method engineering. *33rd IEEE Hawaii International Conference on System Sciences (HICSS'00)*, 2, January 2000.
- [Rug96] Spencer Rugaber. Program understanding. In Allen Kent and James G. Williams, editors, *Encyclopedia of Computer Science and Technology*, pages 341–368. Marcel Dekker, 1996.
- [Rut01] Lloyd Rutledge. Multimedia standards: Building blocks of the web. *IEEE MultiMedia*, 8(3):13–15, July–September 2001.
- [RW96] Spencer Rugaber and Linda M. Wills. Creating a research infrastructure for reengineering. *3rd IEEE Working Conference on Reverse Engineering (WCRE'96)*, pages 98–102, November 1996.
- [RY02] Claudio Riva and Yaojin Yang. Generation of architectural documentation using XML. *9th IEEE Working Conference on Reverse Engineering (WCRE'02)*, pages 161–169, October 2002.
- [Sal94] Peter H. Salus. *A Quarter Century of UNIX*. UNIX and Open Systems Series. Addison-Wesley, 1994.
- [Sal98] Peter H. Salus, editor. *Little Languages and Tools*, volume III of *Handbook of Programming Languages*. MacMillan Technical Publishing, 1998.
- [Sam97] Johannes Sametinger. *Software Engineering with Reusable Components*. Springer-Verlag, 1997.
- [SBM01] Margaret-Anne Storey, Casey Best, and Jeff Michaud. SHriMP views: An interactive environment for exploring Java programs. *9th IEEE International Workshop on Program Comprehension (IWPC'01)*, pages 111–112, May 2001.
- [SC97] Mary Shaw and Paul Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. *27th IEEE International Computer Software and Applications Conference (COMPSAC'03)*, pages 6–14, August 1997.
- [Sca01] Walt Scacchi. Process models in software engineering. In J. J. Marciniak, editor, *Encyclopedia of Software Engineering*. John Wiley & Sons, December 2001. <http://www.ics.uci.edu/~wscacchi/Papers/SE-Encyc/Process-Models-SE-Encyc.pdf>.
- [SCD03] Nikita Synytskyy, James R. Cordy, and Thomas R. Dean. Robust multilingual parsing using island grammars. *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'03)*, pages 266–278, October 2003.

- [SCG05] Margaret-Anne D. Storey, Davor Cubranic, and Daniel M. German. On the use of visualization to support awareness of human activities in software development: A survey and a framework. *ACM Symposium on Software visualization (SoftVis'05)*, pages 193–202, May 2005.
- [Sch96] Kurt Schneider. Prototypes as assets, not toys: Why and how to extract knowledge from prototypes. *18th ACM/IEEE International Conference on Software Engineering (ICSE'96)*, pages 522–531, May 1996.
- [Sch98] Richard Schwaninger. Rapid prototyping with Tcl/Tk. *Linux Journal*, May 1998. <http://www.linuxjournal.com/article/2172>.
- [Sch04] Michael Schrage. Never go to a client meeting without a prototype. *IEEE Software*, 21(2):42–45, March/April 2004.
- [SCHC99] Susan Elliott Sim, Charles L. A. Clarke, Richard C. Holt, and Anthony M. Cox. Browsing and searching software architectures. *15th IEEE International Conference on Software Maintenance (ICSM'99)*, pages 381–390, August 1999.
- [SCZC97] Kevin J. Sullivan, Jake Cockrell, Shengton Zhang, and David Coppit. Package-oriented programming of engineering tools. *19th ACM/IEEE International Conference on Software Engineering (ICSE'97)*, pages 616–617, May 1997.
- [SDC99] Kevin J. Sullivan, Joanne Bechta Dugan, and David Coppit. The galileo fault tree analysis tool. *12th International Symposium on Fault-Tolerant Computing (FTCS'99)*, pages 232–237, June 1999.
- [SDF⁺03] S. Shavor, J. D'Anjou, S. Fairborther, D. Kehn, J. Kellerman, and P. McCarthy. Swing interoperability. In *The Java Developer's Guide to Eclipse*, chapter 25, pages 545–554. Addison-Wesley, 2003.
- [Sec05] Security Space. Web authoring tools report, March 2005. http://www.securityspace.com/s_survey/data/man.200502/webauth.html.
- [SEH03] Susan Elliott Sim, Steve Easterbrook, and Richard C. Holt. Using benchmarking to advance research: A challenge to software engineering. *25th ACM/IEEE International Conference on Software Engineering (ICSE'03)*, pages 74–83, May 2003.
- [SEK⁺99] E. Stroulia, M. El-Ramly, L. Kong, P. Sorenson, and B. Matichuk. Reverse engineering legacy interfaces: An interaction-driven approach. *6th IEEE Working Conference on Reverse Engineering (WCRE'99)*, pages 292–302, October 1999.
- [SES05] Janice Singer, Robert Elves, and Margaret-Anne Storey. Navtracks: Supporting navigation in software maintenance. *ICSM 2005 Industrial & Tool Proceedings*, pages 325–334, September 2005.
- [SEU04] Cara Stein, Letha Etzkorn, and Dawn Utley. Computing software metrics from design documents. *42nd ACM Southeast Regional Conference (ACM-SE 42)*, pages 146–151, April 2004.
- [SFM99a] Margaret-Anne D. Storey, F. David Fracchia, and Hausi A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Systems and Software*, 44(3):171–185, January 1999.
- [SFM99b] Margaret-Anne D. Storey, F. David Fracchia, and Hausi A. Müller. Customizing a fisheye view algorithm to preserve the mental map. *Journal of Visual Languages and Computing*, 10(3):245–267, June 1999.
- [SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
- [Sha90] Mary Shaw. Prospects for an engineering discipline of software. *IEEE Software*, 7(6):15–24, November 1990.
- [Sha95] Mary Shaw. Architectural issues in software reuse: It's not just the functionality, it's the packaging. *Symposium on Software Reusability (SSR'95)*, pages 3–6, April 1995.

