INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA

UMI®
800-521-0600

The Reverse Engineering Notebook

by Kenny Wong
M.Sc., University of Victoria, 1991
B.Sc., University of Victoria, 1989

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

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

We accept this dissertation as conforming to the required standard

Dr. H. Müller, Co-Supervisor (Department of Computer Science)

Dr. F. Ruskey, Co-Supervisor (Department of Computer Science)

Dr. N. Horspool, Departmental Member (Department of Computer Science)

Dr. W. Pfaffenberger, Outside Member (Department of Mathematics)

Dr. P. Sorenson, External Examiner (Department of Computing Science, University of Alberta)

Co-Supervisor: Dr. H. Müller

Co-Supervisor: Dr. F. Ruskey

Abstract

Software must evolve over time or it becomes useless. Much of software production today is involved not in creating wholly new code from scratch but in maintaining and building upon existing code. Much of this code resides in old legacy software systems.

Unfortunately, these systems are often poorly documented. Typically, they become more complex and difficult to understand over time. Thus, there is a need to better understand existing software systems. An approach toward this problem would be a first step toward easing changes and extending the continuous evolution of these systems.

This dissertation addresses the problem by enabling continuous software understanding. There should be a base of reverse engineering abstractions that are carried forward during evolution. The proposed approach seeks to redocument existing software structure, capture the analysis decisions made, and support personal, customizable, and live perspectives of the software in an online journal called the Reverse Engineering Notebook.

The premise that software reverse engineering be applied continuously through-

out the lifetime of the software has major tool design implications. Thus, tool integration, process, and adoption are key issues for the Notebook. In particular, data integration requirements, control integration via pervasive scripting, presentation integration through the management of views, user roles, methodology, end user needs, and goal-directed framework for the Notebook are described.

A major theme of the dissertation is learning from the successes and failures of studies involving tool integration and reverse engineering technologies. Case studies and user experiments helped to evaluate various aspects of the Notebook approach and provide feedback into software understanding tool requirements.

Examiners:

Dr. H. Müller, Co-Supervisor (Department of Computer Science)

Dr. F. Ruskey, Co-Supervisor (Department of Computer Science)

Dr. N. Horspool, Departmental Member (Department of Computer Science)

Dr. W. Pfaffenberger, Outside Member (Department of Mathematics)

Dr. P. Sorenson, External Examiner (Department of Computing Science, University of Alberta)

Contents

List of Figures

List of Tables

Acknowledgments

Software engineering research does not occur in isolation, and this thesis work is no exception.

I am especially grateful to my advisor, Hausi Müller, for his support, guidance, and patience throughout this endeavor. His leadership and vision has been truly inspiring. Thanks for fostering such an enriching work and play environment.

As a member of the Rigi group at UVic, I have had the privilege to work with several people over the past years. For their encouragement, energy, friendship, insightful thoughts, witty remarks, doses of reality, implementation help, and much more, I owe many thanks to Peggy Storey, Scott Tilley, Mike Whitney, Jim McDaniel, Brian Corrie, and Jim Uhl.

Words cannot say how deeply I am indebted to my parents and my sister for their caring support and stable influence throughout my life. For lending kind ears and just being there, I owe them much gratitude.

Finally, I acknowledge the support of the University of Victoria, the Institute for Robotics and Intelligent Systems Network of Centres of Excellence (IRIS NCE), and the Natural Sciences and Engineering Research Council (NSERC).

For my parents.

Chapter 1

Introduction

I know engineers, they love to change things.
—Dr. McCoy, Star Trek

1.1 Problem

Software must evolve over time or it becomes useless [67]. Much of software production today is involved not in creating wholly new code from scratch but in maintaining and building upon existing code. Businesses need to understand and leverage their knowledge assets to exploit new opportunities such as electronic commerce. Much of this business knowledge is embedded in old legacy software systems.

Unfortunately, these systems are often poorly documented. Typically, they become more complex and difficult to understand over time. Complexity is a natural phenomenon for any large software system that undergoes change to meet current user needs. It is not that the software was not "done right" and should be scrapped in favor of a completely new system or a new development method. There is no guarantee that a rewritten system would not suffer the same problem. Thus, there

is a need to better understand existing software systems. This is the fundamental problem. An approach toward this problem would be a first step toward easing changes and extending the continuous evolution of these systems.

1.2 Approach

This dissertation addresses the problem by enabling incremental, continuous software understanding. For understanding an evolving software system, there is a gap between what is known, useful, available information and what needs to be known. In the long term, as the system evolves and people leave and documents decay, this gap grows. At some point, the size of this understanding gap may become difficult to span using typical on-demand code browsing techniques, thus hampering further changes. The maintainers may then need to undertake significant reverse engineering activities to rejuvenate and extend the useful lifetime of the system. These activities are likely more costly and more apt to fail than had the gap been better managed. By better managing the understanding gap, future understanding is eased. That is, software understanding should be a smoother, continuous process.

Just as most software development is not done from scratch, software understanding should not be done from scratch. There should be a base of structural software abstractions that are carried forward during evolution. The decisions made in originally deriving these abstractions should not be discarded during the forward engineering or development process. These abstractions should coevolve with the code base. Unfortunately, many legacy systems have missing or inconsistent design documentation. It is necessary to reconstruct the base of structural abstractions. Consequently, the proposed approach centers on software reverse engineering and redocumentation techniques. The approach seeks to redocument existing software structure, capture the decisions made, and support personal, customizable, and live perspectives of the software in an online journal called the Reverse Engineering Notebook.

1.3 Strategy

Major themes of this dissertation and goals of the Notebook include:

- enabling continuous software understanding;
- enhancing compatibility with existing tools, users, and processes by emphasizing integration, process, and adoption issues; and
- learning from the successes and failures of studies involving tool integration and reverse engineering technologies.

By observing existing solutions for software understanding and their interplay with existing tools, users, and processes, the dissertation produces fundamental requirements for the next generation of solutions. The distilled requirements serve as important design criteria for the Notebook. A candidate infrastructure and process for the Notebook is derived that unifies the successful aspects of current approaches. As an advance, the Notebook is aimed toward tackling the continuous understanding problem. For example, the need to make analyses of software understanding abstractions reusable is addressed in the Notebook infrastructure, which uses pervasive scripting to help codify analysis methods and record analysis decisions. The need to make the Notebook usable throughout evolution is addressed in a major focus on adoption issues. The premise that software reverse engineering be applied continuously throughout the lifetime of the software has major tool design implications. Thus, tool integration, process, and adoption are key issues for the Notebook.

1.4 Validation

Individual aspects of the Notebook have been applied or evaluated in several research projects in which I have been directly involved. Lessons learned and limi-

tations discovered in those projects served to improve the understanding of Notebook requirements, infrastructure, process, and adoption issues.

The SQL/DS project [153] evaluated a methodology called structural redocumentation and tested the capabilities of scripting to increase automation yet allow analysts to remain in control of reverse engineering analyses. The RevEngE project [86, 150] assessed the ability of tool integration technologies (a software repository and a message bus) to leverage the complementary capabilities of reverse engineering tools. In particular, the results drove the Notebook preference of simpler, lightweight integration technologies for data and control integration. The Software Bookshelf project [39] evaluated the usefulness of applying standard web technologies for delivering comprehensive software documentation. Specifically, the Notebook capability to present script-enacted, constructive views of software structure was tested. Controlled user experiments [124, 125] tested the effectiveness and usability of reverse engineering tools for various program understanding tasks.

A key theme of this dissertation is to recognize the importance of being compatible with existing tools, users, and processes. An appreciation of this critical need of practitioners is only possible by working on industrially oriented projects with real, legacy software systems. The SQL/DS, RevEngE, and Software Bookshelf projects all involved close collaboration with developers at IBM Toronto to help evaluate what approaches worked and what did not.

1.5 Outline of dissertation

This dissertation is organized as follows. Chapter 2 distills and identifies requirements of tool support for software understanding. The requirements characterize the fundamental needs to be addressed in the design of reverse engineering tools, such as the Notebook. Chapter 3 describes the Notebook concept and approach for continuous software understanding. Aspects of the Notebook infrastructure are also discussed, including data integration requirements, control integration

via pervasive scripting, and presentation integration through the management of views. Chapter 4 covers process aspects of the Notebook approach, including user roles and a methodology. Chapter 5 discusses adoption issues addressed by the Notebook. In addition, a goal-directed framework is detailed for the insertion and placement of the Notebook approach into a software change process. Chapter 6 summarizes lessons learned in various studies that have validated particular concepts of the Notebook infrastructure and process. Chapter 7 describes other related work and relevant technologies. Chapter 8 outlines several specific recommendations for reverse engineering tool research.

Chapter 2

Requirements

> The painter constructs, the photographer discloses.
> —Freeman Patterson

This chapter identifies requirements for software understanding in general and for reverse engineering tools in particular. These high-level requirements help to characterize the fundamental subproblems and issues that need to be effectively addressed in building these tools. These basic requirements are refined by experience in case studies of using reverse engineering tools [86, 150, 153] and through observations of users trying to understand software [125]. In the following discussion, the statement of a requirement follows the explanatory text.

2.1 Legacy systems

A large part of software engineering effort is involved not in producing code from scratch, but rather the acquisition, enhancement, and integration of existing code. That is, the focus in software engineering is shifting to long-term software *evolution*.

For businesses, much of their applications and data reside in large, *legacy* systems

[1]. "A legacy system is an application of value inherited from the past" [142]. These systems are typically mature and mission-critical. To exploit new business opportunities such as electronic commerce effectively, companies need to leverage these systems. Unfortunately, these systems are highly complex, poorly documented, and difficult to change.

Since the expense in the existing software is considerable, it is not always possible to scrap it and start over. Even starting over with new technologies is no guarantee that these new systems would not become the legacy systems of tomorrow. The environment around a software system naturally changes over time. Business processes adapt to the system; the system adapts to new business needs. Thus, the continued evolution of software is critical for keeping up with changing needs [67].

The evolution process, however, is complicated because of changing platforms and environments, languages, tools, methodologies, education, hardware, staff, target audience, and competitive pressures. Hence, one research goal is to support long-term software evolution in an environment of increasing complexity and diversity.

REQUIREMENT 1
Handle the scale, complexity, and diversity of large software systems.

2.2 Program understanding

Program understanding is a critical part of software evolution and is the first step in making sense of software assets. Indeed, effective understanding is necessary for properly finding and fixing defects, factoring and optimizing code, porting to different platforms, exploiting reusable components, integrating old and new technologies, and adding new features. Moreover, this understanding must be communicated among the staff and, because of staff turnover, be captured to train new employees.

The shift from new development to the maintenance and evolution of software

does not mean that structured and object-oriented analysis methodologies are not useful. The understanding needed to model the real world and build a new software system shares much with the understanding needed to model an existing system and its surrounding environment. For example, in both cases, hierarchies are used to form abstractions and reduce complexity. Object-oriented structure models might be useful when reengineering an existing system to be more object-oriented. Software engineering principles and patterns used to develop software are equally helpful to understand existing software architecture.

REQUIREMENT 2
Recognize that design abstractions are useful not only for the initial design but also as a way to convey high-level understanding during evolution.

REQUIREMENT 3
Capture both informal and formal information.

Different design criteria exist for decomposing code into modules, such as information hiding [93], separation of concerns, coupling and cohesion [84], and closure [76]. Because of software changes, the software architecture degrades in that the code no longer satisfies the original design criteria. This is especially true for legacy software systems, which are typically poorly structured. Such structures might start following organizational lines and ownership relationships, rather than established software engineering principles [17]. Sprawling and interleaving patchworks of code are used to implement required features. Maintenance becomes a write-only affair. Somebody still has to understand these high entropy systems. Thus, program understanding is an important part of software *preservation*.

Software understanding occurs on several fronts. Maintainers must know the structure and behavior of the code. This understanding involves architectural structures, resource flows, imports and exports, the individual behavior of software objects, and the emergent system-wide behavior of interactions among those objects [65]. Also, maintainers need external problem-domain and computing-domain expertise to put this information into context. Problem-domain expertise includes knowledge in the application area (such as finance, science, and com-

puting), historical knowledge behind the software, policies, design rationale, and intent. Computing-domain expertise includes knowledge about algorithms, data structures, programming interfaces, software architectures, engineering processes, and quality characteristics.

REQUIREMENT 4
Exploit expertise and capture concepts in both the problem and computing domains.

REQUIREMENT 5
Capture information during development to make future software understanding potentially easier.

2.3 Reverse engineering

Reverse engineering is an approach to understanding software structure, given the source code. This process identifies software building blocks, extracts structural dependencies, produces higher-level abstractions, and presents pertinent summaries. Building blocks include procedures, modules, types, classes, and global variables. Structural dependencies include client/supplier, subtyping, control flow, and, for object-oriented systems, instantiation (instance-of), inheritance (is-a), and composition (part-of). Maintainers can search for specific software artifacts, examine their properties, and follow their neighboring dependencies.

One goal of reverse engineering is to produce architectural views of the large-scale structure of the software. Higher-level views manage the complexity of the lowest level. One classical use of these views is to redocument an existing software system whose documentation is lost or lacking [26]. Maintainers often resort to reading the source code to manually build higher-level mental models—a tedious and errorprone process. Such a process would benefit from specialized tools and rationalization.

Reverse engineering helps by providing computer assistance, often using compiler technologies (lexical, syntactic, and semantic analysis). Static (semantic) analysis

strengthens the study by inferring relationships that may not be obvious from the syntax of the code, without running the program. For example, static analysis can check types, identify unreachable code, aliasing, uninitialized variables, and common subexpressions.

The software structure has an enormous impact on understanding and maintenance. Modern programming languages emphasize structure. For example, object-oriented languages like C++ ease maintenance by using classes and objects to specify interface contracts, promote encapsulation and localization of changes, and provide support for unit testing [100]. Inheritance hierarchies are used to share and reuse specifications and implementations (somewhat at the expense of locality). For C++, one distinguishing aspect is that these high-level programming constructs are built into the syntax of the language and are statically analyzable from the source code. This differs for earlier languages like COBOL that require naming and numbering conventions to build structural abstractions. The philosophy is that solutions and concepts can be "expressed directly and concisely" [126].

Human understanding is particularly important since incorrect and undisciplined use of these constructs can lead to confusing and difficult-to-maintain software. A taxonomy of defects in object-oriented software isolated nine bug types that can be introduced during the software lifecycle [100]. Inadequate knowledge and poor understanding played a major role in all bug types.

REQUIREMENT 6
Produce useful summaries of software structure while highlighting anomalies and filtering irrelevant details.

2.4 User needs

With any analysis process, one important need is presenting the right kind of information, in the right amount, at the right level of abstraction. There are different needs among customers, programmers, designers, and managers. Different people also have different ways of thinking. They may use a variety of program

comprehension strategies [145]. Some strategies proceed in a top-down fashion, starting with the main functions and proceeding to the specifics [16]. Other strategies proceed bottom-up, starting with the primitives and figuring how they are coordinated [119]. Also, some maintainers prefer to be systematic, examining the program in great detail before any change is made [69]. Systematic analysis, however, is not feasible for very large systems with millions of lines of code. Other maintainers are more opportunistic, working with subsets and discovering only what is needed to get the desired task done.

In practice, with experience, a mixed approach of understanding strategies is most often used. Thus, it is necessary to present the information in a way people actually think (not how they should think) [144]. Tools for software understanding must not impose a rigid comprehension strategy [122] nor some awkward process [151]. Tools should not be missing important capabilities, such as source code searching [125].

Tools should accommodate the creativity of users [57]. Important operations of the analysis tool should support spontaneity, immediacy, exploration, interaction, and trial-and-error. Analysts should be free to explore, work in whatever order they like, change their minds, and undo actions safely. A detailed analysis may need to be rolled back to a prior stage. This form of reverse engineering would better support the just-in-time comprehension needs of software maintainers [19].

REQUIREMENT 7
Support different users, their needs, and their preferred comprehension strategies, development tools, and engineering processes.

REQUIREMENT 8
Provide the right abstractions, as appropriate for the goals, tasks, and roles of the users.

REQUIREMENT 9
Exploit, integrate, and deliver diverse software analysis technologies.

A software system is a moving target that cannot easily be frozen while a massive reverse engineering effort takes place. Analysis tools need to be incremental and

work on partial systems [11]. Incrementality requires that analysis information can be refined by a deeper analysis or by considering additional units of code without destroying or redoing the earlier analysis information.

REQUIREMENT 10
Support the incremental analysis of software.

There is a synergy between reverse and forward engineering. Reverse engineering is performed in the context of domain models, user scenarios, and non-functional requirements. Forward engineering is driven by the current design architecture and functional requirements. The bridge that closes the loop involves architecture and decisions (rationale, intent, history). Certain analysis information such as architectural abstractions should be updated and evolved as the software system itself evolves. That is, an evolvable base of understanding is needed to ease future software changes. In addition, traceability is needed to relate requirements, design, and implementation artifacts [102]. For example, a developer may want to determine the rationale of a software component by tracing to find what requirement(s) it satisfies.

REQUIREMENT 11
Support the continuous understanding of software.

By applying reverse engineering techniques incrementally and continuously, we maintain continuous understanding for improved software change. By essentially tightening the loop between forward engineering and reverse engineering, we need to produce analysis tools that can be used frequently during the evolution of the software. Thus, there is a need to address the technology transfer or innovation diffusion barriers to adopting reverse engineering tools.

In general, several perceptual factors contribute to the rate of technology adoption, including relative advantage, compatibility, complexity, trialability, and visibility of results [111]. Software tools have to solve a real, immediate problem to convince programmers to use them [57]. Advanced technology and new paradigms with long term benefits for the company are not as convincing. Other problems are that

software tools involve too much training, do not fit with existing practices, or do not work on real-world, industrial software, due to large multi-million line bodies of source code and proprietary languages. Issues such as ease-of-use, scalability, and flexibility are important. Total lifecycle costs [52] and management support [56] are also critical.

REQUIREMENT 12
Address the practical issues underlying reverse engineering tool adoption.

2.5 Cognitive issues

People seem to understand the workings of things by recognizing patterns, making intuitive leaps, validating and refuting hypotheses according to observations, and making adjustments to mental models as needed. For example, programmers try to identify familiar fragments of code based on conventions, expectations, and previous experience. The understanding process is very individualistic and full of trial and error; not everyone builds the same mental model or builds it in the same way or uses the same interpretations. (In a sense, the structure of a software system is observer-created.)

REQUIREMENT 13
Represent multiple simultaneous software architectures and perspectives.

We manage complexity by categorizing facts and building hierarchical taxonomies [45]. Understanding is eased when new concepts are clarified through an existing mental framework. The creative organization of information creates new information for understanding [108]. Furthermore, alternative representations and arrangements convey information with different emphasis. Structure is a way for organizing the seemingly arbitrary rules and principles in these fields. Forming comparisons and recognizing sameness brings about generalizations and possibly interesting associations.

Conversely, people forget, get tired, and have problems with keeping data accurate, consistent, and up-to-date. Decisions can also be delayed or erroneous because of ambiguity, the lack of solid facts, a flood of unimportant detail, and incorrect information. Mental models can be difficult to throw away; we keep elaborating the model rather than admitting that the model is unsuitable [129]. Mistaken interpretations close ourselves to possible answers [2]. Judgements are biased by first impressions, recent events, and preconceived notions.

Ultimately, people are far more complex than the above descriptions indicate [129]. Nevertheless, this naïve view of human strengths and weaknesses still suggests a number of desirable properties for tools to aid understanding. Important facets of the analysis should support categorization. Let analysts organize analyses, form subsystems of software components, classify events, arrange views, and manage informal information. These categorizations can record patterns in the problem domain. Important areas of the analysis should support association. Let analysts make relationships between domains, such as dynamic information, static structure, and problem-domain knowledge.

REQUIREMENT 14

Support the organization of abstractions over the software to manage the complexity of understanding it (including hierarchy, generalization, categorization, aggregation, and association).

The strengths and weaknesses of humans and computers complement each other well. The most successful tools work semi-automatically and interactively in an active partnership with the user. People learn better by being actively engaged in the task [4]. Continuous feedback and quick responses are critical. The tool could work in a multi-threaded manner to allow computations to proceed while the user continues interacting, possibly stopping ongoing computations in midstream [109]. Analysts define the goals, guide the tool, and form new questions to advance the analysis. The user should be in control [4]. Computers provide real-time visualizations, ensure consistent views, recall details, and make suggestions.

REQUIREMENT 15

Provide interactive, consistent, and integrated views, with the user in control.

2.6 Process

Program understanding activities can be somewhat chaotic, with analysts making guesses, questions, and actions using their own experiences, heuristics, and judgment. Analysis activities need to become more mature and systematic, but this is difficult to achieve. These difficulties may stem from the problem that heuristics are often very domain-specific, whereas mature and systematic techniques tend to be quite general.

Another approach may be to develop a framework in which particular program understanding actions can be selected to answer specific questions about a software change. These actions should be repeatable without being overly restrictive. In particular, an action may need to be applied to another baseline of code (to support continuous understanding). The examination of past and partial analyses may be useful toward capturing and codifying experience for useful actions. Moreover, the process of reverse engineering should become more mature and rational by taking steps to make it more repeatable, defined, managed, and optimized [53, 54].

REQUIREMENT 16
Rationalize the reverse engineering process.

Trial and error in program understanding may produce a flood of generated information because of different data inputs, analysts, versions of tools, preference settings, or chosen analyses. An organizational structure is needed to allow analysis results to be rederived on either the original inputs or new ones. For example, a sense of repeatability is important in confirming user experiments in reverse engineering or in conducting consistent experimental sessions. Repeatability is one step toward increasing the maturity of the reverse engineering process.

REQUIREMENT 17
Organize the diverse array of information artifacts involved in software understanding.

2.7 Documentation

Maintainers need documentation, yet the rush to meet deadlines often means that the developers focus more on the code and the documentation is put aside, sometimes indefinitely. As such, maintainers are often faced with inconsistent, out-of-date, and at-worst, missing, incorrect, or downright unhelpful documentation. An important need, then, is to record incrementally the knowledge gained from the analyses into a repository for the benefit of other maintainers [11]. The repository should contain interlinked information [40, 28, 42] and support program structures, schema evolution, and millions of objects. However, such a repository is likely to degrade with time because maintainers might change the software without updating the documentation or enter redundant or conflicting information. Links between the source code and the repository are needed to ensure consistency, support traceability, and ease code review. For example, whenever some fragment of code is changed, the effected documentation in the repository is highlighted and perhaps even automatically regenerated.

Hypertext is a way of organizing text that uses links to associate pieces of text [24]. For example, a special term in a chunk of text may be linked to its definition. Hypermedia extends this notion to include other media, such as pictures, movies, and voice. The World Wide Web has popularized the notion of navigating a hypermedia network by following links and retrieving content at those links. Hypertext and web technologies can be exploited for representing software structures, dependencies, and documentation [39]. The major issues are in determining what nodes and links represent and look like, avoiding cognitive overload and too much choice, providing structure and cues, supporting tours, and maintaining the organization and associations without intensive labor.

REQUIREMENT 18
Provide consistent, continually updated, hypermedia software documentation.

REQUIREMENT 19
Address usability and cognitive overload problems of interlinked information.

2.8 Human-computer interaction

Pictures play an important role for understanding by exploiting the sense of vision [12] and making concrete the abstract. The ability to present very large numbers of objects effectively is critical; limited screen space and limited human cognitive abilities [78] are major concerns. Visual information and spatial notions such as location, orientation, proximity, area, and symmetry are important for easing pattern recognition. Effective use of color and texture can convey common themes, emphasize clusters of objects, and depict natural categories [73].

Still, the importance of words must not be overlooked; there is a balance between pictures and words. Visual representations of software are not necessarily superior to textual representations [97]. With a graph visualization, there may be no clue where to start inspecting it, whereas text can always be read linearly. Symmetry and layout may provide misleading information about content. A graph allows the eye to wander freely and casually to obtain an overall, visual impression of the software structure. Expert programmers, however, are typically deliberate, purposeful readers with specific goals and hypotheses in mind, not art viewers [97]. Some of these questions may be best answered in a textual form, such as a compact table [23] or a nicely formatted source code listing.

The benefit of language is the quality of abstraction it provides [63]. Clusters and layouts may be unlabeled and essentially non-verbal, but still have meaning and substantial semantics behind them. To reduce misinterpretation, the meaning and reason for being must be evident and codified through language, such as a script.

REQUIREMENT 20
Integrate graphical and textual software views, where effective and appropriate.

There are many kinds of diagrams [70]. Time lines depict histories of events. Outlines illustrate tree hierarchies. Graphs and maps show associations between and relationships among objects. Cross-sections show various slices of a hierarchical structure. Tables can show values of attributes of objects in various user-specified sorting orders. Many diagrams can be fairly abstract; the relationship between

the diagram and the data must be well-defined, evident, and explained (especially for newcomers). Furthermore, the data representation, being the basis for the diagrams, must be formal and sound. For program understanding, the focus is more on flexible understanding and less on flashy graphics. Program understanding must support analysts in exploring and making sense of software and forming new insights, *beyond* showing material they already know [77].

REQUIREMENT 21

Answer questions that the asker does not know about the software, rather than produce documentation for documentation's sake.

2.9 Visualization

For maintenance and program understanding, general and flexible visualizations are needed. These visualizations must also be robust since the subject software may have defects. The VIPS system [118] communicates with the conventional UNIX `dbx` debugger to animate dynamic data structures like linked lists. GraphTrace [65] provides dynamic structural and behavioral graph displays for an object-oriented language. One structural display highlights classes in the inheritance hierarchy as methods are invoked. One behavioral display presents the tree-like method invocation graph. PIGS [98] animates Nassi-Shneiderman diagrams, a form of structured flow chart. PUNDIT [89] also provides animated control flow and function call graphs. PROVIDE [80] allows its depictions to be adjusted on-the-fly, during execution of the monitored program. The result is that motion and animation are effective methods for conveying the temporal relationships of running software.

Static views are also needed for presenting non-animated information, such as static software structures, source code, data formats, informal text. Pecan [106] provides numerous static and dynamic views based on the abstract syntax tree of the subject software, including expression trees, symbol tables, and control flow graphs. FIELD, the successor to Pecan, coordinates these views and tools through an annotation mechanism supported by a message bus [107]. Such a mechanism

allows objects of interest to be tagged, linked to other information, and have scripts attached. The HY+ visualization system [75] supports a colorful visual language for querying higraph structures and relationships.

Some visualization techniques, such as graphical fish-eye views [115] and tree maps [59, 58, 141], are effective at conveying large hierarchies of objects. The Xerox PARC Information Visualizer [109] presents hierarchical information with three-dimensional cone trees and linear information with perspective walls. All these techniques support zooming in on details without losing the context. This is necessary for browsing and documenting complex software structures [122].

Static views can also present summaries and statistics about dynamic behavior. Such views may be more useful than fleeting animations. Seesoft [34] represents files of source code with rectangular strips and uses colored markings to show statement frequencies; hot spots are colored in shades of red. Traceview provides several Gantt charts to summarize trace information [72]. The result is the importance in views of stable states and reference points that analysts can fixate on while understanding dynamic information.

REQUIREMENT 22
Integrate static and dynamic views of software.

2.10 Evaluation

There has been a substantial amount of human-computer interaction research. This is bringing, for example, sophisticated software architectures for interactive applications, seamless document-based presentation integration, semi-autonomous agents and critics, and collaborative kits of task-specific components. Direct manipulation user interfaces exploit our physical motor sense and perceptive visual sense to complement our limited recall memory. Consistent and modeless operation allow users to learn the software quickly. For program understanding tools, careful design and usability testing of the user interface is important [155]. Despite this importance, not enough experiments have been done to evaluate the usage

and effectiveness of software tools [44, 146, 132, 156].

REQUIREMENT 23
Evaluate the effectiveness of reverse engineering tools and techniques through empirical studies.

2.11 Chapter summary

This chapter distills many important requirements for software understanding tool support. It is a significant research challenge to address all these requirements. The next chapter describes the approach taken by the Notebook.

Chapter 3

Notebook Infrastructure

> You don't invent nor reveal the answer, you remember.
> —John Mylopoulos

This chapter describes an approach for addressing many of the requirements outlined in the previous chapter. It is difficult to discover an infrastructure to integrate all the requirements, but it is highly desirable. To date, no tool or environment has achieved it. The Notebook infrastructure is a step in that direction.

The supporting technologies or *infrastructure* for the Notebook approach are discussed. The infrastructure serves as a foundation for enabling the approach. A key idea is to use integration technologies for a more extensible, flexible infrastructure that can co-exist with existing tools.

3.1 Concept

The vision of the dissertation is to enable continuous understanding during software evolution, as outlined in Chapters 1 and 2. An integrated design strategy is desirable to form a flexible foundation or vehicle for continuous understand-

ing. This dissertation proposes an unified approach called the Reverse Engineering Notebook, refined by lessons learned in developing and applying integrated reverse engineering environments in industrial settings [86, 150, 39]. The Notebook forms a personalized information space [136] for producing, managing, and delivering reverse engineering analyses of software structure. Also, the Notebook provides an *active* document that contains both data and scripts to generate and control its visual presentation [92].

Besides infrastructure issues, there are issues of process or usage. In brief, the intended use of the Notebook is to be *continuous throughout software evolution*, to capture structural abstractions to make future understanding easier. Reverse engineering is proposed as not only a process of making and presenting discoveries about software, but also one of recording key decisions during the process (for reenactment on other baselines of code). By recording and codifying certain reverse engineering tasks in scripts and by ensuring analyses are consistent and repeatable, the Notebook concept begins to rationalize the process of reverse engineering. The idea is similar to maintaining a scientific lab journal. A Notebook can serve as a platform for reverse engineering research and education, aiding in conducting and documenting experimental case studies.

3.2 Integration

One goal is to have a versatile but compatible infrastructure to support continuous understanding. To exploit and deliver diverse reverse engineering techniques flexibly and practically, the Notebook infrastructure employs two complementary approaches: an integrated toolset and the notion of a reusable component. That is, a Notebook could serve as an extensible and customizable architecture for a reverse engineering toolset and it could serve as a single, reusable component in another environment.

The Notebook theme of compatibility is especially relevant. Lightweight integration technologies are preferred in the Notebook infrastructure. From experience

in the RevEngE project, there are significant difficulties in using and maintaining advanced integration technologies. Also, it must be possible to move the Notebook to the development platform, process, and people as quickly and simply as possible. This is a generalized notion of portability.

Tool integration is necessary to combine diverse techniques effectively to meet software understanding needs. In general, a wide variety of tool capabilities and integration mechanisms are available today. It is up to the infrastructure builder to survey, evaluate, and combine these facilities to provide a layer of useful general services. These services then need to be adapted and made compatible for specific projects, processes, people, etc. Pragmatically, software integration is more a process of investigation and evaluation than a set of ready-made technologies.

Tool integration involves at least three dimensions: data, control, and presentation [148, 21, 130]. Data integration involves the sharing of information among tools. Control integration involves the coordination of tools to meet a goal. Presentation integration involves concerns of user interface consistency.

As main infrastructure contributions, the following sections

- explore data modeling and interchange issues,
- explore pervasive scripting issues, and
- outline strategies for managing presented views.

3.3 Data integration

To advance the discipline of software engineering, there need to be standard data models and interchange formats for information about software. Without such standards, there would be unnecessary reinvention of basic capabilities and a limitation in the ability to compare techniques and share results. One particular need is to form a foundation to share information about a software system among reverse engineering tools. That is, parsers should emit and analyzers should accept

an agreed data model represented by some agreed interchange format. Too much effort is often expended in writing parsers for specialized program representations and not enough in analyzing code, producing abstractions, and sharing discoveries. The data standards could be implemented by a software repository or program database, a common component of integrated reverse engineering toolsets.

Having a common format also allows bodies of potentially proprietary software to be parsed and saved as benchmarks for reverse engineering research. An advantage arises if the format is easier to process than the source language(s) of the software.

Establishing universal standards for software representation is difficult. Flexibility is critical. There are many potential uses to consider, from compilation and optimization to software understanding. Interested and affected parties have their own favored representations.

Constructing an open data standard is beyond the scope of the dissertation. Nevertheless, a discussion of pertinent issues and requirements is useful. Based on experience with developing and using software repositories, desirable aspects of a data model for representing software are described. The discussion uses ideas from conceptual modeling [15] more so than the low-level notions of file and data structures. Conceptual modeling has been applied to reverse engineering as a way to model concepts in the programming and application domains [133]. Here, the specific use is representing information about software for continuous software understanding.

3.3.1 Desirable aspects

A conceptual model about software must address three important levels of information: the fact, the schema, and the metaschema. Each level may be individually expressed using some physical syntax or encoding and have associated semantics (perhaps implemented procedurally and/or defined formally). The fact level concerns actual content, such as the text of the main function definition or the rela-

tionship that function `main` calls the function `printf`. The schema level describes the structure of the data, specifying what can be expressed as valid facts. This level (also known as a domain model [85]) consists of concepts, relations, and properties relevant in some context, serving as important common ground for understanding and communication among people. The metaschema level, in turn, describes valid schemas.

A formal metaschema adds flexibility whereby a schema can be instantiated and extended without breaking the fact base. This flexibility to evolve schemas dynamically and incrementally is especially important in software understanding. New needs and concepts often arise over time. It is necessary to accommodate these needs in the schema without invalidating previous information.

DATA REQUIREMENT 1
Support dynamically evolvable schemas.

Graph models for information about software are natural, well understood, and widely used. Examples of graph models include entity-relationship models [22], software interconnection models [96], and semantic networks [37]. Nodes (entities) represent things and arcs (relationships) represent directed dependencies or associations. Multiple arcs are allowed between two nodes. The notion of type is used to classify nodes and arcs. For example, there may be typed graph nodes for procedures and variables and typed arcs for call relationships among procedures. Nodes and arcs have attached sets of attributes or properties. For example, a procedure node may have an attribute whose value is the filename of the file in which the procedure is defined. A call arc may have an attribute whose value is the line number in a file where the function call appears. Typically, all nodes or arcs of a given type have the same sets of attributes but differing attribute values. Informal commentary or annotation may be attached using attributes.

DATA REQUIREMENT 2
Use a graph model with multiple node types, arc types, and attached attributes.

To help people understand a graph model, there are several useful ways of or-

ganizing the model to manage complexity. In general, structuring notions may be applied broadly at the schema level or specifically at the fact level. (The schema describes valid classes of graphs; facts describe a particular graph.) Nested or inclusion graphs [48] have composite nodes that themselves represent (sub)graphs, thus imposing a kind of aggregation or containment relationship. Such relationships are useful in reverse engineering to produce hierarchies of grouping abstractions over related lower-level nodes and arcs. For example, the schema may express that file nodes aggregate one or more function nodes. A fact may express that some specific function nodes, related by some criteria, should be grouped. In general, there could also be composite arcs which contain other arcs and possibly subgraphs. Object-oriented models use inheritance or generalization/specialization to share and extend attributes among certain types of nodes and arcs (or among specific nodes and arcs). Constraints are enforceable rules that further govern the graph structure to ensure integrity. For example, a constraint may assert that all functions with a given prefix in their names are located in a specific file. A cardinality constraint may assert a one-to-many relationship between a source file and its function definitions.

DATA REQUIREMENT 3

Support aggregation, inheritance, hierarchy, and constraints.

Forward engineering focuses on mappings from requirements to implementation. Software understanding involves reconstructing inverse mappings from implementation to functionality to specific requirements and application concepts. That is, by studying the implementation, portions of the system are identified with certain functionalities and particular requirements. This reconstruction process involves several levels of abstraction for a software system, including text (source code), syntax, flow, structure, and function. The schema must be able to define, for example, valid source code, abstract syntax trees, control/data flow graphs, and architectural graphs. Different analyses may focus on different types of graphs.

DATA REQUIREMENT 4

Support the representation of software at multiple levels of abstraction.

The inverse mappings between abstraction levels have associated semantics. For example, there are semantics describing the syntactic meaning of the source code text. These semantics are typically encoded in the agents (tools or humans) that implement the inverse mappings. For example, parsers are tools that recognize textual source code and, based on semantic actions, build a syntax tree. Continuous software understanding requires incrementally carrying forward the inverse mappings to future baselines of the software. Also, traceability between artifacts at different abstraction levels requires that the mappings and inverse mappings be managed. For example, a developer may want to trace forward from a given requirement to the software components that implement it.