- [Sha01] Mary Shaw. The coming-of-age of software architecture research. *23rd ACM/IEEE International Conference on Software Engineering (ICSE'01)*, pages 657–664a, May 2001.
- [Sha03] Mary Shaw. Writing good software engineering research papers: Minitutorial. *25th ACM/IEEE International Conference on Software Engineering (ICSE'03)*, pages 726–736, May 2003.
- [SHD05] Nikita Synytskyy, Richard C. Holt, and Ian Davis. Browsing software architectures with LSEdit. *13th IEEE International Workshop on Program Comprehension (IWPC'05)*, pages 176–178, May 2005.
- [SHE02] Susan Elliott Sim, Richard C. Holt, and Steve Easterbrook. On using a benchmark to evaluate C++ extractors. *10th IEEE International Workshop on Program Comprehension (IWPC'02)*, pages 114–123, June 2002.
- [Shn80] Ben Shneiderman. *Software Psychology: Human Factors in Computer and Information Systems*. Winthrop Publishers, 1980.
- [Sim00] Susan Elliott Sim. Next generation data interchange: Tool-to-tool application program interfaces. *7th IEEE Working Conference on Reverse Engineering (WCRE'00)*, pages 278–280, November 2000.
- [Sim03] Susan Elliott Sim. *A Theory of Benchmarking with Applications to Software Reverse Engineering*. PhD thesis, University of Toronto, October 2003.
- [Sin03] C. A. Singer. Context-specific intellectual capital—the next link in the knowledge chain. *IBM Systems Journal*, 42(3):446–461, March 2003.
- [Sin04] Janice Singer. Creating software engineering tools that are useable, useful, and actually used. Talk given at the University of Victoria, December 2004.
- [SJG03] Sihem Ben Sassi, Lamia Labeled Jilani, and Henda Hajjami Ben Ghezala. COTS characterization model in a COTS-based development environment. *10th IEEE Asia-Pacific Software Engineering Conference (APSEC'03)*, pages 352–361, 2003.
- [SK92] Torbjorn Skramstad and Md. Khaled Khan. Assessment of reverse engineering tools: A MECCA approach. *2nd IEEE Symposium on Assessment of Quality Software Development Tools*, pages 120–126, May 1992.
- [SK96] Kevin J. Sullivan and John C. Knight. Experience assessing an architectural approach to large-scale systematic reuse. *18th ACM/IEEE International Conference on Software Engineering (ICSE'96)*, pages 220–229, May 1996.
- [SK01] Susan Elliott Sim and Rainer Koschke. Wosef: Workshop on standard exchange format. *ACM SIGSOFT Software Engineering Notes*, 26(1):44–49, January 2001.
- [SKBS97] Danny Soroker, Michael Karasick, John Barton, and David Streeter. Extension mechanisms in Montana. *8th Israeli Conference on Computer Systems and Software Engineering (CSSE'97)*, pages 119–128, June 1997.
- [SKC⁺96] Kevin J. Sullivan, John C. Knight, Jake Cockrell, , and Shengtong Zhang. Product development with massive components. *21st NASA Goddard Space Flight Center Software Engineering Workshop*, December 1996.
- [SKM01] Tarja Systä, Kai Koskimies, and Hausi Müller. Shimba—an environment for reverse engineering Java software systems. *Software—Practice and Experience*, 31(4):371–394, 2001.
- [SL96] Janice Singer and Timothy C. Lethbridge. Methods for studying maintenance activities. *2nd Workshop on Empirical Studies of Software Maintenance (WESS'96)*, pages 105–109, November 1996.
- [SL98] Janice Singer and Timothy Lethbridge. Studying work practices to assist tool design in software engineering. *6th IEEE International Workshop on Program Comprehension (IWPC'98)*, pages 173–179, June 1998.

- [SLVA97] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. An examination of software engineering work practices. *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'97)*, pages 209–223, November 1997.
- [Sly93] David Sly. AutoCAD-based industrial layout planning and material flow analysis in FactoryFLOW and Plan. *IEEE Winter Simulation Conference (WSC'93)*, pages 281–284, December 1993.
- [Sly98] Dave Sly. Object-oriented factory layout in AutoCAD. *IEEE Winter Simulation Conference (WSC'98)*, pages 275–277, December 1998.
- [SM86] Sidney L. Smith and Jane N. Mosier. Guidelines for designing user interface software. Technical Report ESD-TR-86-278, The MITRE Corporation, August 1986. <http://www.dfki.de/~jameson/hcida/papers/smith-mosier.pdf>.
- [SM04] Ahmed Seffah and Eduard Metzker. The obstacles and myths of usability and software engineering. *Communications of the ACM*, 47(12):71–76, December 2004.
- [SMC04] Dennis Smith, Edwin Morris, and David Carney. Adoption centric problems in the context of systems of systems interoperability. *4th International Workshop on Adoption-Centric Software Engineering (ACSE'04)*, pages 63–68, May 2004.
- [Sne95] Harry M. Sneed. Reverse engineering as a bridge to CASE. *7th IEEE International Workshop on Computer-Aided Software Engineering (CASE'95)*, pages 304–317, July 1995.
- [Sne97] Harry M. Sneed. Measuring the performance of a software maintenance department. *23rd IEEE EUROMICRO Conference (EUROMICRO'97)*, pages 119–127, March 1997.
- [Sne01] Harry M. Sneed. Extracting business logic from existing COBOL programs as a basis for redevelopment. *9th IEEE International Workshop on Program Comprehension (IWPC'01)*, pages 167–175, May 2001.
- [Sne04] Harry M. Sneed. Reengineering reports. *11th IEEE Working Conference on Reverse Engineering (WCRE'04)*, pages 17–26, November 2004.
- [Som05a] Ian Sommerville. Integrated requirements engineering: A tutorial. *IEEE Software*, 22(1):16–23, January/February 2005.
- [Som05b] Ian Sommerville. Software construction by configuration: Challenges for software engineering research. *21st IEEE International Conference on Software Maintenance (ICSM'05)*, page 9, September 2005.
- [Spa00] Michael Sparling. Lessons learned through six years of component-based development. *Communications of the ACM*, 43(10):47–53, October 2000.
- [Spe05] C. M. Sperberg-McQueen. XML and semi-structured data. *ACM Queue*, 3(8):34–41, October 2005.
- [Spi02] Diomidis Spinellis. Unix tools as visual programming components in a GUI-builder environment. *Software—Practice and Experience*, 32(1):57–71, January 2002.
- [Spi03] Diomidis Spinellis. *Code Reading: The Open Source Perspective*. Addison-Wesley, 2003. <http://www.spinellis.gr/codereading/>.
- [SPLY01] Giancarlo Succi, Witold Pedrycz, Eric Liu, and Jason Yip. Package-oriented software engineering: A generic architecture. *IT Pro*, 3(2):29–36, March/April 2001.
- [SS02] Pamela Samuelson and Suzanne Scotchmer. The law & economics of reverse engineering. *Yale Law Journal*, 111:1575ff, 2002.
- [SS04] Dimidis Spinellis and Clements Szyperski. How is open source affecting software development? *IEEE Software*, 21(1):28–33, January/February 2004.
- [SSJ⁺02] Inderjeet Singh, Beth Stearns, Mark Johnson, et al. *Designing Enterprise Applications with the J2EE Platform*. Addison-Wesley, second edition, March 2002. http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/.