DATA REQUIREMENT 5

Represent the semantics of the inverse mappings and record instances of these mappings to enable traceability of software artifacts.

Over time, new types of information about the software are needed. A schema must be evolvable on demand to, for example, add new node types, add arc types, refine attached attributes, and specify default attribute values. Schemas need to be modular and composable to handle, for example, heterogeneous software systems that span multiple application and programming-language domains. Spatial and visual aspects in a schema should be separated concerns. Core parts of the schema should be distinguishable from extended, project-specific parts. Different aspects within the schema could be layered based on domain or project specificity.

DATA REQUIREMENT 6

Support multiple, composable, modular, dynamically extensible schemas.

At the fact level, actual nodes, arcs, and attribute values are expressed. It should be possible to edit the facts (e.g., create new nodes and arcs). Fact bases also need to indicate the schema definitions to which they conform and identify the portion of the software system that they represent. To support incrementality, collections of facts extracted from one portion (e.g., a compilation unit) of the software system should be composable and reconcilable with facts extracted from another portion. The policies should be clear about the resolution of conflicts.

DATA REQUIREMENT 7
Support multiple, composable, editable fact bases.

DATA REQUIREMENT 8
Provide version control over schemas and fact bases.

The computational structure used to access, manipulate, and otherwise reason about the facts must be capable of handling large, multi-million line software systems. Also, nodes and arcs need to be equally accessible entities through some application program interface. For example, a query may request all function nodes with a given filename attribute value. A query may request all call arcs that cross a system boundary. For flexibility, if a node can have a certain attribute (e.g., color, weight, label, etc.), arcs should have a corresponding attribute. An interface to query the schema itself is also needed.

DATA REQUIREMENT 9
Handle multi-million line software systems.

DATA REQUIREMENT 10
Provide a query mechanism that supports nodes and arcs as equally accessible entities.

DATA REQUIREMENT 11
Support introspection of schemas.

The metaschema, schema, and facts may be specified using some syntax or notation—the interchange format. The interchange format should be textual, so that it is human readable and easily edited. Policies for character encoding, whitespace, and quoting should be clear. The format should be lightweight in the sense that fixes for the quirks of a particular software system or language tend to be repairable and localized. Often, parsers have problems in generating facts because of these quirks. A textual format allows these problems to be more easily found using commonly available text processing and searching tools.

DATA REQUIREMENT 12
Use a simple, human-readable, lightweight format.

Three needs arise in dealing with software artifacts. First is the need to name certain entities uniquely. For example, local variables should be distinguishable from others of the same name (in other scopes). Second is the related need to locate an entity precisely and robustly. For example, a call statement should be locatable on some line number, in some file, within some directory, etc. Third is the occasional need to order (and reorder) entities. For example, the called functions of a given function appear in a natural order by line number. This order should be expressible in the immediate call tree descended from the given function. Policies to address these needs should be clear in the model.

DATA REQUIREMENT 13

Address the name, location, and order problems.

3.3.2 Relevant technologies

There have been a number of data interchange formats or languages for representing software.

Rigi Standard Format (RSF) [133] was developed as an input format for the Rigi software reverse engineering tool [81]. RSF is very simple, both syntactically and semantically. It is a textual, language-independent format, based on a typed multigraph model with directed binary relations [152]. The RSF schema or domain model lists the names of valid node types, arc types, node attributes, and arc attributes. Standard schema for a particular domain are beyond the scope of RSF. RSF facts specify directed arcs and node attribute values via three-element (verb, subject, object) tuples. For example, if `call` is an arc type between two function nodes, then the tuple `call main printf` declares the static existence of a procedure call between functions `main` and `printf`. If `lineno` is a node attribute, then the tuple `lineno main 10` declares that function `main` begins on line 10. The actual meaning behind names like `call` and `lineno` are externally encoded in the parsers used to generate RSF facts. RSF facts are stored in a file with one fact per line. The sequence of facts is essentially order independent, making the facts easily generated in pieces, composed, and sorted. The type of a node (or arc) is

carried in a mutable attribute.

There are many significant limitations with RSF. All node types have the same attributes, even if inappropriate. Arc attributes values cannot be specified (except in an private dialect). Nodes are named, but arcs are unnamed. For an arc type, the allowed types of the associated starting and ending nodes cannot be defined from among a finite set of two or more specific types. Annotations and source code fragments are stored in external files that are linked in as attribute values, making RSF files not self-contained. (The use of RSF tends to focus on the structural level of software abstraction, above syntax and text.) RSF fact files are separate from RSF schema files, with no formal way to indicate to which schema a fact file conforms. It is generally impossible to express structuring in the schema (except perhaps through comments and conventions).

Tuple-Attribute Language (TA) [51] was developed as an improvement to RSF to record information about the same kinds of graphs. TA has two sublanguages, with different interpretations depending on whether declarations appear in fact or schema sections. In the fact section, the tuple sublanguage of TA is similar to how RSF specifies arcs. The attribute sublanguage is used to specify the values of node or arc attributes. This sublanguage has similar capabilities to Graph Modeling Language (GML) [50]. In the schema section, the tuple sublanguage specifies valid arc types and what node types they relate. Also, the attribute sublanguage specifies node or arc attributes and default attribute values. Multiple inheritance is provided as a structuring construct in the schema to share attributes among various node or arc types. Each type can inherit attributes of its base types and can add additional attributes or override default values. Nodes or arcs of a derived type are acceptable anywhere the base type is allowed. Attributes can be grouped as a way to organize long lists of attributes. Facts can be bundled with the schema to which they conform in the same file. Files of schema and fact information can be composed via file inclusion. The TA syntax is clearly specified through a grammar. Standard schema for a particular domain are generally beyond the scope of TA. There have been attempts to define schemas for C, C++, Java, and a universal schema for procedural languages. These have usually focused on the structural level of software abstraction.

The XML Metadata Interchange (XMI) format [154] is based on the Object Management Group (OMG) [90] Meta Objects Facility (MOF). XMI specifies a mapping between MOF and XML Document Type Definitions (DTDs) and XML documents. XMI defines the standard mapping for the Unified Modeling Language (UML). UML is a standard and widely used object-oriented software design model. That is, XMI defines an XML representation for UML models and metamodels. UML, however, is not suited to the detailed, code-level artifacts of initial software understanding.

In summary, much discussion and research continues over standard software exchange formats, especially for reverse engineering and static program analysis. At this stage, an XML-based representation seems most likely, such as GraX [33], RDF [103], or an XML version of TA. Much work remains to define standard schemas for specific domains, such as for particular programming languages or general programming paradigms. Also, research is needed on refining the data requirements for dynamic program analysis to help relate dynamic and static analysis [127].

3.4 Control integration

Control integration entails capabilities for tools to control or coordinate each other on a shared task. There are many techniques for control integration, including the HP SoftBench Broadcast Message Server (BMS), Sun ToolTalk, Remote Procedure Call (RPC), and Java Remote Method Invocation (RMI). One versatile technique for control integration is to use a scripting language.

3.4.1 Pervasive scripting

The Notebook infrastructure depends on a scripting language that allows analysts to codify, customize, and automate continuous understanding activities. Scripts are written to enact reverse engineering patterns, commonly used tasks or solutions to produce understanding in particular situations. The analyst can comple-

ment the built-in operations with external algorithms for graph layout, measures, software analysis, etc. Complex analysis tasks can be automated for more consistency and repeatability. For example, some analysis tasks require significant human input and a time intensive process. The final answer of such an analysis should not be the only tangible result. The decisions made could serve to ease a similar analysis on another version of the software. Consequently, there are benefits to the ability to repeat these analysis decisions over a new revision after an incremental, evolutionary change. By reenacting these decisions automatically, human effort is reduced.

The notion of pervasive scripting involves several concepts:

- scriptability,
- recordability,
- attachability,
- constructability, and
- undoability.

Scriptability is essentially programmability of an application and serves as an extension, customization, and integration mechanism. Recordability requires the ability to capture scripts from interactive input to an application. (Non sequential actions, however, would require some manual editing.) Attachability requires that scripts be attachable to virtually any action, event, or artifact of interest (to attach special semantics or behavior). Constructability requires that derived information, abstractions, and views be generated through scripts. Undoability requires that actions be reversible.

3.4.2 Scripting language

Although scripting languages may not solve some challenges of tool integration (such as registration and encapsulation), they are useful for prototyping tool envi-

ronments.

The Notebook scripting language is based on Tcl, a mature, proven, and multi-platform technology. Tcl is a language for sequencing and invoking tools and tool operations [91]. Tcl was developed to address the problem that many tools have their own little languages to automate actions. Tcl provides a core command language and an extensible, embeddable interpreter. The core language can be extended with high-level, application-specific commands and constructs.

The following scripts illustrate salient aspects of the scripting language, including graph layout and traversal.

This script command `columns` arranges the nodes in the active window into columns by their node type.

```
proc columns {} {
    # select all nodes in the active window
    rcl_select_all

    # for each node that is selected ...
    foreach nodeID [rcl_select_get_list] {
        # check off its node type as being used
        set nodeTypes([rcl_get_node_type $nodeID]) 1
    }

    # get the number of distinct node types
    set numNodeTypes [array size nodeTypes]
    if {$numNodeTypes == 0} {
        # do nothing if the window has no nodes
        return
    }

    # set horizontal cursor increment and starting point
    set xDelta [expr [rcl_win_canvas_width] / $numNodeTypes]
```

```
    set xPos [expr $xDelta / 2]

    # for each node type used ...
    foreach nodeType [array names nodeTypes] {
        # select all nodes in the active window of that type
        rcl_select_type $nodeType
        # set the cursor
        rcl_cursor_set $xPos 0
        # lay out the selected nodes vertically at the cursor
        rcl_group_vertically
        # step the cursor
        incr xPos $xDelta
    }

    # scale the nodes to fit inside the active window
    rcl_scale_to_window
    # deselect all nodes in the active window
    rcl_select_none
}
```

In the following example, for a selected subsystem node, the `slice` script extracts dependency information and the `circle` script presents this information in a circular layout with four quadrants containing:

- internalizations in the selected subsystem (northwest),
- leaf nodes to and from which the subsystem provides and requires (northeast),
- leaf nodes to which the subsystem provides (southwest), and
- leaf nodes from which the subsystem requires (southeast).

```
proc slice { pA } {
```

```
set S [rcl_select_get_list]

set rn {}
set pn {}
set in {}

foreach s $S {
  # get exact interface
  set ei \
  [typed_node_interface $s $pA]

  # arcs are returned
  set ra [lindex $ei 0]
  set pa [lindex $ei 1]
  set ia [lindex $ei 2]

  # get nodes
  foreach n $ra {
    ladd rn [rcl_get_arc_dst $n]
    ladd in [rcl_get_arc_src $n]
  }
  foreach n $pa {
    ladd pn [rcl_get_arc_src $n]
    ladd in [rcl_get_arc_dst $n]
  }
  foreach n $ia {
    ladd in [rcl_get_arc_src $n]
    ladd in [rcl_get_arc_dst $n]
  }
}

set common [lintersect $rn $pn]
set rn [ldiff $rn $common]
```

```
  set pn [ldiff $pn $common]

  # get all nodes
  set an {}
  foreach n $rn {
    lappend an $n
  }
  foreach n $pn {
    lappend an $n
  }
  foreach n $common {
    lappend an $n
  }
  foreach n $in {
    lappend an $n
  }
  return [list $rn $pn \
    $in $common $an]
}

proc circle { L } {

  rcl_select_all
  rcl_open_projection

  set q(0) [lindex $L 0]
  set q(1) [lindex $L 1]
  set q(2) [lindex $L 2]
  set q(3) [lindex $L 3]
  puts [lindex $L 4]

  rcl_select_none
  foreach n [lindex $L 4] {
```

```
    rcl_select_id $n 1
  }
  rcl_select_invert
  rcl_filter_selection

  rcl_select_all
  rcl_scale_none
  rcl_set_scale_factor 25
  rcl_scale_by_factor
  # got all the nodes involved

  set a 500
  set b 500
  set pi 3.1415926

  for {set i 0} {$i < 4} \
  {incr i} {
    set d [expr ($pi / 2) / \
      ([llength $q($i)] + 2)]
    set t [expr $i * $pi / 2]

    foreach n $q($i) {
      rcl_set_node_position $n \
        [expr round($b * cos($t)) + $b] \
        [expr round($a * sin($t)) + $a]
      set t [expr $t + $d]
    }
  }
  rcl_refresh
}
```

An example invocation that focuses on call dependencies is:

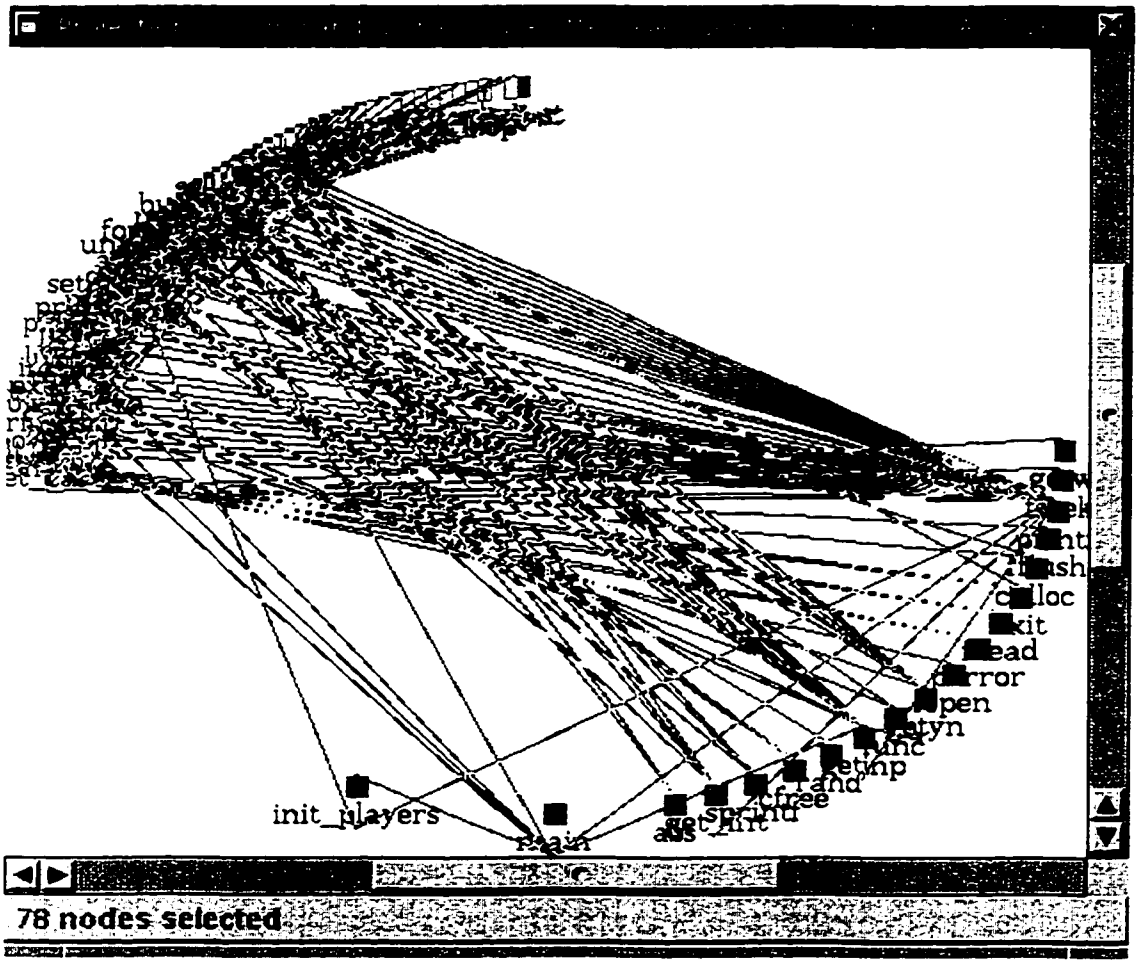

Figure 3.1: Circular layout of dependencies

```
circle [slice [list call]]
```

The resulting layout for the Play subsystem of a Monopoly game [125] appears in Figure 3.1.

3.5 Presentation integration

3.5.1 View management

Many software systems have internal documentation that is often out-of-date and thus unreliable. Even when the documentation exists, it may be dispersed in several places and may not be well structured. Yet accurate, complete, well-organized, and maintainable documentation is critical for software maintenance. As such, a programmer often falls back on the source code to understand how the software works, spending much time and perhaps duplicating the efforts of others. This problem becomes more critical when maintaining large, legacy software systems. Thus, there is a need for a process called redocumentation [40]. One key goal of the Notebook is to support redocumentation effectively.

Reverse engineering tools present pertinent summaries for maintenance and reengineering (program understanding) purposes. These summaries may be presented in visual or textual *frames* that are derived from information in a centralized repository. A visual frame might contain, for example, a high-level overview of the system, a function call graph, an inheritance hierarchy, or a control/data flow chart. A textual frame might contain, for example, software-metric results or a cross-reference listing. A frame displayed in a window need not be static, but can be fully interactive. For example, one can adjust a module interconnection graph in a visual frame by grouping, arranging, filtering, scaling, clustering, and editing operations [135]. One can select a function listed in a text frame and link to additional documentation that describes the function in more detail [134]. This auxiliary documentation might be external to the repository. Moreover, the frames might be derived from many different tools.

Pictorial, animated, audio, and video frames can serve as annotation. Frames can be bundled and saved into a *view* for redocumentation purposes [135]. For flexibility, the constituent frames should have a well-defined constructive representation and be dynamically updated. The constructive representation is a script for regenerating or retrieving the frame. The script also allows the possibility of ad-

justing, filtering, or even modifying the contents of a frame prior to presentation. Dynamically-updated frames refresh themselves to portray the appropriate subset of the repository (or auxiliary documentation) accurately at all times. Thus, unlike traditional software documentation and static images, a view remains consistent and up-to-date.