- [SSK00] Guy St-Denis, Reinhard Schauer, and Rudolf K. Keller. Selecting a model interchange format: The SPOOL case study. *33rd IEEE Hawaii International Conference on System Sciences (HICSS'00)*, January 2000.
- [SSW02] Margaret-Anne D. Storey, Susan Elliott Sim, and Kenny Wong. A collaborative demonstration of reverse engineering tools. *ACM SIGAPP Applied Computing Review*, 10(1):18–25, Spring 2002.
- [Sta81] Richard M. Stallman. Emacs: The extensible, customizable self-documenting display editor. *ACM SIGPLAN SIGOA Symposium on Text Manipulation*, pages 147–156, June 1981.
- [Sta84] Thomas A. Standish. An essay on software reuse. *IEEE Transactions on Software Engineering*, SE-10(5):494–497, September 1984.
- [Sto98] Margaret-Anne D. Storey. *A Cognitive Framework for Describing and Evaluating Software Exploration Tools*. PhD thesis, Simon Fraser University, December 1998.
- [Sto05] Margaret-Anne Storey. Theories, methods and tools in program comprehension: Past, present and future. *13th IEEE International Workshop on Program Comprehension (IWPC'05)*, pages 181–191, May 2005.
- [Sul97] Kevin J. Sullivan. Better, faster, cheaper tools: A case study and demonstration. *IEEE Reliability and Maintainability Symposium (RAMS'97)*, pages 216–219, January 1997.
- [Sul01] Kevin J. Sullivan. Designing models of modularity and integration. In George T. Heineman and William T. Councill, editors, *Component-Based Software Engineering: Putting the Pieces Together*, chapter 19, pages 341–354. Addison-Wesley, 2001.
- [SvGB05] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A taxonomy of variability realization techniques. *Software—Practice and Experience*, 35(8):705–754, July 2005.
- [SW04] Dabo Sun and Kenny Wong. On understanding software tool adoption using perceptual theories. *4th International Workshop on Adoption-Centric Software Engineering (ACSE'04)*, pages 51–55, May 2004.
- [SWF⁺96] M.-A. D. Storey, K. Wong, P. Fong, D. Hooper, K. Hopkins, and H. A. Müller. On designing an experiment to evaluate a reverse engineering tool. *3rd IEEE Working Conference on Reverse Engineering (WCRE'96)*, pages 31–40, November 1996.
- [SWM97] Margaret-Anne D. Storey, Kenny Wong, and Hausi A. Müller. How do program understanding tools affect how programmers understand programs. *4th IEEE Working Conference on Reverse Engineering (WCRE'97)*, pages 12–21, October 1997.
- [SYM00] Tarja Systä, Ping Yu, and Hausi Müller. Analyzing Java software by combining metrics and program visualization. *4th IEEE European Conference on Software Maintenance and Reengineering (CSMR'00)*, pages 199–208, February 2000.
- [Sys00a] Tarja Systä. *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*. PhD thesis, University of Tampere, Finland, May 2000.
- [Sys00b] Tarja Systä. Understanding the behavior of java programs. *7th IEEE Working Conference on Reverse Engineering (WCRE'00)*, pages 214–223, November 2000.
- [SZP96] H. Shen, J. Zucker, and D. L. Parnas. Table transformation tools: Why and how. *11th IEEE Conference on Computer Assurance (COMPASS'96)*, pages 3–11, June 1996.
- [Szy98] Clemens Szyperski. Emerging component software technologies—a strategic comparison. *Software – Concepts & Tools*, 19(1):2–10, June 1998.
- [Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, second edition, 2002.
- [TAB03] Barbara Tyson, Cecilia Albert, and Lisa Brownsword. Implications of using the capability maturity model integration (CMMI) for COTS-based systems. In H. Erdogmus and T. Weng, editors, *2nd International Conference on COTS-Based Software Systems (ICCBSS'03)*, volume 2580 of *Lecture Notes in Computer Science*, pages 229–239. Springer-Verlag, 2003.

- [TAFM97] P. Tonella, G. Antoniol, R. Fiutem, and E. Merlo. Points-to analysis for program understanding. *5th IEEE International Workshop on Program Comprehension (IWPC'97)*, pages 90–99, March 1997.
- [Tal03] Marcelo Tallis. Semantic word processing for content authors. *Knowledge Markup and Semantic Annotation Workshop (SEMANNOT'03)*, October 2003. <http://mr.teknowledge.com/daml/publications/semannot-2003-article.pdf>.
- [Tan87] Andrew S. Tanenbaum. A UNIX clone with source code for operating systems courses. *ACM SIGOPS Operating Systems Review*, 21(1):20–29, January 1987.
- [TB01] Marcelo Tallis and Robert Balzer. Document integrity through mediated interfaces. *DARPA Information Survivability Conference & Exposition II (DISCEX'01)*, pages 263–270, June 2001.
- [TBB88] Roberto Tamassia, Giuseppe Di Battista, and Carlo Batini. Automatic graph drawing and readability of diagrams. *IEEE Transactions on Systems, Man, and Cybernetics*, 18(1):61–79, January/February 1988.
- [TBB03] Mark Turner, David Budgen, and Pearl Brereton. Turning software into a service. *IEEE Computer*, 36(10):38–44, October 2003.
- [TBG01] Marcelo Tallis, Robert M. Balzer, and Neil M. Goldman. The briefing associate: Role for COTS applications in the semantic web. *International Semantic Web Working Symposium 2001 (SWWS'01)*, pages 463–775, July 2001.
- [TBG02] Marcelo Tallis, Robert M. Balzer, and Neil M. Goldman. The briefing associate: Easing authors into the semantic web. *IEEE Intelligent Systems*, 17(1):26–32, January/February 2002.
- [TCG⁺02] Tommi Tulasalo, Rune Carlsen, Andre Guirard, Pekka Hartikainen, Grant McCarthy, and Gustavo Pecky. *Domino Designer 6: A Developer's Handbook*. IBM Redbooks, December 2002.
- [TD99] Sander Tichelaar and Serge Demeyer. SNiFF+ talks to Rational Rose: Interoperability using a common exchange model. *SNiFF+ User's Conference*, January 1999. <http://www.iam.unibe.ch/~demeyer/Tich99m/>.
- [TD01] Scott Tilley and Mohan DeSouza. Spreading knowledge about gnutella: A case study in understanding net-centric applications. *9th IEEE International Workshop on Program Comprehension (IWPC'01)*, pages 189–198, May 2001.
- [TD04] Scott Tilley and Damiano Distante. On the adoption of an approach to reengineering web application transactions. *4th International Workshop on Adoption-Centric Software Engineering (ACSE'04)*, pages 13–20, May 2004.
- [TDD00a] Sander Tichelaar, Stephane Ducasse, and Serge Demeyer. FAMIX and XMI. *7th IEEE Working Conference on Reverse Engineering (WCRE'00)*, pages 296–298, November 2000.
- [TDD00b] Sander Tichelaar, Stephane Ducasse, and Serge Demeyer. FAMIX: Exchange experiences with CDIF and XMI. *Workshop on Standard Exchange Format (WoSEF)*, June 2000. <http://www.ics.uci.edu/~ses/wosef/papers/Tichelaar.ps>.
- [Tea02] CMMI Product Team. Capability maturity model integration (cmmi), version 1.1. Technical Report CMU/SEI-2002-TR-011, Software Engineering Institute, Carnegie Mellon University, March 2002. <http://www.sei.cmu.edu/pub/documents/02.reports/pdf/02tr011.pdf>.
- [TG01] Qiang Tu and Michael W. Godfrey. The build-time software architecture view. *17th IEEE International Conference on Software Maintenance (ICSM'01)*, pages 398–407, November 2001.

- [TGH⁺04] S. Tilley, J. Gedes, T. Hamilton, S. Huang, H. Muller, D. Smith, and K. Wong. On the business value and technical challenges of adopting Web services. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(1-2):31–50, January–April 2004.
- [TH01] Scott Tilley and Shihong Huang. Evaluating the reverse engineering capabilities of Web tools for understanding site content and structure: A case study. *23rd ACM/IEEE International Conference on Software Engineering (ICSE'01)*, pages 514–523, May 2001.
- [TH02] Scott Tilley and Shihong Huang. On selecting software visualization tools for program understanding in an industrial context. *10th IEEE International Workshop on Program Comprehension (IWPC'02)*, pages 285–288, June 2002.
- [TH06] Scott Tilley and Shihong Huang. On the “selling” of academic research to industry. *International Workshop on Software Technology Transfer in Software Engineering (TT'06)*, pages 41–42, May 2006.
- [Tho04] Andre Thomas. Erfahrungsbericht – methoden in der anwendungsbetreuung, September 2004. www.uni-koblenz.de/~ist/repro2004/beitraege/thomas.pdf.
- [THP03] Scott Tilley, Shihong Huang, and Tom Payne. On the challenges of adopting ROTS software. *3rd International Workshop on Adoption-Centric Software Engineering (ACSE'03)*, pages 3–6, May 2003.
- [Tic92] Walter F. Tichy. Programming-in-the-large: Past, present, and future. *14th ACM/IEEE International Conference on Software Engineering (ICSE'92)*, pages 362–367, May 1992.
- [Tic01] Sander Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, Universität Bern, December 2001.
- [Til94] Scott R. Tilley. Domain-retargetable reverse engineering II: Personalized user interfaces. *10th IEEE International Conference on Software Maintenance (ICSM'94)*, pages 336–342, September 1994.
- [Til95] Scott Robert Tilley. *Domain-Retargetable Reverse Engineering*. PhD thesis, Department of Computer Science, University of Victoria, 1995.
- [Til97] Scott Tilley. Discovering DISCOVER. Technical Report CMU/SEI-97-TR-012, Software Engineering Institute, Carnegie Mellon University, October 1997. <http://www.sei.cmu.edu/pub/documents/97.reports/pdf/97tr012.pdf>.
- [Til98a] Scott Tilley. A reverse-engineering environment framework. Technical Report CMU/SEI-98-TR-005, Software Engineering Institute, Carnegie Mellon University, April 1998. <http://www.sei.cmu.edu/pub/documents/98.reports/pdf/98tr005.pdf>.
- [Til98b] Scott R. Tilley. Coming attractions in program understanding ii: Highlights of 1997 and opportunities in 1998. Technical Report CMU/SEI-98-TR-001, Software Engineering Institute, Carnegie Mellon University, February 1998. <http://www.sei.cmu.edu/pub/documents/98.reports/pdf/98tr001.pdf>.
- [Til00] Scott Tilley. The canonical activities of reverse engineering. *Annals of Software Engineering*, 9:249–271, May 2000.
- [TJ03] Marco Torchiano and Letizia Jaccheri. Assessment of reusable COTS attributes. In H. Erdogmus and T. Weng, editors, *2nd International Conference on COTS-Based Software Systems (ICCBSS'03)*, volume 2580 of *Lecture Notes in Computer Science*, pages 219–228. Springer-Verlag, 2003.
- [TJSW02] Marco Torchiano, Letizia Jaccheri, Carl-Fredrik Sorensen, and Alf Inge Wang. COTS products characterization. *14th ACM/IEEE International Conference on Software Engineering and Knowledge Engineering (SEKE'02)*, pages 335–338, July 2002.