These views can be organized into *global* views for the workgroup and *local* views for the individual user. Moreover, the relevant views can be assembled and targeted for different classes of users (designers, programmers, and managers), different levels of experience (beginners versus experts), and different kinds of purposes (guided tours, overviews, and details). This improves program understanding and eases decision-making by programmers and especially managers [135]. Thus, views are a high-level abstraction for incorporating more application and domain-specific types of knowledge.

Further structuring mechanisms are necessary to manage the large number of created views that is needed to document a large software system.

Subviews

Views can be enhanced to contain a pointer to other views, forming a *subview* relationship between views and building a hierarchy of views. This idea allows views to be tailored or specialized for different interpretations more easily and to share effort in their construction in a way similar to component-based development. A manager may want the same high-level overview as a programmer, but with additional annotation frames to present, for example, personnel assignments, funding levels, and scheduling information. The regeneration of a view becomes recursive; changing a view updates the views to which it is a subview.

Sequenceable views

Analogous to hypertext/hypermedia, links can be used to associate arbitrary views (and frames), forming a web of views. However, for a large enough number of views, there are significant problems with user disorientation during navigation and cognitive overhead for managing the links [24]. One solution to these problems is to use the concept of a *path*, an ordered traversal of some links in the web [157]. The notion of a *sequenceable view* unifies subviews and sequential paths. The top-level views and frames of a sequenceable view are linearly ordered and played back one at a time. When encountering a sequenceable view during the playback, the user can be given the choice to skip past it or to "dive" into it. This capability provides a simple roaming and zooming mechanism as well as a branching technique. Note that there could be a mix of unordered and sequenceable views in the hierarchy.

Template views

To maintain consistency and completeness between different sets of views, possibly for differing software versions and variants, there is a need for standardized sets of views. The notion of *templates* [42, 61] or frameworks for standardizing software documentation and hypertext structures can be applied to views. Another benefit is that template views promote reusability.

A parameterized script representation for frames is needed. Template views are instantiated into real views by binding the parameters of the constituent frames appropriately.

Inclusion graphs

To enhance understanding and promote insight, visual representations of software are generally desirable [12]. Thus, a visual method for presenting the structure of a set of views is needed. One promising approach is to use inclusion graphs where

views and frames are represented by labelled boxes (rectangles) and the subview relationship is represented by the recursive nesting of boxes. The chief advantage of this "space-filling" approach to visualizing and browsing hierarchical structures, versus a tree or graph drawing, is the efficient use of display area [58]. This technique has been used for visualizing designs, where smooth navigation among drastically different levels of abstraction is required [47]. With some enhancements for position, size, color, and font cues, the technique has also been used for visualizing very large knowledge structures in a "virtual museum metaphor" [139].

Canvases

The inclusion graph, up to some depth, can be rendered in a single window on the screen. The window or *canvas* becomes the working area and focus of related view construction, editing, and navigation activities. Multiple canvases are possible; each could capture a different set of environmental and configuration possibilities of the subject software. Views begin as empty boxes and can be edited by dragging and dropping the boxes of frames and views inside each other. Boxes can be "opened" to generate, playback, and present their contents (frames and/or views). Boxes can be "raised" to allow deeper structures to be rendered and "folded" to mask their structure.

Metaviews

It is instructive to think of a canvas as simply another kind of frame that provides a high-level "map" for a number of views. Bundling one or more of these canvas frames into a view forms a *metaview*; symbolizing one or more of these metaviews on a canvas forms a *metacanvas*. The number of levels of abstraction is unlimited. This allows the organization of views to be scalable, a necessity for understanding large software systems. One could, for example, use a metacanvas to design a map that shows the different sets of view documents for different releases of the software (according to different baselines of the repository). This is useful to separate

and manage the program understanding analyses.

3.6 Chapter summary

This chapter introduced the Notebook concept and approach. Aspects of a supporting infrastructure for the approach were discussed. These aspects included data, control, and presentation integration. To place this infrastructure into action, issues of process and methodology are covered in the following chapter.

Chapter 4

Notebook Process

Defining a workable and feasible methodology is a particularly difficult problem of any technique. This chapter discusses process and usage issues of the Notebook, including user roles and methodology.

4.1 User roles

There are specific user roles involved with the Notebook. It is useful to separate the roles to avoid mixing responsibilities or forcing end users to become tool developers. There are three roles (and corresponding perspectives) involved in directly constructing, evolving, and using a Notebook: the builder, the mediator, and the end user. A role may involve several people and a person may act in more than one role.

The three roles of builder, mediator, and end user are increasingly project-oriented and task-specific. The builder focuses on generic mechanisms that are useful across multiple application domains or evolution projects over a long period. The mediator focuses on generating information and writing scripts that are useful to a particular evolution effort, but across multiple end users, to help lower the effort of

adopting the Notebook in practice. The end user focuses on obtaining fast access to information relevant to the specific understanding task at hand.

The roles have correspondingly different requirements. The roles are related in that the mediator must satisfy the needs of the end user, and the builder must satisfy the needs of the mediator (and indirectly the end user). Consequently, the builder and mediator must have more than their own requirements and perspectives in mind.

4.1.1 Builder

The builder constructs the Notebook infrastructure (as discussed in Chapter 3), focusing mostly on generic mechanisms for gathering, structuring, and storing information to satisfy the needs of the mediator. The builder also integrates tools, such as parsers, analyzers, converters, and visualizers to allow the mediator to enter information from a variety of sources.

Since the information needs of the mediator and end user cannot all be foreseen, the builder requires conceptual modeling and flexible information storage and access capabilities that are extensible enough to accommodate diverse types of content. Also, the builder requires flexible tool integration mechanisms to allow access between the Notebook and other analysis or visualization tools. Finally, the builder requires that the implementation of the Notebook infrastructure be based on standard, widely available technologies, to ensure that the Notebook infrastructure can be easily modified or ported to new platforms.

4.1.2 Mediator

The mediator enters information into the Notebook that is specific to the software system of interest. Sources of information may include source code files and documentation. The mediator assesses the usefulness of each piece of information based on the future understanding needs of the evolution project. The mediator

must also reconcile conflicting information, perhaps in old documentation, with the actual software system. The mediator may also generate new content, such as architectural views derived from discussions with the original software developers. For continuous understanding and ease of maintenance, these views must be composed constructively over the software using scripts so that they can be reconstituted and carried over to later baselines of code. The mediator also writes scripts to enact useful analyses or produce useful visualizations.

4.1.3 End user

The end user accesses the Notebook content and could be a developer, manager, or anyone needing more detail about the legacy code. The end user is able to browse and restructure the existing content and add personal annotations to highlight key issues. As well, the end user can generate new information specific to the task at hand by running analysis and visualization scripts. For the end user, the Notebook is a unified combination of documentation and toolset that has been customized and targeted for the project.

The end user has a few major requirements. Most importantly, the Notebook content must pertain specifically to the evolution project and be accurate, well organized, and easily accessible. The end user also requires the Notebook environment to be effective, easy-to-use and yet flexible enough to assist diverse understanding tasks.

4.2 Methodology

The methodology used to instantiate a Notebook and enable continuous understanding contains the following major activities:

- evaluating supporting technologies (builder),

- integrating analysis and visualization capabilities (builder),
- creating a conceptual model (mediator),
- extracting software facts (mediator),
- constructing analyses and views via scripts (mediator and end user), and
- exploring analyses and views (end user).

4.2.1 Evaluate supporting technologies

The Notebook builder evaluates potential technologies for the Notebook infrastructure to enable continuous software understanding (as introduced in Chapter 3). For example, pervasive scripting was proposed as a basis to record analyses and generate reusable scripts. Consequently, one need is selecting an embeddable scripting language upon which a recording capability can be based.

A potential such language is Tcl, which provides a library that is linked into applications. The library contains an interpreter for which new commands can be registered to call application-specific routines, such as Notebook analysis operations. All Notebook analysis operations pass through this interpreter. The interpreter then serves as a point where a hook is inserted to record the commands and parameters being executed during an analysis. The recorded sequence of these commands is saved as a script or *constructive analysis*. That is, not only is the result of an analysis important; the steps needed to construct or realize it are also equally important and potentially reusable.

4.2.2 Integrate analysis and visualization capabilities

The Notebook builder needs to maintain and deliver diverse capabilities for analysis and visualization. Some of these capabilities may be provided by external tools, including graph layout, metric, and clustering algorithms. Thus, the builder

must apply tool integration technologies in the Notebook infrastructure to access these capabilities. For example, to exchange information between the Notebook and other tools, common data interchange formats are used. To coordinate the invocation of external tools by the Notebook a scripting language is used. The scripting language also allows external tools to customize the Notebook as a programmable component. The Notebook architecture is centered on an extensible, general, lightweight tool with innate scripting and graph representation capabilities. More domain-specific capabilities are accessed externally by connecting to other tools and/or programmed internally through scripts.

4.2.3 Create conceptual model

To represent concepts in the application and implementation domain, a conceptual model or domain model is used [133]. This model is created by the mediator for a particular project. One goal is to keep the Notebook infrastructure relatively generic and make it customizable to the specifics of different domains. For example, in the implementation domain, software concepts such as files, procedures, global variables, user-defined types can be defined as specific types of entities or nodes in a graph. Dependencies such as file containment, procedure call, and data access can be defined as relationships or arcs in the graph. Locations of artifacts in source code can be expressed in pathname and line number attributes. Attributes can also provide links to annotations, external documentation, and fragments of source code. The domain model serves to specify valid classes of graphs.

As an example, the following is an RSF domain model for C.

Node types:

```
Unknown
Collapse
File
Proc
GlobalVar
```

```
UserType
```

Arc types:

```
level    Unknown Unknown
composite
infile   File    Unknown
call     Proc    Unknown
access   Proc    Unknown
```

Node attributes:

```
attr     Node    loc
attr     Node    endloc
attr     Node    position
attr     Node    annotate
attr     Node    scope
attr     Node    lineno
attr     Node    file
attr     Node    nodeurl
```

Arc attributes:

```
attr     Arc     loc
attr     Arc     name
attr     Arc     annotate
attr     Arc     inheritance
attr     Arc     arcurl
```

4.2.4 Extract software facts

To produce the specific graph of entities, relationships, and attributes for a given piece of source code, the Notebook mediator performs a process of fact extraction

(see Figure 4.1). This process may involve parsers that automatically traverse the source code, possibly also consulting support systems for software configuration management and project management. For various entities, such as a function definition, the source code fragment is linked in as an attribute. There is a mechanism to access the source code for a function node in the graph. Facts that are not necessarily expressed in the software can be added as new entities, relationships, or attributes. For example, a subsystem entity could be used to express that a group of functions are related because of common authorship. Attributes could be used to record management information, such as status and personnel assignments of various modules in the software. Since some parts of the model may not have been entirely foreseen, the Notebook support for domain models must be incrementally evolvable and extensible.

To manage the complexity of the constructed graph, the extraction phase is following by a largely semi-automatic one where the mediator and end user further elaborate the graph with informal information and simplify the graph by identifying subsystems.

4.2.5 Construct analyses and views

One reverse engineering analysis is subsystem abstraction, a recursive process whereby software building blocks such as data types, procedures, and subsystems are clustered into composite subsystems [83]. This analysis builds multiple, layered hierarchies as higher-level abstractions that reduce the complexity of understanding large software systems (see Figure 4.2). The criteria for what comprises a subsystem depends on the analysis goal, audience, and application domain. For example, a subsystem may represent an abstract data type, a functionality, authorship, defect category, level of editing activity, or any clustering concept. The mediator or end user consults a visual representation of the graph to apply human pattern matching capabilities to recognize meaningful structures. For example, vertical and horizontal slices of the abstracted graph can be shown (see Figures 4.3 and 4.4).

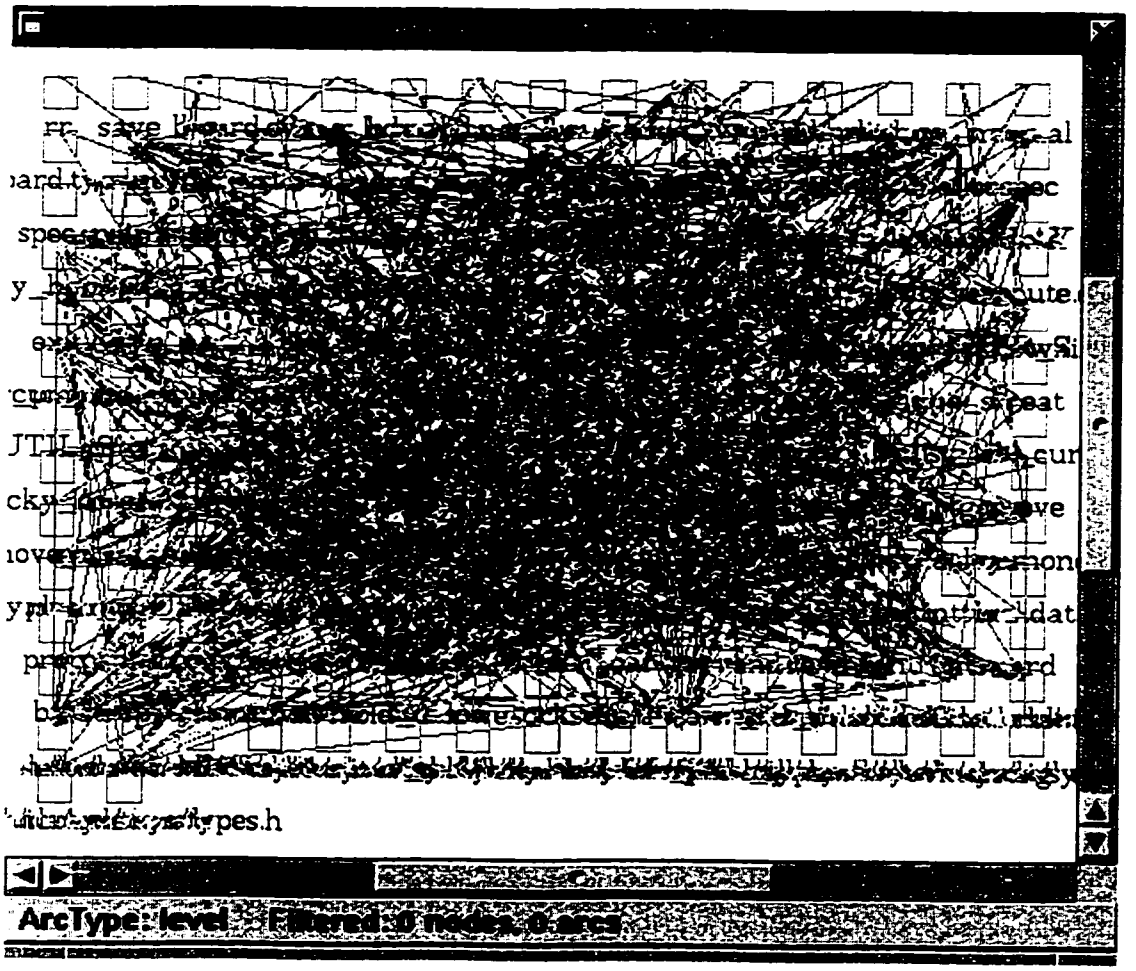

Figure 4.1: Initial flat graph of software structure

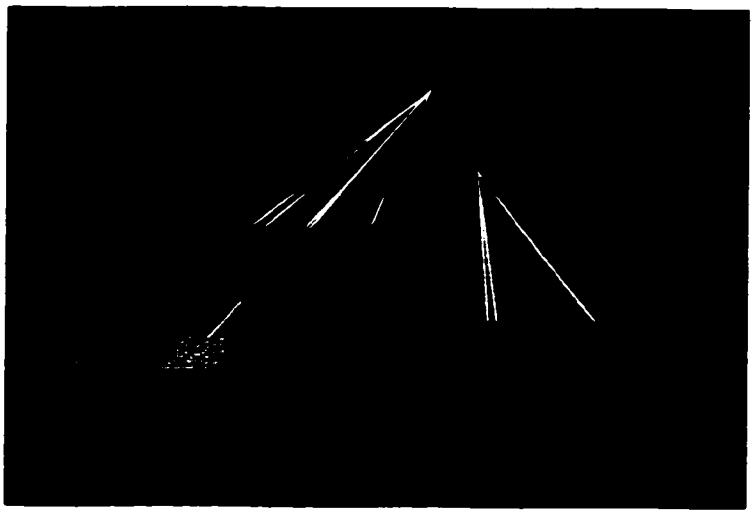

Figure 4.2: A subsystem abstraction hierarchy

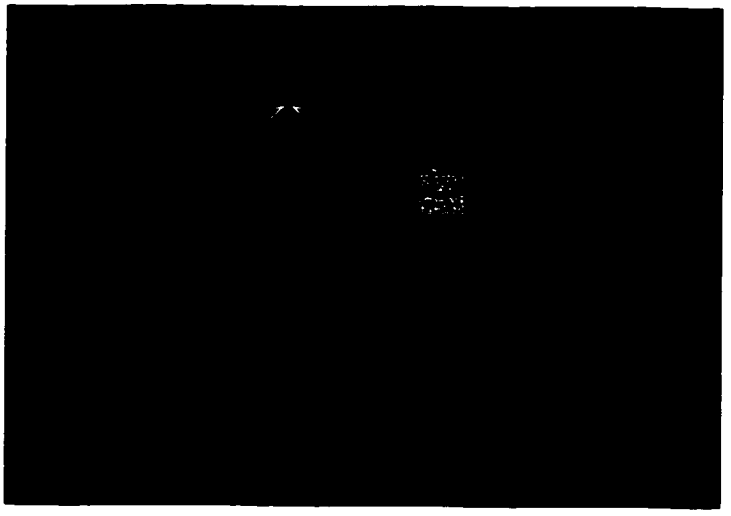

Figure 4.3: Vertical slices through the subsystem hierarchy

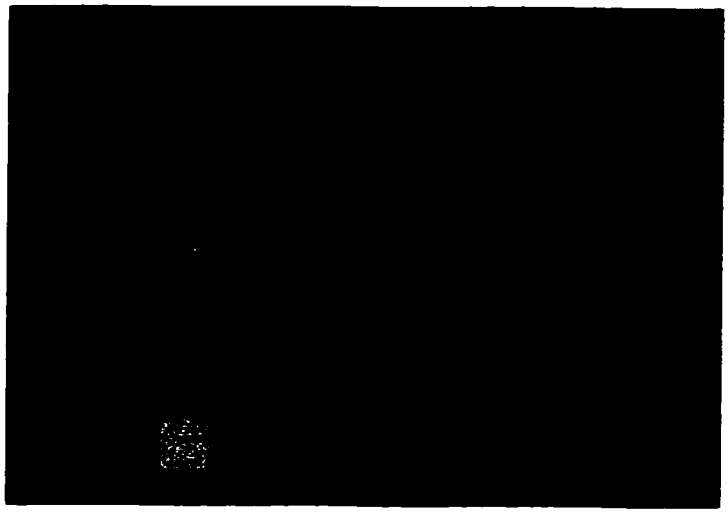

Figure 4.4: Horizontal slices through the subsystem hierarchy

Ideally, end users should perform the subsystem abstraction process themselves as a way to learn the software system (i.e., learning by doing), but the process is time-intensive. For speed, they may want a fully automatic process, but there is a conflict in that they will likely not trust the results. To address this problem, end users and mediators create scripts by recording the sequences of spatial analysis operations invoked to identify and cluster subsystems. The scripts then specify a *constructive abstraction*. These operations are replayed to reconstitute the human-directed analysis for sharing with other end users (possibly on other baselines of code). See Appendix A.1 for an abstraction script that produces a decomposed graph. A vertical slice of the decomposition is shown in Figure 4.5. A horizontal slice is shown in Figure 4.6.

A Notebook view is a snapshot or bookmark that reflects the spatial state of the graph model and the visual state of the user interface. In particular, a view is a reloadable bundle of visual and textual frames that contain, for example, call graphs, overviews, reports, and annotations. Each view reconstitutes a particular perspective to highlight maintenance constraints and problems. The focus is to construct readable, accurate, and up-to-date documentation about the subject system. A view is implemented constructively through a script. That is, the se-

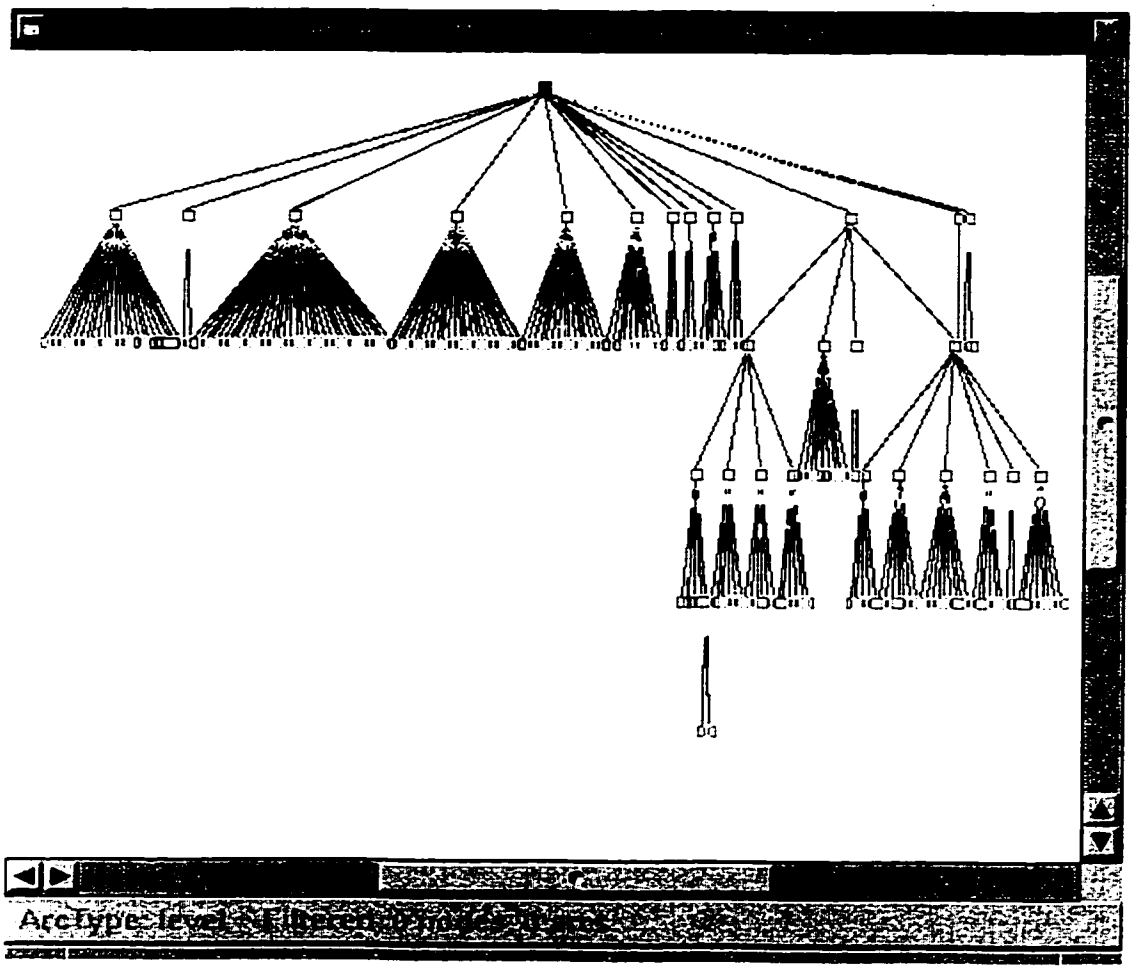

Figure 4.5: Overview after subsystem abstraction

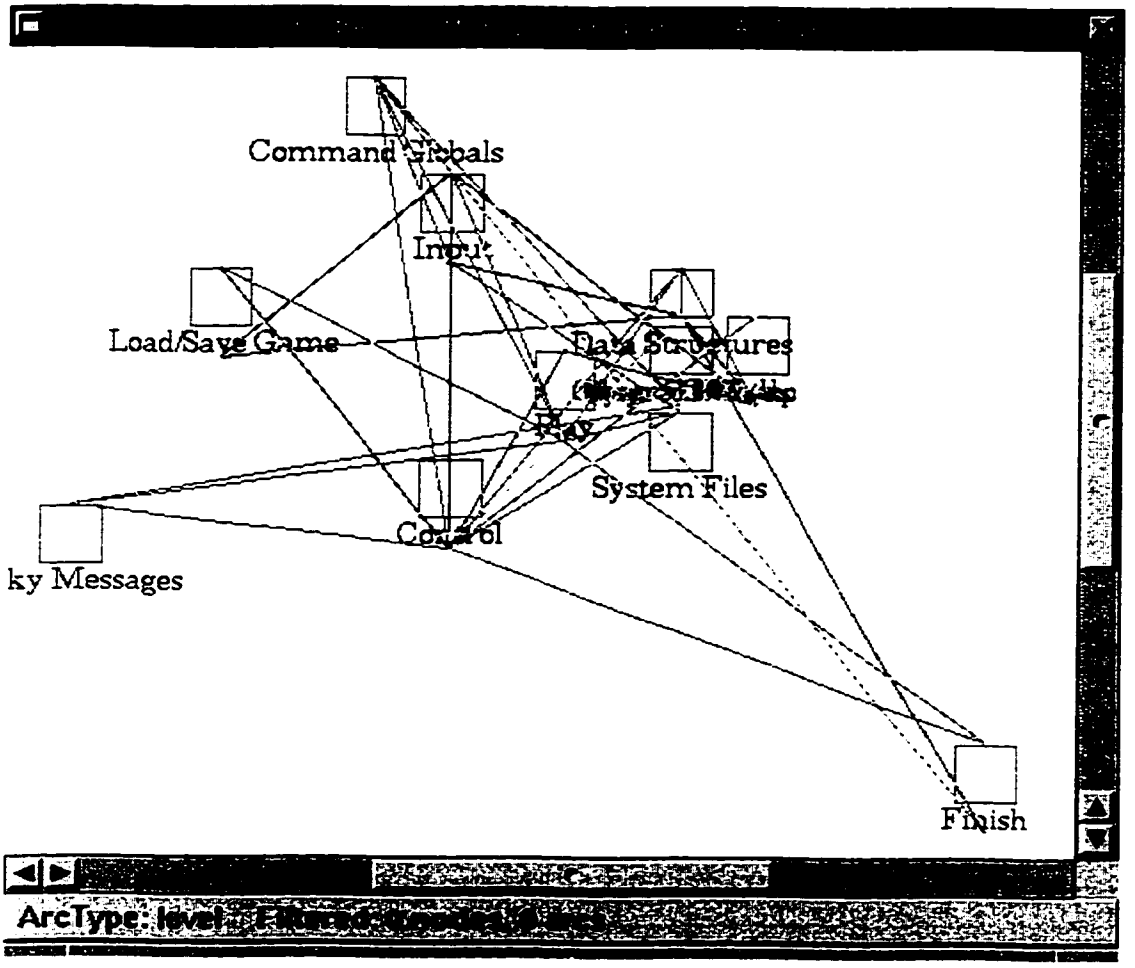

Figure 4.6: Children of the root after subsystem abstraction

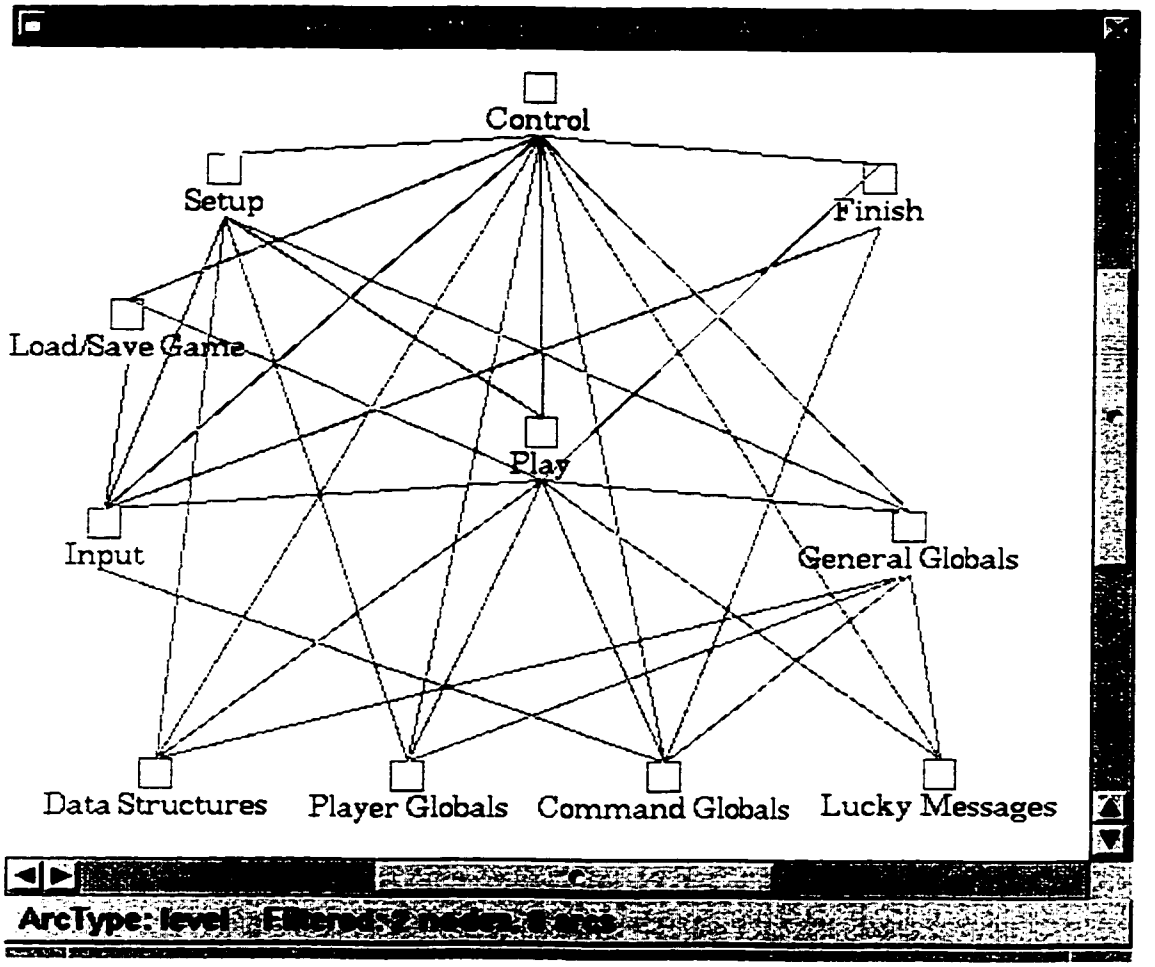

Figure 4.7: Graph after layout

quence of visual operations invoked to lay out and filter graph nodes and arcs are recorded as *constructive views*. See Figure 4.7 for the final layout.

4.2.6 Explore analyses and views

To explore structural subsystem abstractions and their views, the end user runs the appropriate constructive script to reconstruct them from the source code or prepared fact base. The resulting subsystem structures can be further analyzed and

manipulated, forming an additional, personalized layer of recorded spatial and visual operations and other annotations. For example, a view could be adjusted to better highlight anomalies of the software structure for managing risk and deciding personnel assignments. Highly complex, central components are then best handled by the senior maintainers. Dependencies between two subsystems, each authored by a different programmer could indicate areas where the programmers need to coordinate their changes.

4.3 Chapter summary

This chapter described the Notebook methodology and responsibilities from the point of view of three user roles: the builder, the mediator, and the end user. Still, significant adoption challenges face the Notebook. The next chapter describes an approach to address these challenges.

Chapter 5

Notebook Adoption

This chapter covers aspects of Notebook adoption and contains many lessons of our tool adoption experiences. Adoption is probably the single most difficult problem. After several years of research with IBM, we have not solved this problem.

In general, tools must also fit into existing processes at both large and small scales. For adoption, tools must be compatible with the engineering processes of the organization. For effectiveness, tools must support the personal, preferred comprehension strategies of individual developers. In short, tools need to be used to be effective; tools need to be effective to be used. The exploration of adoption issues helps to refine the Notebook technology; the improvement of the technology helps to streamline its use.

5.1 Technology insertion

For improvement in understanding software and making changes, we need a well-defined process for understanding and making change. We need to understand the task structure, have clear goals, and enable planned solutions. New technologies such as the Notebook are inserted in the context of existing end user goals and

tasks, not for their own sake.

As a step to rationalizing the reverse engineering process, the Notebook concept aims to:

- identify reverse engineering analysis and visualization scripts,
- insert these actions into a goal-directed framework, and
- enact these actions.

5.1.1 Change process

The software change process and the role of analysis and reverse engineering tools within that process are important for evolving quality software.

The fundamental business goals that drive the evolution of a software product [46] are the needs to: maximize customer satisfaction, minimize effort and cost (to increase earnings), minimize defects, and minimize time to market. Software analysis and understanding play a large role in achieving these goals throughout the lifetime of the product.

Software changes to address these goals can be categorized into three types: adaptive changes, perfective changes, and corrective changes. Adaptive changes address customer needs and accommodate technological improvements. Perfective changes seek to make future evolution somehow better, more manageable, and less costly. Corrective changes focus on detecting, tracking, and diagnosing defects and their root causes. The effective management and execution of all these types of changes is critical.

5.1.2 Analysis questions

Changes generally arise by reviewing customer needs for new features and considering requests for fixes. Each change leads to a specific change process where it is typically planned (scheduled), analyzed, developed, inspected, and tested. Software understanding can play an effective role in the analysis part of the process. Analysis results can then be used as input into code modification, inspection, and metrics gathering activities.

The analysis must answer several questions:

- what is the change,
- where to place the change,
- how to carry out the change,
- when to perform the change,
- who should perform the change, and
- whether to perform the change.

The final question studies the costs, benefits, impact, and feasibility of the change. This provides a deeper analysis of the change beyond the initial review and may result in further decomposing the change into smaller pieces or rescheduling it. Various agents are involved in producing the answers to these questions. Tool support, such as the Notebook, can aid programmers in answering the where and how questions and help managers in answering the when, who, and whether questions.

5.1.3 Goal-directed framework

The complex nature of analysis and understanding suggests a goal-directed approach oriented to actual needs (and the software changes that arise) instead of a

technology-oriented approach on perceived needs. That is, program understanding technologies such as slicing [71], structural redocumentation [153], pattern matching [95, 29], defect filtering [18, 140], and program visualization [99, 112, 31] need to be inserted within the context of actual needs.

A goal-directed approach is more task-oriented and tends to avoid the situation of a solution looking for a problem. The goal-directed framework is based on paradigm of goal, question, analysis, tool, and action (inspired by [66] and GQM [9]). The paradigm guides the Notebook end user from business, quality-related, or project-specific goals and subgoals, through specific questions, through relevant program understanding analyses, through appropriate tools and features, to recommended actions for achieving the goals. The framework is a context for inserting reverse engineering analysis capabilities of the Notebook to serve the required goals and questions of the Notebook end user. The framework seeks to match end user needs to Notebook capabilities.

Typically, a goal leads to finer subgoals and a tree of questions, analyses, tools, and actions. A goal-to-action path is called a scenario and represents a distinct task. The analyst generally passes through multiple, related, possibly prioritized scenarios for analyzing a change and achieving a high-level goal. The compilation of such scenarios forms a useful, high-level framework for the introduction of program understanding techniques to organizations as well as individual developers and managers.

The paradigm is an opportunistic, short-sighted, but "real-world" strategy. Tools and techniques are only invoked on a lazy, as-needed basis to answer some direct question and achieve some goal. In general, this strategy is natural as the understanding process itself is opportunistic, iterative, and piecemeal, where analysts make intelligent guesses, gather many different pieces of information, and then integrate the pieces in some sensible way. Mechanisms and novelties become less important than immediate results.