- [TLH97] Vu Tran, Dar-Biau Liu, and Brad Hummel. Component-based systems development: Challenges and lessons learned. *8th IEEE International Workshop on Incorporating Computer Aided Software Engineering*, pages 452–462, July 1997.
- [TM04] Marco Torchiano and Maurizio Morisio. Overlooked aspects of COTS-based development. *IEEE Software*, 21(2):88–93, March/April 2004.
- [TMR02a] Alexandru Telea, Alessandro Maccari, and Claudio Riva. An open toolkit for prototyping reverse engineering visualizations. *Symposium on Data Visualization 2002 (VISSYM'02)*, pages 241–249, May 2002.
- [TMR02b] Alexandru Telea, Alessandro Maccari, and Claudio Riva. An open visualization toolkit for reverse architecting. *10th IEEE International Workshop on Program Comprehension (IWPC'02)*, pages 3–10, June 2002.
- [TMSM02] M. Taylor, J. McWilliam, J. Sheehan, and A. Mulhane. Maintenance issues in the Web site development process. *Journal of Software Maintenance and Evolution: Research and Practice*, 14(2):109–122, March/April 2002.
- [TMWW93] Scott R. Tilley, Hausi A. Müller, Micheael J. Whitney, and Kenny Wong. Domain-retargetable reverse engineering. *9th IEEE Conference on Software Maintenance (CSM'93)*, pages 142–151, September 1993.
- [TN92] Ian Thomas and Brian A. Nejme. Definitions of tool integration for environments. *IEEE Software*, 9(2):29–35, March 1992.
- [TP03] Paolo Tonella and Alessandra Potrich. Reverse engineering of the interaction diagrams from C++ code. *19th IEEE International Conference on Software Maintenance (ICSM'03)*, pages 159–168, September 2003.
- [Tra01] Will Tracz. COTS myths and other lessons learned in component-based software development. In George T. Heineman and William T. Councill, editors, *Component-Based Software Engineering: Putting the Pieces Together*, chapter 6, pages 99–111. Addison-Wesley, 2001.
- [TS96] Scott R. Tilley and Dennis B. Smith. Coming attractions in program understanding. Technical Report CMU/SEI-96-TR-019, Software Engineering Institute, Carnegie Mellon University, December 1996. <http://www.sei.cmu.edu/pub/documents/96.reports/pdf/96.tr.019.pdf>.
- [TS97] Scott R. Tilley and Dennis B. Smith. On using the web as infrastructure for reengineering. *5th IEEE International Workshop on Program Comprehension (IWPC'97)*, pages 170–173, March 1997.
- [Tur00] Lori Turner. *Automating Microsoft Office 97 and Office 2000*. Microsoft Product Support Services White Paper, March 2000. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dno2kta/html/offaut.asp>.
- [TvH00] Vincent Traas and Jos van Hilleberg. The software component market on the internet: Current status and conditions for growth. *ACM SIGSOFT Software Engineering Notes*, 25(1):114–117, January 2000.
- [TW97] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems: Design and Implementation*. Prentice Hall, second edition, 1997.
- [TW98] Mark A. Toleman and Jim Welsh. Systematic evaluation of design choices for software development tools. *Software – Concepts & Tools*, 19(3):109–121, 1998.
- [TW00] Arthur Tateishi and Andrew Walenstein. Applying traditional unix tools during maintenance: An experience report. *7th IEEE Working Conference on Reverse Engineering (WCRE'00)*, pages 203–206, November 2000.
- [TWMS93] Scott R. Tilley, Michael J. Whitney, Hausi A. Müller, and Margaret-Anne D. Storey. Personalized information structures. *11th ACM International Conference on Systems Documentation (SIGDOC'93)*, pages 325–337, October 1993.

- [Ude94] Jon Udell. Visual Basic custom controls meet OLE. *Byte*, March 1994. <http://www.byte.com/art/9403/sec11/art1.htm>.
- [US04] Pauliina Ulkuniemi and Veikko Seppänen. COTS component acquisition in an emerging market. *IEEE Software*, 21(6):76–82, November/December 2004.
- [VBM05] Marian Vittek, Peter Borovansky, and Pierre-Etienne Moreau. A collection of C, C++, and Java code understanding and refactoring plugins. *ICSM 2005 Industrial & Tool Proceedings*, pages 61–64, September 2005.
- [VD97] Mark R. Vidger and John Dean. An architectural approach to building systems from COTS software components. *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'97)*, pages 131–143, November 1997.
- [VD98] M. R. Vidger and J. C. Dean. Building maintainable COTS based systems. *14th IEEE International Conference on Software Maintenance (ICSM'98)*, pages 132–138, November 1998.
- [vdBdJJK03] M. G. J. van den Brand, H. A. de Jong, P. Klint, and A. T. Kooiker. A language development environment for Eclipse. *2003 OOPSLA workshop on eclipse technology eXchange*, pages 55–59, October 2003.
- [vdBdJKO00] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Software—Practice and Experience*, 30(3):259–291, March 2000.
- [vdBHDJ⁺01] Mark van den Brand, Jan Heering, Hayco de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter Olivier, Jeroen Scheerder, Jurgen Vinju, Eelco Visser, and Joost Visser. The ASF+SDF meta-environment: a component-based language development environment. In R. Wilhelm, editor, *Compiler Construction 2001 (CC'01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.
- [vdBSV98] Mark van den Brand, Alex Sellink, and Chris Verhoef. Current parsing techniques in software renovation considered harmful. *6th IEEE International Workshop on Program Comprehension (IWPC'98)*, pages 108–117, June 1998.
- [vdBV05] Mark van den Brand and Jurgen Vinju. Generic language technology: Components for automated transformations. *Dagstuhl Seminar 05161, Schloss Dagstuhl*, April 2005. <http://www.dagstuhl.de/05161/Materials/>.
- [vdDHK⁺04] Arie van Deursen, Chrisine Hofmeister, Rainer Koschke, Leon Moonen, and Claudio Riva. Symphony: View-driven software architecture reconstruction. *4th IEEE/IFIP Conference on Software Architecture (WICSA'04)*, pages 122–132, June 2004.
- [vDK99] Arie van Deursen and Tobias Kuipers. Building documentation generators. *15th IEEE International Conference on Software Maintenance (ICSM'99)*, pages 40–49, August 1999.
- [vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.
- [vDM98] Arie van Deursen and Leon Moonen. Type inference for COBOL systems. *5th IEEE Working Conference on Reverse Engineering (WCRE'98)*, pages 220–230, October 1998.
- [vDM00] Arie van Deursen and Leon Moonen. Exploring legacy systems using types. *7th IEEE Working Conference on Reverse Engineering (WCRE'00)*, pages 32–41, November 2000.
- [vEM02] Eva van Emden and Leon Moonen. Java quality assurance by detecting code smells. *9th IEEE Working Conference on Reverse Engineering (WCRE'02)*, pages 97–106, October 2002.
- [Ves91] Iris Vessey. Cognitive fit: A theory-based analysis of the graphs versus tables literature. *Decision Sciences*, pages 219–240, 1991.

- [Ves97] Iris Vessey. Problems versus solutions: The role of the application domain in software. *7th Workshop on Empirical Studies of Programmers (ESP'97)*, pages 233–240, 1997.
- [Ves03] Thomas Vestdam. Elucidative programming in open integrated development environments for Java. *2nd International Conference on Principles and Practice of Programming in Java (PPPJ'03)*, pages 49–54, June 2003.
- [VG98] Iris Vessey and Robert Glass. Strong vs. weak approaches to systems development. *Communications of the ACM*, 41(4):99–102, April 1998.
- [vGBS01] Jilles van Gurp, Jan Bosch, and Mikael Svahnberg. On the notion of variability in software product lines. *2nd IEEE/IFIP Working Conference on Software Architecture (WICSA'01)*, pages 45–54, August 2001.
- [VGD96] Mark R. Vigder, W. Morven Gentleman, and John Dean. COTS software integration: State of the art. Technical Report NRC 39198, National Research Council Canada, January 1996. http://it-iti.nrc-cnrc.gc.ca/publications/nrc-39198_e.html.
- [Vid01] Mark Vidger. The evolution, maintenance and management of component-based systems. In George T. Heineman and William T. Councill, editors, *Component-Based Software Engineering: Putting the Pieces Together*, chapter 30, pages 527–539. Addison-Wesley, 2001.
- [Vin02] Steve Vinoski. Middleware "dark matter". *IEEE Internet Computing*, 6(5):92–95, September–October 2002.
- [Vit03] Padmal Vitharana. Risks and challenges of component-based software development. *Communications of the ACM*, 46(8):67–72, August 2003.
- [VMB03] Mark Vigder, Toby McClean, and Francis Bordeleau. Evaluating COTS based architectures. In H. Erdogmus and T. Weng, editors, *2nd International Conference on COTS-Based Software Systems (ICBSS'03)*, volume 2580 of *Lecture Notes in Computer Science*, pages 240–250. Springer-Verlag, 2003.
- [vMV93] A. von Mayrhauser and A. M. Vans. From program comprehension to tool requirements for an industrial environment. *2nd IEEE Workshop on Program Comprehension (WPC'93)*, pages 78–86, July 1993.
- [vMV95a] A. von Mayrhauser and A. M. Vans. Identification of dynamic comprehension processes during large scale maintenance. *IEEE Transactions on Software Engineering*, 22(6):424–437, June 1995.
- [vMV95b] Anneliese von Mayrhauser and A. Marie Vans. Program comprehension during software maintenance and evolution. *IEEE Computer*, 28(8):44–55, August 1995.
- [VN02] Thomas Vestdam and Kurt Normark. Aspects of internal program documentation—an elucidative perspective. *10th IEEE International Workshop on Program Comprehension (IWPC'02)*, pages 43–52, June 2002.
- [Voa98] Jeffrey M. Voas. The challenges of using COTS software in component-based development. *IEEE Computer*, 31(6):44–45, June 1998.
- [Voa99] Jeffrey M. Voas. Disposable information systems: The future of software maintenance? *Journal of Software Maintenance: Research and Practice*, 11(2):143–150, March/April 1999.
- [vOB02] Rob van Ommering and Jan Bosch. Widening the scope of software product lines—from variation to composition. In G. Chastek, editor, *SPLC2*, volume 2379 of *Lecture Notes in Computer Science*, pages 328–347. Springer-Verlag, 2002.
- [Vre03] K. Vredenburg. Building ease of use into the IBM user experience. *IBM Systems Journal*, 42(4):517–531, April 2003.
- [Wal98] Kurt C. Wallnau. A basis for COTS software evaluation: Foundations for the design of COTS-intensive systems. *SEI Interactive, Software Engineering Institute*, 1998. http://www.sei.cmu.edu/cbs/cbs_slides/ease98/index.htm.