5.1.4 Decision factors

Potential analyses may further be organized along several dimensions, based on characteristics of the task (e.g., scope, granularity, abstraction level, automation level), the program (e.g., language, type of program), and the analyst (e.g., knowledge, user expertise). For scope, a task may focus on global and interprocedural constructs (e.g., global data types) or focus on local and intraprocedural constructs (e.g., local variables). For granularity, a task may require a high-level, large-scale, coarse-grained overview of a software system (e.g., software architecture and architectural style [117]) or require a low-level, small-scale, fine-grained detail of a specific data structure or algorithm. For abstraction level, a task may focus on textual, syntactic, structural, functional, behavioral, and application aspects of the software. For example, cut-and-paste reuse analysis [60] focuses on the source text, including whitespace and comments. For automation level, a task could be fully automatic and computerized, semi-automatic with the user interacting with a computer, or manual. Programs are written in a variety of implementation languages and paradigms. Analysts may contain novice, intermediate, and expert users, with differing levels of application and implementation domain knowledge.

These dimensions, along with other system constraints and cost limitations, form a basis for making decisions among potential analyses and tools.

5.1.5 Example scenarios

The goal-directed framework is aimed toward end user activities. Builders and mediators are more forward-thinking and focus on longer-term issues of continuous understanding. Essentially, they champion and manage the Notebook approach and have a more cosmopolitan viewpoint [101]. For example, the mediator creates conceptual models, parses collections of code, and writes scripts, which are somewhat tangential to actually exploring code. That is, housekeeping and support tasks are outside the strict goal-directed strategy. Nevertheless, tasks such as code preparation, parsing, and population of a repository should be streamlined

and automated as much as possible to lower a potential barrier to adoption. The task of writing scripts can be partially streamlined by recording them. In general, there is a need for lightweight, easily repairable tools without too much built-in state. Large overheads could indicate future scalability and adoption problems.

Various scenarios may be applicable to answer the analysis questions (what, where, how, when, who, whether) depending on the overall goal.

For an adaptive change request to add a feature:

- **goal:** maximize customer satisfaction
- **question:** what, where, who to place the changes
- **analysis:** structural redocumentation
- **tool:** Notebook constructive views
- **action:** reconstitute and consult architectural view

Initially, a global viewpoint is needed. Structural redocumentation is a process of producing architectural views of the software using subsystem abstractions [153]. As decision factors, scope is global, granularity is large, abstraction level is structural, and knowledge is application specific. For the Notebook, most of the process would be conducted by the mediator, leaving the task of retrieving and consulting the architectural views to the end user (to answer the immediate questions and the real problem).

Subsystems abstractions based on functionality can be consulted to determine what and where in general to add a feature. The complexity of the graphical view for this functionality can suggest who should perform the software change. For increased detail, a dependency analysis may be requested (see below). Subsystem abstractions based on past personnel assignments can be consulted to determine who has the appropriate experience to add the feature.

For the same change request:

- **goal:** maximize customer satisfaction
- **question:** whether to perform the change
- **analysis:** dependency analysis
- **tool:** exact interfaces [82]
- **action:** perform change if no external components are affected

The analysis determines the impact of changes to other components should the given component be modified. The extent or scope of this impact is useful information for assigning personnel and estimating the effort to perform the change.

An exact interface report for a selected subsystem provides three kinds of information:

- provisions,
- requirements, and
- internalizations.

A provision is a dependency from a node inside the subsystem to a node outside the subsystem. A requirement is a dependency from a node outside the subsystem to a node inside the subsystem. An internalization is a dependency between two nodes inside the subsystem. To determine potential change impact, the provisions need to be consulted.

5.1.6 Transition

It is likely that new tools, techniques, and processes associated with the Notebook cannot be inserted into the existing change process without further preparation and broad management support. The selection of new tools and techniques requires much evaluation and a metaprocess in itself. Moreover, we need to prepare

the new target process, understand the inherent methodologies assumed by the Notebook capabilities, and define a process for the transition. The transition may involve skills assessment, user training, motivation, data conversions, new terminology, assignment of roles, and changes in operating procedures. These activities need to be enumerated and planned so that they can be budgeted and measured in terms of cost and status of progress. The benefits of the Notebook approach has to be gauged against not only the cost of the transition, but the total lifecycle cost.

5.2 Technology adoption

Technology adoption issues are of major concern to software engineering researchers and are difficult to address. The largest barrier to adoption may stem from the perceptions of end users. In general, several perceptual factors contribute to the rate of technology adoption, including relative advantage, compatibility, complexity, trialability, and visibility of results [111]. These factors form a major design criteria for the Notebook. Most related approaches neglect adoption issues. For the Notebook, the issues are tackled directly. Specifically, there are several needs of end users to keep in mind.

5.2.1 End user needs

Tools must solve a real, immediate problem and provide a clear, obvious, visible benefit. The Notebook concept is generally oriented for long term continuous understanding to enable continuous change in large software systems. Nevertheless, the goal-directed framework is a way to understand the placement of various program understanding technologies provided by the Notebook toolset to answer specific questions about software changes. Tool research should not be entirely focused on new paradigms, but should address real user needs and expectations.

Tools must be compatible with existing tools, users, and processes for developing software. The Notebook toolset can invoke development tools [152] (e.g., to edit a

source file) and be invoked by another tools as a component [39] (e.g., to retrieve a view). The aim is to avoid taking a self centered approach whereby the Notebook becomes the only access point to code and documentation. Builder, mediator, and end user responsibilities are separated, allowing the end user to focus on the task at hand.

Tools must not require full access to the source code. The Notebook does not own the source code nor manage access to it. The Notebook infrastructure is focused on producing reusable derived information, leaving the actual management of original (source) information to a software configuration management or version control system [14].

Tools must be small, easy to install, simple to try out, and fast. The Notebook infrastructure generally focuses on simple, familiar approaches, rather than heavyweight tool integration approaches that are difficult to administer, install, and extend. Increased perceived complexity of tools decreases perceived productivity and quality [56].

Tools must support open standards. The Notebook infrastructure aims (in the long term) to use non-proprietary standards and open source technologies (e.g., Tcl) to avoid portability problems of locking into a specific platform. There are benefits to building on existing technology [41]. The aim is to allow the Notebook to be installable on the development platform and to allow others to modify the infrastructure as they see fit.

Tools must report visible results with industrial systems. Individual aspects of the Notebook have been applied in the SQL/DS, RevEngE, and Software Bookshelf projects with IBM Toronto [153, 86, 150, 39]. The projects tested a methodology called structural redocumentation, evaluated the capabilities of scripting to codify and automate analyses, weighed the benefits of integration technologies, and assessed the use of web technologies to deliver constructive views of software structure.

5.3 Chapter summary

Notebook adoption requires fitting in with existing processes and meeting end user needs. This chapter described a goal-directed framework to match Notebook capabilities to end user analysis questions. The following chapter summarizes the lessons learned in several studies that test these individual capabilities.

Chapter 6

Studies

> Good judgement comes from experience, and experience comes from bad judgement.
> —Fred Brooks

This chapter outlines studies in several software evolution, program understanding, and tool integration projects with which I have been directly involved. Lessons learned of what does and does not work are summarized.

6.1 Effectiveness

Program understanding tools need to be effective to be used. To evaluate effectiveness, empirical studies are needed (especially with actual practioners) [44, 146, 132]. Although many program understanding tools already exist, the majority of these tools have not been adopted by industrial maintainers. In general, the evaluation of many tools has been lacking. The value of many research ideas has not been adequately substantiated through empirical studies. Also, by not evaluating existing tools and approaches, we are unable to discover which approaches are effective and which ones are not.

Various types of studies are possible [64, 156]. A case study is where researchers actively interact with practitioners in a project using the tool and collect observations over time. The costs are relatively low in collecting the observations, but the lessons learned may be difficult to generalize. A controlled experiment supports multiple, consistent sessions of observation to provide statistical validity of the results. A survey tries to capture what is happening broadly over large groups of projects.

6.2 SQL/DS

In this project, the SQL/DS (Structured Query Language/Data System) product was used as the testbed for evaluating the structural redocumentation approach supported by Rigi [153]. SQL/DS was a relational database management system, developed by IBM in the 1970s. SQL/DS was successful and mature, but was also evolving to run on new environments and support a growing functionality. For these reasons, it was deemed to be an excellent example of a legacy system. The SQL/DS system consists of over one thousand compilable units containing roughly three million lines of source code written in PL/AS (Programming Language/Advanced Systems), an internal IBM language. PL/AS is similar to PL/I, but has extensions to support IBM System/370 Assembler statements. The case study involved interactions between the researchers and the managers and developers of the SQL/DS system. The researchers presented views of subsystem abstractions.

6.2.1 Lessons learned

In producing subsystem abstractions over the SQL/DS code, many operations were repetitive and tedious (especially with so many artifacts). Drawing operations were optimized, but this was not enough. It was necessary to automate some tasks while still leaving the analyst in charge. Consequently, the Rigi editor was

made end user programmable [87] through an extensible scripting language based on Tcl [91]. Scripting was a useful way to integrate diverse capabilities such as graph layout, graph decomposition, and complexity metrics. The Notebook approach, however, does not contend that the end user should write scripts. The trend was that Rigi was becoming more of a reusable component or "engine" that could be adapted for various visualization purposes [137] and possibly inserted into various environments. For example, Rigi serves as a visualization component in the Dali workbench for reconstructing architectural documentation [62].

In dealing with SQL/DS, it was critical to determine the right abstractions for the audience. For example, call graphs were meaningless to the managers we encountered. More domain knowledge was needed. Subsystem abstractions based on naming conventions were more effective. Nevertheless, one developer basically used Rigi as a drawing program to reproduce an architectural diagram in the documentation (an unexpected result). Another need was to respect the developers and not complain, at least openly, that their system is poorly structured or in a state of crisis.

Successes included:

- the ability to handle multi-million line systems (by throwing out intraprocedural information),
- meaningful subsystem abstractions based on naming conventions,
- developer interactions with views to verify their mental models and documentation,
- a programmable graph editor based on Tcl to automate tedious operations,
- integration with external graph layout algorithms,
- verifying the importance of domain knowledge,
- verifying the need for management and developer perspectives of software,
- verifying a redocumentation methodology, and

- verifying the usefulness of high-level software structure for understanding.

Failures included:

- inability to truly penetrate the deep complexity of SQL/DS.

6.3 RevEngE

The goal of the RevEngE project was to develop an advanced, integrated toolset for program understanding [86, 150] with a diverse complement of analysis capabilities to help improve software quality. The prototype implementation involved integrating Rigi and the commercial Software Refinery product by Reasoning Systems (REFINE) [104]. Rigi provided high-level views of software abstractions. REFINE supported detailed code analyses and transformations. The focus was on using the deeper analysis capabilities of REFINE to provide useful input into the structural redocumentation process. In particular, clustering strategies based on data bindings were tested. Data integration was enabled by the Telos software repository [85], which included a common, global schema and was implemented using the ObjectStore persistent object base by Object Design [88]. Control integration was enabled through a software message bus.

6.3.1 Lessons learned

The benefit of combining Rigi and REFINE may have been outweighed by the brittleness of the integration. Advanced integration was difficult to implement and keep working due to tool and version changes. The global schema became large and unwieldy to store all the types of artifacts of the REFINE abstract syntax tree. To help address the scalability problem, only information that would be shared between Rigi and REFINE was modeled in the schema. That is, the tools each have their own local workspaces to store information. 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.

Telos supported a dynamically evolvable schema. However, even with a fixed schema, REFINE would renumber the identification numbers of generated artifacts when rerunning analyses, making it difficult to update the results of those analyses incrementally on the Rigi side. Analyses generally had to be done from scratch to ensure synchronicity, but this tended to be time consuming and definitely non-interactive. REFINE had problems dealing with partial systems and needed full access to the entire source code to resolve all references. In general, it was difficult to have a smooth integration because of subtle tool limitations.

Successes included:

- access to detailed analyses from REFINE to Rigi,
- verifying the need for lightweight technologies, and
- verifying the importance of structuring mechanisms in software repositories.

Failures and risks included:

- difficulties in adapting proprietary technologies,
- brittleness of the integration mechanisms,
- scalability problems for deep analyses of large software systems,
- tool limitations hampered usefulness of the integrated environment,
- message bus never used to truly coordinate tool actions, and
- relatively low level of industrial participation.

6.4 Software Bookshelf

The goal of the Software Bookshelf project was to provide an easily accessible collection of comprehensive documentation about a legacy system [39]. This project was part of a larger effort called the Consortium for Software Engineering Research (CSER) [27], which involved several universities and industrial partners studying aspects of migrating legacy systems to modern platforms. In particular, the Software Bookshelf was used to consolidate key information about the architecture and central data structures of a compiler backend while one of its modules was rewritten from PL/I to C++.

The prototype implementation involved using standard web technologies to enhance documentation delivery and object-oriented repository technologies to store the diverse range of information artifacts. One goal was to use a web browser to provide remote access to centrally maintained documentation and help ease adoption. As part of the Bookshelf project, we conducted an integration experiment with Rigi to help understand the technology.

In particular, there was a need for live, specialized, computed views as web content. It was not possible to prefabricate all the views one might want and store them directly as static HyperText Markup Language (HTML) pages or graphic images.

There are a number of server-side solutions for creating dynamic pages. Web authors often use Common Gateway Interface (CGI) scripts or web server modules to construct content dynamically. Also, a meta-language of preprocessing and transformation directives can extend HTML to provide more dynamic pages. Server Side Includes (SSI) are a primitive form of such a meta-language.

Besides the server-side approaches, there are also client-side strategies that operate from the web browser, including helper applications, plug-ins, Java applets, and JavaScript handlers. Helpers are independent programs that can provide sophisticated views. Plug-ins are software components that conform to an interface for communicating with the browser and drawing into its windows. Java applets are

platform-neutral programs fetched over the network and run on a Java-enabled browser. JavaScript handlers are scripts that are triggered on certain events, such as the clicking of a link. These scripts are embedded in HTML pages and are interpreted by JavaScript-enabled browsers. All of these strategies are flexible for presenting interactive views of bookshelf data. However, some strategies may be easier to exploit than others.

To exploit its reverse engineering and software analysis capabilities, the Rigi tool was integrated into the bookshelf environment. The basic idea was to allow Rigi to render views constructively via scripts, based on information stored in the repository. This was an advance over approaches that only retrieve static, ready-made images. By building views on-the-fly, the user could filter immaterial artifacts, emphasize relevant components, and customize the views to the analysis task at hand. The views remained live and manipulable. Also, changes to the software being reengineered were more easily reflected without requiring batch updates to statically stored images.

The Rigi system could be tightly integrated with the bookshelf environment by rewriting the user interface in Java. However, the programmability of Rigi allowed for a loose integration strategy that required no changes to the editor. Rigi was connected to the bookshelf environment using a CGI script (called `rigiserver`) and a helper application (called `rigiclient`), both written in Perl [147]. Access to Rigi and its constructive views from the bookshelf web browser had to be as simple as following a URL. Consequently, we specified a special form of URL that invokes the CGI script with a sequence of keyword/value pairs. These pairs specified required parameters, including: project name, domain model, database, version, user ID, session data, display host, computational host, requested view, and context. The CGI script parsed the pairs and sent the parameters to the helper application as a custom MIME type. The helper converted the parameters into Tcl and generated a custom configuration file, as well as a startup script that was used to launch Rigi to produce the view. If Rigi was already running, then the helper conveyed the requested view in a file that Rigi periodically polled. See Appendix B.

The time needed to convey the request to Rigi was short compared to the time needed to compute and present the requested view in a window. Since constructive views were computed by another process, possibly on another machine, there were no memory problems or security limitations incurred by trying to render these views within the browser using plug-in modules or Java applets. This integration strategy was general and could be readily adapted for any standalone analysis tool that was programmable and/or provided a comprehensive API (application program interface). It was especially important, here, to focus on small, well-defined tasks for automation.

6.4.1 Lessons learned

As an observation, web browsers may be best suited for browsing information and lightweight interaction. Significant interactions may require a traditional graphical user interface. For example, the bookshelf content itself is not editable to the end user and essentially provides only one perspective of the software. This is unlike the Notebook approach of allowing end users to explore and manipulate retrieved information and add their own annotations and perspectives.

Successes included:

- leverage of standard web technologies,
- consolidated documentation,
- usefulness of computed, constructive views,
- separation of bookshelf roles into builder, librarian, and reader, and
- significant interest from other development groups.

Failures and risks included:

- server side of the bookshelf environment was difficult to install,

- no documented process for generating or regenerating a bookshelf,
- time-consuming effort to produce bookshelf content,
- no integration with code development tools,
- limited actual adoption of the bookshelf, except by novices, and
- transliteration of PL/I code to C++ did not require deep knowledge as provided by the bookshelf.

6.5 User studies

A user experiment involving 30 participants was conducted to compare the effectiveness of three tools on a selection of program understanding tasks [125]. The tools were Rigi, SHriMP, and SNiFF+ (as shown in Figures 6.1, 6.2, and 6.3). The Simple Hierarchical Multi-Perspective (SHriMP) tool [123] displayed software architectural diagrams using nested graphs. This interface embedded source code inside the graph nodes and integrated a hypertext metaphor for following low-level dependencies with animated panning, zooming, and fisheye-view actions for viewing high-level structures. The SNiFF+ system [128] was a commercial, integrated development environment for C and C++ that provided source code browsing and cross referencing features.

One goal of the experiment was to gain some insight on the interplay of the preferred comprehension strategies of the participants and the implicit strategies embodied by the tools. In short, we conjectured that program understanding tools needed to support a combination of comprehension strategies, provide ways to easily enter and effortlessly switch between strategies while solving a task, and reduce the level of cognitive overhead as the program is explored.