- [Wal02] Andrew Walenstein. *Cognitive Support in Software Engineering Tools: A Distributed Cognition Environment*. PhD thesis, Simon Fraser University, May 2002.
- [Wal03a] Andrew Walenstein. Improving adoptability by preserving, leveraging, and adding cognitive support to existing tools and environments. *3rd International Workshop on Adoption-Centric Software Engineering (ACSE'03)*, pages 36–41, May 2003.
- [Wal03b] Andrew Walenstein. Observing and measuring cognitive support: Steps toward systematic tool evaluation and engineering. *11th IEEE International Workshop on Program Comprehension (IWPC'03)*, pages 185–194, May 2003.
- [Was90] Anthony I. Wasserman. Tool integration in software engineering environments. In F. Long, editor, *Software Engineering Environments: International Workshop on Environments*, volume 467 of *Lecture Notes in Computer Science*, pages 137–149. Springer-Verlag, 1990.
- [WBM95] Chales H. Wells, Russel Brand, and Lawrence Markosian. Customized tools for software quality assurance and reengineering. *2nd IEEE Working Conference on Reverse Engineering (WCRE'95)*, pages 71–77, July 1995.
- [WBM99] Paul Warren, Cornelia Boldyreff, and Malcom Munro. The evolution of websites. *7th IEEE International Workshop on Program Comprehension (IWPC'99)*, pages 178–185, May 1999.
- [WCK99] Steven Woods, S. Jeromy Carriere, and Rick Kazman. A semantic foundation for architectural reengineering and interchange. *15th IEEE International Conference on Software Maintenance (ICSM'99)*, pages 391–398, August 1999.
- [Weg89] Peter Wegner. Capital-intensive software technology. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability*, volume I: Concepts and Models. ACM Press, 1989.
- [Weg96] Peter Wegner. Interoperability. *ACM Computing Surveys*, 28(1):285–287, March 1996.
- [Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.
- [Wes04] David West. Looking for love (in all the wrong places). *ACM SIGPLAN Notices*, 39(12):57–63, December 2004.
- [WFS02] Elizabeth E. Wierba, Thomas A. Finholt, and Michelle P. Stevens. Challenges to collaborative tool adoption in a manufacturing engineering setting: A case study. *35th IEEE Hawaii International Conference on System Sciences (HICSS'02)*, pages 3594–3603, January 2002.
- [WG92] Niklaus Wirth and Jürg Gutknecht. *Project Oberon: The Design of an Operating System and Compiler*. Addison-Wesley, 1992.
- [WHG⁺03] T. Wang, A. Hassan, A. Guedem, W. Abdelmoez, K. Goseva-Popstojanova, and H. Ammar. Architectural level risk assessment tool based on UML specifications. *25th ACM/IEEE International Conference on Software Engineering (ICSE'03)*, pages 808–809, May 2003.
- [Wii92] Uffe Kock Wiil. Issues in the design of EHTS: A multiuser hypertext system for collaboration. *25th IEEE Hawaii International Conference on System Sciences (HICSS'92)*, ii:629–639, January 1992.
- [Wil01] David S. Wile. Supporting the DSL spectrum. *Journal of Computing and Information Technology*, 9(4):263–287, 2001.
- [Wil04] Gregory V. Wilson. Extensible programming for the 21st century. *ACM Queue*, 2(9):48–57, December/January 2004.
- [Wir86] Niklaus Wirth. *Compilerbau*. Teubner, fourth edition, 1986.
- [Wir95] Niklaus Wirth. A plea for lean software. *IEEE Computer*, 28(2):64–68, February 1995.

- [WKM02] Anke Weber, Holger M. Kienle, and Hausi A. Müller. Live documents with contextual, data-driven information components. *20th ACM International Conference on Documentation (SIGDOC'02)*, pages 236–247, October 2002.
- [WL05] Kevin F. White and Wayne G. Lutters. Insightful illusions: Requirements gathering for large-scale groupware systems. *ACM SIGGROUP Conference on Supporting Group Work (GROUP'05)*, pages 348–349, November 2005.
- [WMSL04] Xiaomin Wu, Adam Murray, Margaret-Anne Storey, and Rob Lintern. A reverse engineering approach to support software maintenance: Version control knowledge extraction. *11th IEEE Working Conference on Reverse Engineering (WCRE'04)*, pages 90–99, November 2004.
- [WOL⁺98] Steven Woods, Liam O'Brien, Tao Lin, Keith Gallagher, and Alex Quilici. An architecture for interoperable program understanding tools. *6th IEEE International Workshop on Program Comprehension (IWPC'98)*, pages 54–63, June 1998.
- [Won96] Kenny Wong. On inserting program understanding technology into the software change process. *4th IEEE Workshop on Program Comprehension (WPC'96)*, pages 90–99, March 1996.
- [Won98] Kenny Wong. *Rigi User's Manual: Version 5.4.4*. Department of Computer Science, University of Victoria, June 1998.
- [Won99] Kenny Wong. *The Reverse Engineering Notebook*. PhD thesis, Department of Computer Science, University of Victoria, 1999.
- [WS00] Jingwei Wu and Margaret-Anne D. Storey. A multi-perspective software visualization environment. *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'00)*, pages 15–24, October 2000.
- [WTMS95] Kenny Wong, Scott R. Tilley, Hausi A. Müller, and Margaret-Anne D. Storey. Structural redocumentation: A case study. *IEEE Software*, 12(1):46–54, January 1995.
- [WW01] Huaqing Wang and Chen Wang. Open source software adoption: A status report. *IEEE Software*, 18(2):90–95, March/April 2001.
- [WWB⁺03] Qin Wang, Wei Wang, Rhodes Brown, Karel Driesen, and Bruno Dufour. EVOlve: An open extensible software visualization framework. *ACM Symposium on Software Visualization (SoftVis'03)*, pages 37–38, June 2003.
- [WYM01] Anke Weber, Fang Yang, and Hausi Müller. Office XP in scenarios. *CASCON 2001 Workshop on Adoption-Centric Tool Development (ACTD'01)*, November 2001. <https://www-927.ibm.com/ibm/cas/toronto/publications/TR-74.182/37/anke-ACTD.pdf>.
- [Yan91] Hongji Yang. The supporting environment for a reverse engineering system—the maintainer's assistant. *Conference on Software Maintenance (CMS'91)*, pages 13–22, October 1991.
- [Yan03] Fang Yang. Using Excel and PowerPoint to build a reverse engineering tool. Master's thesis, Department of Computer Science, University of Victoria, 2003.
- [Yav04] Jonathan Yavner. Back-propagation of knowledge from syntax tree to C source code. *ACM SIGPLAN Notices*, 39(3):31–37, March 2004.
- [YBB99] Daniil Yakimovich, James M. Bieman, and Victor R. Basili. Software architecture classification for estimating the cost of COTS integration. *21st ACM/IEEE International Conference on Software Engineering (ICSE'99)*, pages 296–302, May 1999.
- [YBBP05] Ye Yang, Jesal Bhuta, Barry Boehm, and Daniel N. Port. Value-based processes for COTS-based applications. *IEEE Software*, 22(4):54–62, July/August 2005.
- [YDMA05] Yijun Yu, Homayoun Dayani-Fard, John Mylopoulos, and Periklis Andritsos. Reducing build time through precompilations for evolving large software. *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 59–68, September 2005.

- [YHC97] Alexander S. Yeh, David R. Harris, and Melissa P. Chase. Manipulating recovered software architecture views. *19th ACM/IEEE International Conference on Software Engineering (ICSE'97)*, pages 184–194, 1997.
- [Zay02] Iyad Zayour. *Reverse Engineering: A Cognitive Approach, a Case Study and a Tool*. PhD thesis, University of Ottawa, 2002.
- [ZC97] Marvin Zelkowitz and Barbara Cuthill. Application of an information technology model to software engineering environments. *Journal of Systems and Software*, 37(1):27–40, April 1997.
- [ZCK⁺03] Qin Zhu, Yu Chen, Piotr Kaminski, Anke Weber, Holger Kienle, and Hausi A. Müller. Leveraging Visio for adoption-centric reverse engineering tools. *10th IEEE Working Conference on Reverse Engineering (WCRE'03)*, pages 270–274, November 2003.
- [Zel96] Marvin V. Zelkowitz. Modeling software engineering environment capabilities. *Journal of Systems and Software*, 35(1):3–14, October 1996.
- [ZL00] Iyad Zayour and Timothy C. Lethbridge. A cognitive and user centric based approach for reverse engineering tool design. *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'00)*, November 2000.
- [ZL01] Iyad Zayour and Timothy C. Lethbridge. Adoption of reverse engineering tools: a cognitive perspective and methodology. *9th IEEE International Workshop on Program Comprehension (IWPC'01)*, pages 245–255, May 2001.
- [Zub97] John C. Zubeck. Implementing reuse with RAD tools' native objects. *IEEE Computer*, 30(10):60–65, October 1997.
- [ZW98] Marvin V. Zelkowitz and Dolores R. Wallace. Experimental models for validating technology. *IEEE Computer*, 31(5):23–31, May 1998.

Acronyms

4GL	fourth-generation language
ABAS	Attribute-Based Architectural Style
ACRE	Adoption-Centric Reverse Engineering
ACSE	Adoption-Centric Software Engineering
ANDF	Architecture Neutral Distribution Format (Open Software Foundation)
ASF	Algebraic Specification Formalism (ASF+SDF Meta-Environment)
ASIS	Ada Semantic Interface Specification
AST	abstract syntax tree
API	application programming interface
ATAM	Architecture Tradeoff Analysis Method (SEI)
AWT	Abstract Widget Toolkit (Java)
CARE	computer-aided reverse engineering
CASE	computer-aided software engineering
CBS	component-based system
CBSE	component-based software engineering
CDE	Common Desktop Environment (The Open Group)
CFG	control-flow graph
CMM	Capability Maturity Model (SEI)
CMMI	Capability Maturity Model Integration (SEI)
COM	Component Object Model (Microsoft)
CORBA	Common Object Request Broker Architecture (OMG)
COTS	commercial off-the-shelf
DCOM	Distributed Component Object Model (Microsoft)
DSVL	domain-specific visual language
DTD	Document Type Definition (W3C)
eTX	eclipse Technology Exchange
GCC	GNU Compiler Collection (Free Software Foundation)
GOTS	government off-the-shelf
GUI	graphical/graphics-based user interface
GXL	Graph eXchange Language
IDE	integrated development environment
I/O	input/output
IR	intermediate representation
JDBC	Java Database Connectivity
LOC	lines of code
MDA	Model Driven Architecture (OMG)
MOF	Meta-Object Facility (OMG)
NDI	nondevelopmental item
ODBC	Open Database Connectivity

OLE	Object Linking and Embedding (Microsoft)
OMG	Object Management Group
OTC	Object-Oriented Technology Center (IBM)
OSMM	Open Source Maturity Model
OSS	open source software
OTS	off-the-shelf
PCTE	Portable Common Tools Environment
PDG	program-dependency graph
POP	package-oriented programming
RAD	rapid application development
REEF	Reverse-Engineering Environment Framework
ROTS	research off-the-shelf
RPM	RPM Package Manager
RSF	Rigi Standard Format
RUP	Rational Unified Process
SAAM	Software Architecture Analysis Method (SEI)
SDF	Syntax Definition Formalism (ASF+SDF Meta-Environment)
SDL	Specification and Description Language
SEI	Software Engineering Institute
SOA	service-oriented architecture
SWT	Standard Widget Toolkit (Eclipse)
UI	user interface
UML	Unified Modeling Language (OMG)
W3C	The World Wide Web Consortium
WoSEF	Workshop on Standard Exchange Format
WSAD	Websphere Application Developer (IBM)
XML	Extensible Markup Language (W3C)
XMI	XML Metadata Interchange (OMG)
XP	Extreme Programming

Colophon

“You fling the book on the floor, you would hurl it out of the window, even out of the closed window, through the slats of the Venetian blinds; let them shred its incongruous quires, let sentences, words, morphemes, phonemes gush forth, beyond recomposition into discourse; through the panes, and if they are of unbreakable glass so much the better, hurl the book and reduce it to photons, undulatory vibrations, polarized spectra; through the wall, let the book crumble into molecules and atoms passing between atom and atom of the reinforced concrete, breaking up into electrons, neutrons, neutrinos, elementary particles more and more minute; through the telephone wires, let it be reduced to electronic impulses, into flow of information, shaken by redundancies and noises, and let it be degraded into a swirling entropy.”

– Italo Calvino [Cal81]

This dissertation has been typeset with \LaTeX on XEmacs and Emacs running GNU/Linux (various SUSE distributions). Drawings have been made with `xfig`.²⁸⁰ Screenshots have been taken with the GIMP.²⁸¹

²⁸⁰<http://www.xfig.org/>

²⁸¹<http://www.gimp.org/>

Creative Commons License

Each dissertation builds on the work of other researchers who have published their results. The free flow of information is a necessary prerequisite to scientific progress. Unfortunately, this flow is inhibited because many scientific information is in the hands of commercial publishers. As a result, such information can be difficult and costly to obtain.

This dissertation has greatly benefitted from the works of others. As a small token, I want to make my results freely available and easily accessible to the scientific community. Unfortunately, the University of Victoria's *Guidelines for the Preparation of Master's Theses and Doctoral Dissertations*²⁸² forces students to put a copyright notice in the title page. To work around the university's restriction, I make this dissertation also available under the following Creative Commons license:



This work is licensed under the Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

²⁸²http://registrar.uvic.ca/grad/documents/Thesis_Guidelines_Nov2005.pdf

University of Victoria Partial Copyright License

I hereby grant the right to lend my dissertation to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this dissertation for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this dissertation for financial gain by the University of Victoria shall not be allowed without my written permission.

Title of Dissertation:

Building Reverse Engineering Tools with
Software Components

Author: _____
Holger Kienle

Signed: _____