A two-hour session with each of the participants contained six time-limited phases: orientation (5 min), training tasks (20 min), practice tasks (20 min), formal tasks (50 min), post-study questionnaire (15 min), and post-study interview and debriefing

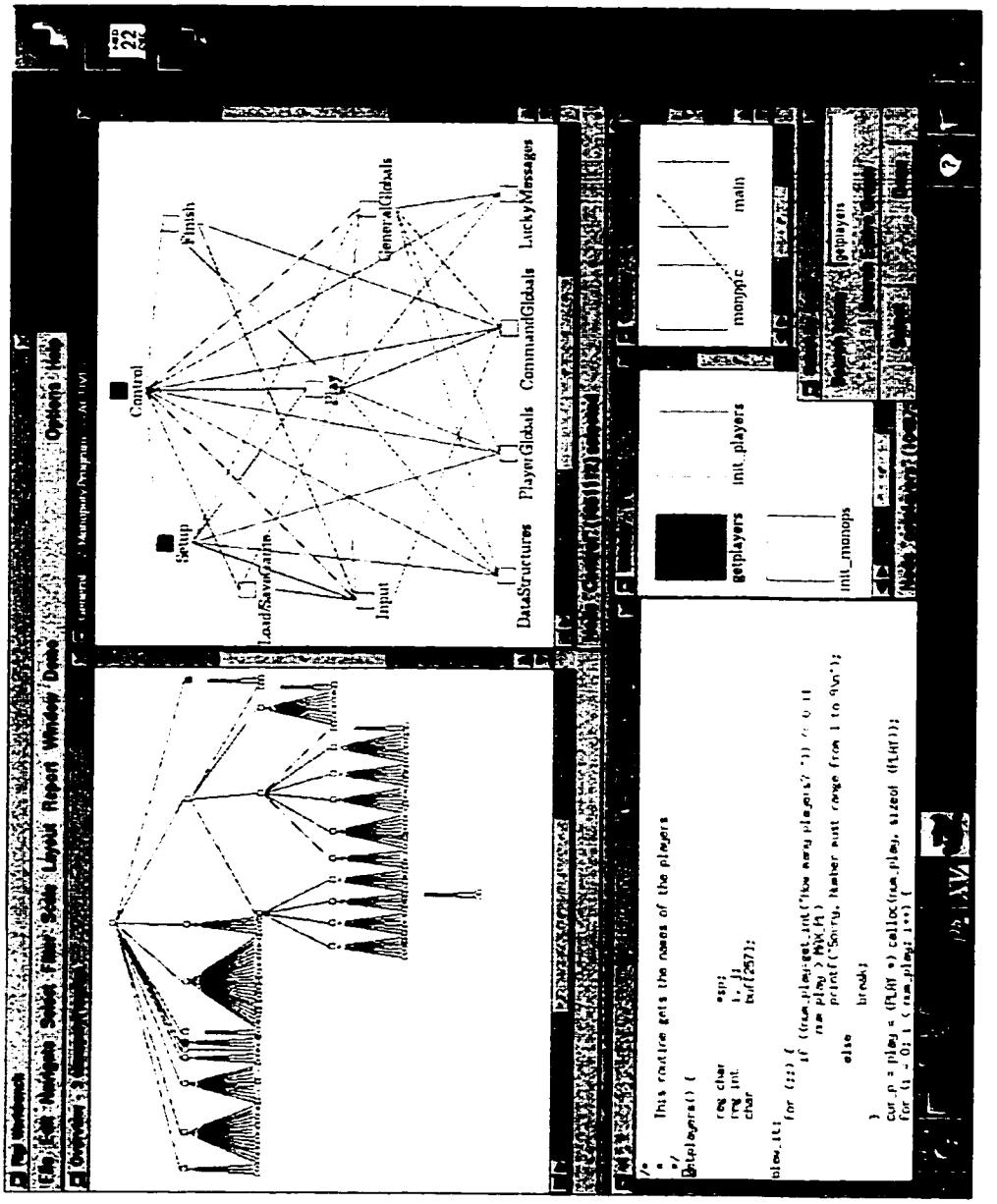

Figure 6.1: Multiple windows of the Rigi tool

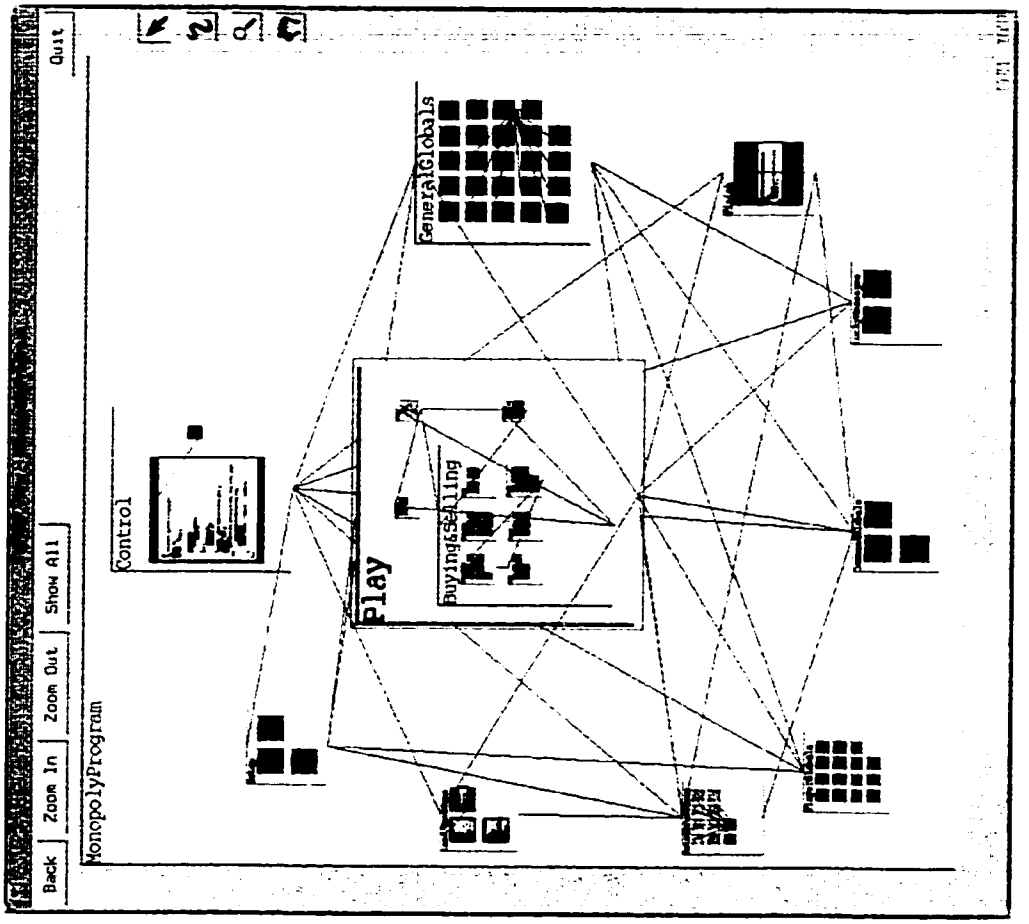

Figure 6.2: Nested graph within the SHriMP tool

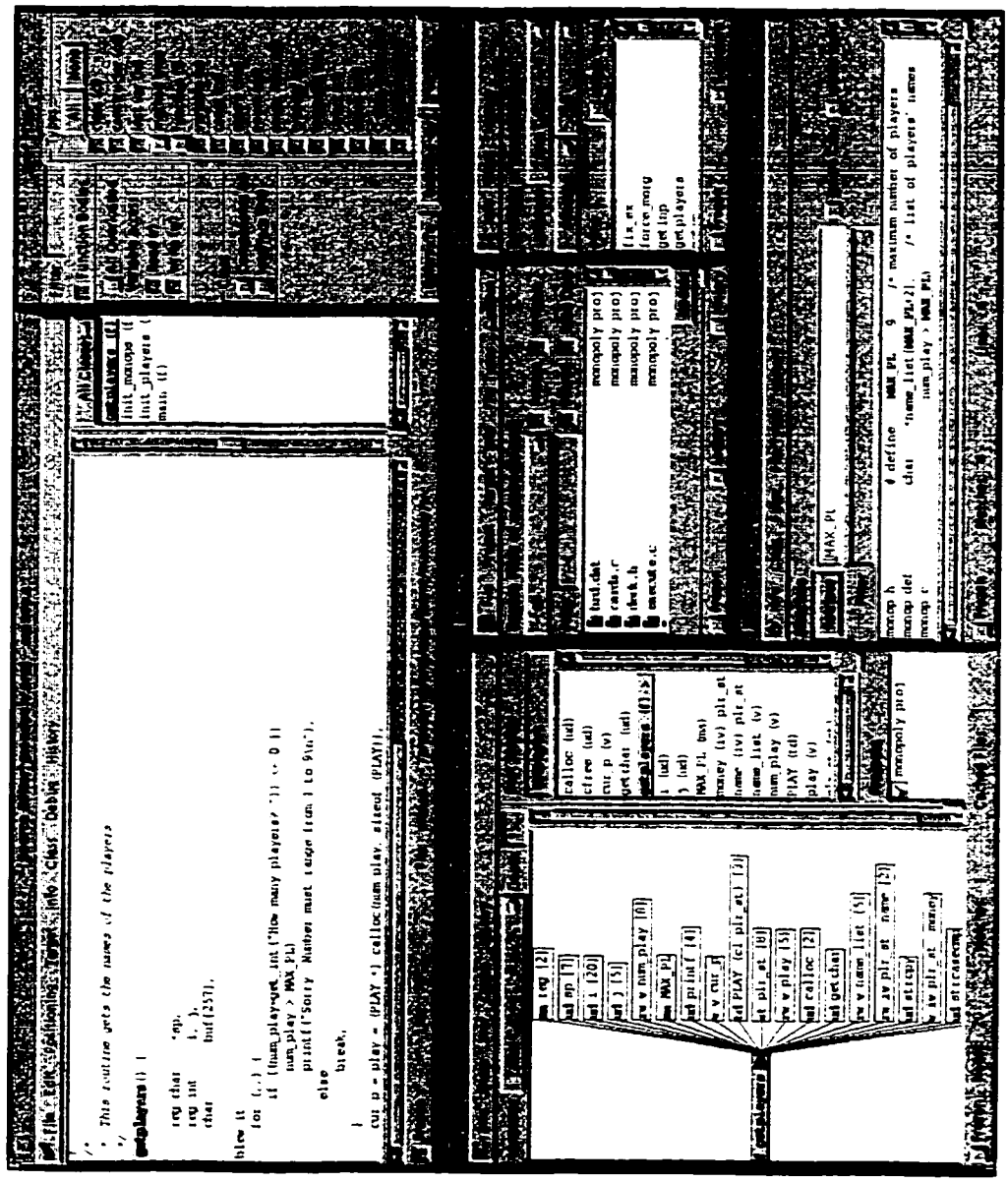

Figure 6.3: The SNiFF+ development environment

(10 min). These durations were based on experience in conducting an earlier pilot study [124]. A detailed experimenter's handbook was written for each tool to provide some consistency and control over the running of each experimental session. General instructions (common to all tools) outlined the structure of the experiment, the rules of conduct, and various procedures to be followed by the experimenter. Tool-specific descriptions contained detailed instructions for each of the experimental phases.

Factors that would affect the performance of the participants and their choice of comprehension strategy included: the test program, task complexity, and user expertise. The chosen program for the formal tasks was a text-based Monopoly game program, written in C. The program contained 1700 lines of code in 17 files, with only sparse comments. The control flow was fairly complex, due to some gotos and a table of function pointers. A common subsystem hierarchy was prepared for the Rigi and SHriMP tools. The SNiFF+ tool did not support subsystem abstraction capabilities.

The formal tasks focused on higher-level program understanding. Some tasks required the participant to understand part of the program to answer a question about its functionality. For example, one task asked the participant to determine whether the game implemented a computer player. Other tasks required a deeper understanding, asking the participant to describe how to change the program to implement a new feature. For example, one task asked the participant to describe how to implement a rule so that players in jail (not visiting) lose the ability to collect rent.

The level of expertise and skill affects a user's performance by contributing significantly to understanding a program or learning a tool's interface. A pre-study questionnaire asked about C programming experience, maintenance experience, number of years as a programmer, experience writing games, etc. Domain knowledge about the Monopoly board game could be an asset by providing useful preconceptions. To strive for consistency across participants, we set up a Monopoly board beside each participant and, if needed, explained the rules of the game. We encouraged them to review the rules and use the board throughout the formal

tasks, although few actually did.

6.5.1 Results

For all three tools, there were times when the users' preferred comprehension strategies were not adequately supported. For example, SNiFF+ was more suited to bottom-up approaches; few facilities were available for showing higher-level information about the program structure. In Rigi, many users had problems trying to read code systematically and follow the control flow. In SHriMP, the biggest problem was the lack of a searching tool, which was often the desired approach for finding hints to verify hypotheses.

The graphical subsystem hierarchy presented by Rigi and SHriMP was effective at conveying a mental map of the program. Many users mentioned that the presented structure was logical and helped them to understand the program. However, we also suspect that by imposing a structure on the Monopoly program, the users perceived it as being more modular than it actually was. The names given to the subsystem nodes were especially important.

Since the users did not actually need to make the changes, they took a nonchalant approach to the maintenance tasks and made educated guesses at the solutions. Indeed, there was no apparent difference in correctness across the tools.

In Rigi, the lack of a searching tool to find text strings in the source code definitely hindered the users. Some users mistakenly thought they were searching for strings in the code rather than searching for node labels in the graph. However, the ability to search on node labels was very useful. Also, navigating from a text editor view of the source code to the graphical view of the subsystem hierarchy was not well supported.

Both Rigi and SNiFF+ were capable of representing larger software systems. However, the multiple window approach used by these tools often disoriented the users. The users were faced with the difficult task of accurately conceptualizing

and integrating the implicit relationships among the contents of individual windows.

Rigi provided the ability to filter irrelevant information. These filters were used very effectively and increased the scalability of the tool. Node labels can be filtered in an overview windows. This reduced some visual clutter, but the labels of important subsystem nodes were also filtered. Consequently, the users had to search for nodes by name to highlight the matching nodes in the overview, or they had to turn off the node label filter for a selected set of nodes. Some users found this awkward.

6.5.2 Lessons learned

There were many practical difficulties in running a study of this size. Although we did not entirely prevent experimental biases from arising, we tried to minimize them. In carrying out the study, we used five experimenters. Small inconsistencies among the sessions run by different experimenters affected the observations. There were a few instances where an experimenter forgot to show an essential feature of a tool, thereby significantly altering the comprehension strategies used. Training the experimenters and following the handbooks carefully helped to reduce these problems. The use of the Rigi and SHriMP tool designers as experimenters introduced a bias. For example, one SHriMP user knew the SHriMP designer and worked more intensely with the tool than usual. To reduce this bias, we rotated the experimenters among two or three tools, videotaped the formal tasks for most users, and tried not to reveal the tool designer.

Another important issue is external validity. That is, do the experimental results and observations generalize? There are several problems that make it difficult to generalize the reported findings to industrial settings. The study was conducted with student programmers. The software to be understood was fairly small. The lengthy preparation of subsystem abstraction hierarchies for Rigi and SHriMP (by a Rigi expert) may not even happen in a time-constrained industrial setting. Excellent effectiveness may be no indication of actual use in practice. That is, tools must

be used so that their effectiveness actually matters [56]. Tool adoption and usage issues cannot be ignored.

Successes included:

- useful, specific feedback for tool developers,
- verifying need to support switching between program comprehension strategies,
- identifying awkward points in tool user interfaces,
- verifying importance of code searching capabilities,
- verifying usefulness of subsystem abstractions, and
- verifying usefulness of visual filters.

Failures and risks included:

- some experimental biases, and
- some external validity concerns.

6.6 Validation

While the Notebook as a whole has not been completely implemented, various aspects of the Notebook have been implemented (mostly on top of Rigi [152]) and validated in the described studies. Table 6.1 indicates Notebook features and how they currently satisfy the requirements of Chapter 2. Table 6.2 indicates the status of satisfying the data requirements of Chapter 3 using RSF [152]. The letters in parentheses indicate whether the requirement is met mostly (M) or to a small extent (S).

Table 6.1: Validation of general requirements

	Requirement	Features
1	Handle complexity	subsystem abstraction [153] (M)
2	Use design abstractions	subsystem abstraction [153] (S)
3	Capture informal information	annotations [152] (S)
4	Exploit domain expertise	conceptual modeling [133] (M), consult domain experts [153] (M)
5	Capture development information	subsystem abstraction [153] (S)
6	Summarize software structures	subsystem abstraction, analysis, and views [153] (M)
7	Support different users	tool customization, configuration, and extensibility [133] (S)
8	Provide right abstractions	subsystem abstraction [153] (M)
9	Deliver diverse analyses	integration of external capabilities [137] (M)
10	Support incremental analyses	constructive analyses and views (M)
11	Support continuous understanding	constructive analyses and views (M)
12	Address tool adoption issues	goal-oriented framework [151] (S), packaging [152] (S)
13	Provide multiple perspectives	subsystem views [153] (M)
14	Organize software abstractions	subsystem abstraction and analysis [153] (M)
15	Provide interactive views	interactive user interface [153] (M)
16	Rationalize the process	scriptable tool [137] (S), redocumentation methodology [153] (S)
17	Organize diverse artifacts	RSF tuples [152] (S)
18	Provide hypermedia documentation	subsystem views [153] (M)
19	Address usability problems	user experiments [124, 125] (S)
20	Integrate graphical and textual views	subsystem views, source code, and annotations (M)
21	Answer genuine questions	subsystem analysis [153] (S)
22	Integrate static and dynamic views	
23	Evaluate tool effectiveness	user experiments [124, 125] (M), case studies [153] (M)

Table 6.2: Validation of data requirements

	Data Requirement	Satisfaction
1	Evolvable schemas	new types dynamically created (S)
2	Graph model	nodes, arcs, and attributes (M)
3	Abstraction and constraints	subsystem abstraction (M)
4	Multiple abstraction levels	subsystem hierarchy graph (M)
5	Mapping between levels	subsystem hierarchy graph (M)
6	Composable, modular schemas	schemas defined in files (S)
7	Composable fact bases	sorted, canonical form (M)
8	Version control	
9	Scalable	limited by main memory (S)
10	Query mechanism	script language and graph traversal (M)
11	Schema introspection	script language (M)
12	Human readable format	plain editable text (M)
13	Name, location, order	parser dependent (S)

6.7 Chapter summary

Case studies and controlled user experiments served to provide many lessons about the usage and effectiveness of reverse engineering tools. The lessons learned, successes, failures, and risks gathered in this chapter help to refine tool requirements. The experiences also served to validate the Notebook against the requirements. The next chapter considers work related to the Notebook.

Chapter 7

Related Work

This chapter describes other work that is relevant to the Notebook as a whole. There are many other tools, methods, infrastructures, techniques, and processes in research and industry that are relevant to the Notebook; here we simply outline some of the ones that have influenced us the most.

7.1 Software engineering support

There have been several approaches to the general understanding problem, including documentation [49, 114], version control [110, 131, 14], development environments [105, 8, 120], and program editors [121]. Traditional software documentation becomes rapidly out-of-date as the code changes. Furthermore, this documentation is difficult to maintain and is often impossible to customize for different developer needs. Version control systems help to store and control revisions to software but do little to improve the understanding of that software. Typical development environments and program editors improve on-demand understanding "in-the-small" but do not adequately address the higher-level and more long-term requirements of understanding "in-the-large." Users of program generators focus on drafting formal specifications from which code can be generated auto-

matically, with the intent that the specifications be easier to understand. Except for certain well-understood domains such as compiler construction [68, 94, 3], these approaches have been difficult to adopt broadly. Also, they do not address the large body of existing code not developed through generation.

7.2 Software Refinery

The Software Refinery by Reasoning Systems [104] is a toolkit for analyzing source code. The three main components of the Software Refinery are Dialect (a customizable parsing/unparsing subsystem), REFINE (a Lisp-like language that traverses and transforms the code as represented in an object-oriented database), and Intervista (a graphical user interface). The Software Refinery is especially flexible, allowing users to develop their own parsers, domain models, analysis techniques, and visualizations. Domain models have a single inheritance class hierarchy (a hierarchy separate from the abstract syntax tree). There are prepackaged versions of the toolkit for the major legacy languages, such as C, COBOL, and Fortran. With this power, however, comes a steep learning curve. Also, the toolkit has significant hardware requirements.

7.3 Electronic books

A Mathematica notebook, by Wolfram Research, is a complete interactive document which combines text, tables, graphics, calculations and other elements [74]. Notebooks are automatically organized in a hierarchy of cells similar to an collapsible outline. Hyperlinks are used to jump within a notebook or between notebooks. Buttons with attachable actions can be inserted. Each cell can be assigned a style from a style sheet. Notebooks are automatically retargeted for on-screen or printed (publication quality) presentation. Mathematica notebooks can be built up using explicit scripted commands as well as interactively.

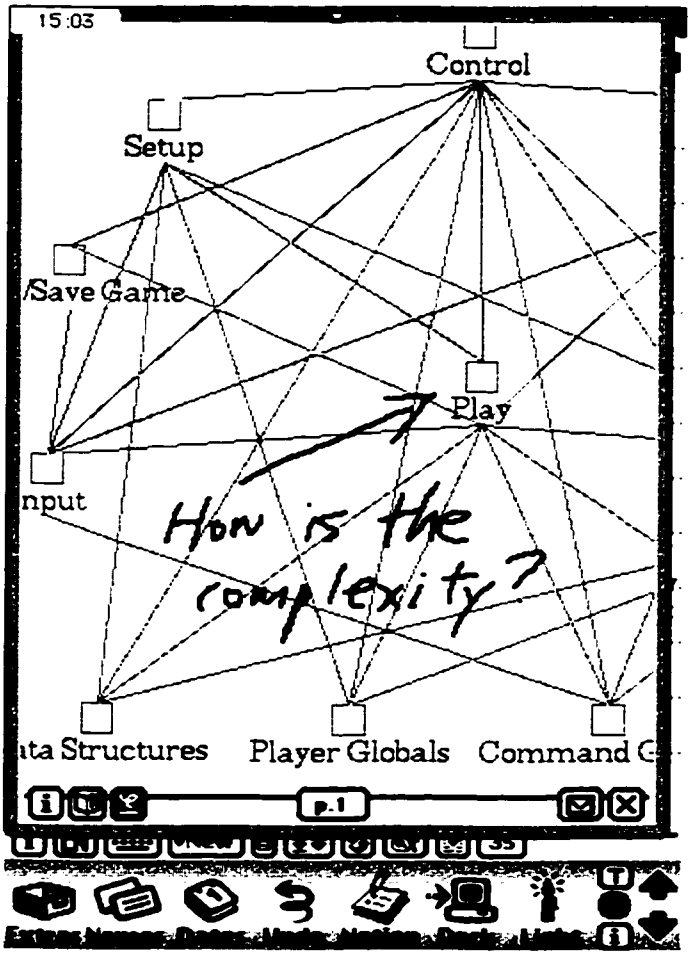

Figure 7.1: Personal annotations over a reverse engineering analysis

The Newton Messagepad, by Apple, is a personal digital appliance for organizing memos, appointments, and names, with support for handwriting recognition to enter information. The Messagepad also serves as a reading appliance for electronic books that can be created by the user from typical wordprocessing document and image formats. Newton books can also contain free-form, personal notes by writing with the stylus on an annotation layer of the book, as shown in Figure 7.1. The ability to make such unstructured ink marks over documents is important. The marks have a rich semantic meaning for people and enhance memory [116].

When a document is converted into Adobe Portable Document Format (PDF), a

file is obtained that can be read, navigated, searched, and printed by anyone with a PDF viewer. Since PDF viewers exist on a wide variety of platforms, PDF files are essentially universal. The look of a PDF file online is identical to a printout of the original document. That is, PDF is useful for documents that would otherwise normally be printed. Other useful navigation and customization capabilities of PDF are the support of thumbnails, bookmarks, articles, links, annotations, notes, and interactive forms.

7.4 Personalized information spaces

A number of systems use a spatial filing metaphor to organize icons on a computer screen [7, 36, 30]. Typically, these icons represent files, folders, documents, and devices. These systems include the Macintosh Finder, mind maps, and Web Squirrel farms [32]. The subsystem abstraction process is a similar technique for organizing and clustering nodes in a graph. One significant problem is scaling to huge numbers of objects at the initial flat graph where all nodes appear. Alternatives such as Lifestreams [79] use a temporal metaphor to order objects chronologically by their time of authorship. Using time as another dimension to structure software systems may help highlight change patterns during evolution.

7.5 Public Common Tool Interface (PCTE)

PCTE [20] was developed as a comprehensive way to integrate software technologies produced as part of a collaborative research and development program. Two major goals were portability and compatibility. The design is centered around providing basic integration mechanisms and separating mechanisms and policies. Basic mechanisms include program execution, communication to an Object Management System (OMS), interprocess communication, recovery and synchronization, concurrency and distribution, and a command language. Objects in the OMS are navigated by traversing relationships. A Schema Definition Set (SDS) defines

object and relationship types, which can be specific to users and tools or common to a community.

7.6 Compound document architectures

OpenDoc is an architecture for producing compound documents [5, 6]. A compound document is based on a container within which content can be embedded. See Figure 7.2 for a document containing a text box, an audio recording item, and an outline containing diagrams. The editors or viewers of the embedded content are called parts. The usage metaphor is document centric, so that documents and content are always instantiated from stationery documents, not by applications. OpenDoc supports dynamic live links and shared information between parts. The user interface of a part is separated from its functionality through scripting and events. The architecture essentially supports the ability to select and integrate lightweight, component software parts to build a compound document. This ability to pick and choose avoids monolithic bloated software applications.

7.7 Code base management systems

Code base management systems (CBMSs) are specialized data base management systems for managing software artifacts. They assist in the maintenance, analysis, and transformation of large sets of software components. Information is recorded about interdependencies, personnel assignments, test suites, analyses, client feedback, and project scheduling. A typical CBMS centers around a query engine, analyzer, transformer, and user interface (similar to the Software Refinery [104]). Consistency is important to maintain relationships between entities whenever possible. This includes triggering the reapplication of analyses, warning of inconsistencies, and providing version access control.

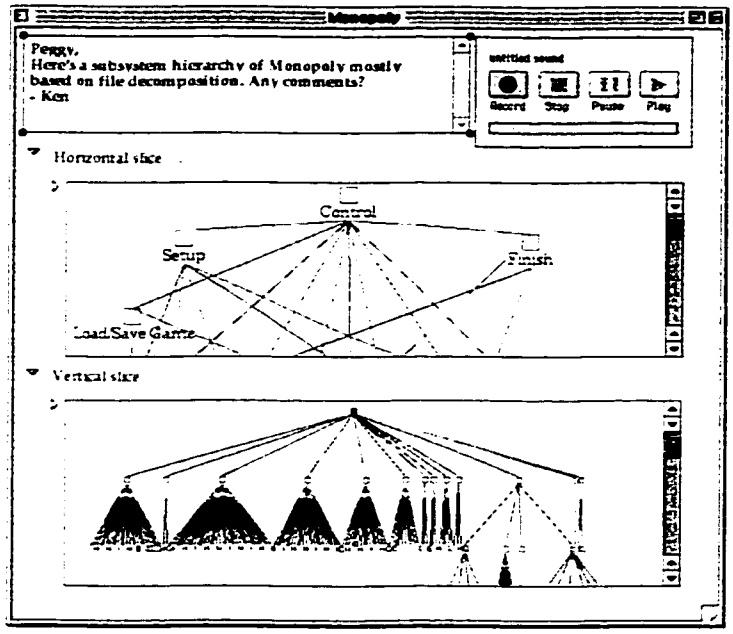

Figure 7.2: An OpenDoc document with reverse engineering content

7.8 Consistency architectures

Design/code consistency architectures involve strategies to maintain synchronization between software design and code. Literate programming combines documentation with code and allows the developer to arrange the parts of a program in any order and extract documentation and code from the same source file [25]. "Roundtripping" involves the capability to generate skeleton code from designs and, after code updates, automatically reconcile these updates by modifying the design to match [138]. The Rosetta system has design and code as two independent but interrelated descriptions of the system. Programmers can use their usual tools for creating code. Automatic consistency checks are used to verify that the design and code match [113].

7.9 Chapter summary

This chapter described several software systems that have been a major influence on the Notebook approach. Next, several recommendations that result from this dissertation are outlined.

Chapter 8

Recommendations

This chapter outlines several recommendations for reverse engineering tool development and research that result from this dissertation. Many of the requirements of Chapter 2 and the desirable aspects of Chapter 3 could also be viewed as useful recommendations. Here, approaches that were particularly beneficial for the Notebook are recommended.

Tool researchers must be aware of the needs of practitioners and the barriers to technology adoption. It is not enough to devise a new technique or paradigm and simply expect practitioners to pick it up and adopt it. Software understanding techniques and tools need to be packaged effectively and made compatible with existing processes, users, and tools. These tools need to be used to be effective.

RECOMMENDATION 1

Make adoption issues an integral part of reverse engineering tool research and design.

Case studies, user experiments, and surveys are all necessary to assess the effectiveness of a reverse engineering tool. In case studies, it is especially important to involve practitioners and test approaches on industrial-scale software systems. In user experiments, not only do potential biases need to be controlled, a discussion of what can be generalized should also be discussed. The tools need to be effective to be used.

RECOMMENDATION 2
Conduct and report empirical studies to evaluate the effectiveness of reverse engineering tools.

Limitations and downsides of a reverse engineering approach need to be discovered and frankly reported (even if discouraging). In times of technological churn, we also need to step back and see what works and what does not.

RECOMMENDATION 3
Summarize and distill lessons learned from reverse engineering experience to derive requirements for the next generation of tools.

For reverse engineering analyses, such as structural redocumentation and subsystem abstraction, pervasive scripting is a very effective way to codify decomposition and layout algorithms, record graph manipulations, automate tasks, and integrate external capabilities. Also, by making tools programmable, they can more easily be incorporated into other toolsets, thus easing an adoption issue of tool compatibility.

RECOMMENDATION 4
Incorporate concepts of pervasive scripting into reverse engineering tool architectures, including scriptability, recordability, attachability, constructability, and undoability.

Support is needed to evolve a base of software understanding as the code evolves. This understanding may include abstractions of software structure or architecture, which are often lost or obsolete in traditional documentation. By maintaining continuous understanding and better managing the understanding gap, future analyses can be eased. Studies of practical documentation architectures are important.

RECOMMENDATION 5
Investigate support for continuous software understanding, including mechanisms to tie code to design abstractions such as scripts and constructive views.

Reverse engineering tool researchers need to be aware of the context within which

their techniques can be applied. Consequently, researchers need to be versatile and obtain a broad grasp of computing technology, engineering processes, and business goals. Social factors and psychology are also important when dealing with information systems technology.

RECOMMENDATION 6

Be aware of all stakeholders for a reverse engineering approach, including the need to support multiple user perspectives and define specific user roles.

Chapter 9

Conclusions

There are no endings, only beginnings.
—Anon

9.1 Summary

Major themes and goals of the dissertation included:

- enabling continuous software understanding;
- enhancing compatibility with existing tools, users, and processes by emphasizing integration, process, and adoption issues; and
- learning from the successes and failures of studies using tool integration and reverse engineering technologies.

Most program understanding activities are short-term, directed to answering specific questions. As the software evolves, it becomes more complex, making it increasingly difficult to produce the required answers. For poorly documented

systems, this may be impossible without undertaking significant work to rediscover design information and reproduce architectural abstractions. This information needs to be maintained. In essence, the understanding of software should be continuous and not reach such a critical point. Continuous understanding is an approach to support continued future change and evolution. The Notebook concept, infrastructure, and process contribute toward enabling continuous understanding. By recording analyses such as subsystem abstraction using scripts and carrying these analyses and views to other baselines of code, understanding is maintained. By maintaining this body of understanding, documentation about the structure of a software system is improved.

Practitioners need tools and techniques that are compatible with existing tools, users, and processes. Addressing this concern is a high priority for the Notebook approach. Data, control, and presentation integration techniques are explored for the Notebook infrastructure that support the sharing of information with other tools, the control and coordination of the Notebook and tools, and the managed presentation of analysis information. User roles and methodology for the Notebook clearly define the increasingly project and task specific responsibilities and perspectives of the Notebook builder, mediator, and end user. A goal-directed framework provides a context within which the Notebook approach can be inserted into a software change process. Several user needs with respect to adoption are also addressed.

The Notebook leverages the lessons learned, successes, and failures of several studies involving tool integration and reverse engineering technologies. These studies include both industrial case studies and controlled user experiments. The lessons serve to refine the Notebook approach, including: the infrastructure technologies centered on pervasive scripting to record, codify, automate, and integrate analyses; the methodology involving structural redocumentation to present personalized views of software architecture; the use of software subsystem abstractions to represent domain concepts; the importance of adoption issues to ease compatibility with other tools, users, and processes; and the construction and evolution of a base of understanding to enable continuous software understanding.

9.2 Contributions

The dissertation identifies requirements of tool support for software understanding (Chapter 2). The requirements characterize the fundamental needs to be addressed in the design of reverse engineering tools, such as the Notebook. By distilling observations of reverse engineering tool usage in various studies, the requirements aim to improve the design, capabilities, adoption, and effectiveness of these tools.

The dissertation describes the Notebook concept and approach for continuous software understanding (Chapter 3). Aspects of the Notebook infrastructure are discussed, including data integration requirements, control integration via pervasive scripting, and presentation integration through the management of views. Pervasive scripting supports the ability to record, codify, and automate analyses, as well as allowing the capability to interoperate with other tools. Consequently, the Notebook forms a foundation for future tool development, as either the core of a toolset or a component of another environment.

The dissertation details process aspects of the Notebook approach, including user roles and a methodology for these roles (Chapter 4). Different roles involved in building, enhancing, and using the Notebook are defined. The builder evaluates tool technologies and constructs the Notebook infrastructure. The mediator enters information into and continuously evolves the Notebook content. The end user consults and annotates the Notebook to solve a specific understanding task.

The dissertation discusses adoption issues addressed by the Notebook (Chapter 5). Factors such as relative advantage, compatibility, complexity, trialability, and visibility are major design criteria for the Notebook. As well, a goal-directed framework is detailed for the insertion and placement of the Notebook approach into a software change process.

The dissertation summarizes lessons learned, successes, and failures of various studies that have validated individual aspects of the Notebook (Chapter 6). To ensure that reverse engineering tools are effective and used, empirical studies such as

case studies and user experiments are necessary. The observations serve to define better requirements to improve the next generation of reverse engineering tools to meet end user needs.

The dissertation outlines specific recommendations for reverse engineering tool research (Chapter 8). The strategy followed in designing the Notebook is also applicable to the design of other software engineering tools. Compatibility with existing tools, users, and processes should be a high priority for tool researchers.

9.3 Future work

The Notebook research itself undergoes continuous evolution. The evolution follows a spiral model [13], an evolutionary life-cycle model of engineering a product based on prototyping, risk management, and iteration. The product of each iteration through the model is not necessarily a finished piece of software. A product could be foundational requirements or unifying principles. Each iteration builds an evolutionary prototype that is intended to be a solid foundation for the next iteration. At present, requirements, concepts of operation, and design for the Notebook have been determined. Each iteration of the Notebook involves several stages, including studies with users, evaluating alternative technologies, considering compatibility constraints and adoption risks, validating the product, and planning for the next phase. Consequently, the next phase is one of more user observation, implementation, and evaluation of the Notebook approach.

Several tool requirements have not yet been addressed or adequately studied in the described Notebook approach. These include the potential reconstruction and use of forward engineering design notations, the storage and query mechanisms for information about software, the effectiveness of textual representations of software structure, the specific program comprehension tasks of end users, and integrating static and dynamic models of software structure and behavior. Also, long-term studies are needed to better evaluate the continuous program understanding concept for improving long-term software evolution. An avenue of future research

is answering the hypocrisy question of applying the Notebook approach to the Notebook infrastructure itself.

References

[1] http://www.aberdeen.com/. Aberdeen Group.

[2] James L. Adams. *Conceptual Blockbusting*. Addison-Wesley, Reading, Massachusetts, third edition, 1974.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

[4] Apple Computer, Inc. *Macintosh Human Interface Guidelines*, 1992.

[5] Apple Computer, Inc. *OpenDoc Human Interface Specification for the Macintosh Implementation*, November 1995.

[6] Apple Computer, Inc. *OpenDoc Programmer's Guide for the Mac OS*, 1995. Addison-Wesley.

[7] Deborah Barreau and Bonnie A. Nardi. Finding and reminding: File organization from the desktop. *ACM SIGCHI Bulletin*, 27(3):39–43, July 1995.

[8] David R. Barstow, Howard E. Shrobe, and Erik Sandewall. *Interactive Programming Environments*. McGraw-Hill, New York, New York, 1984.

[9] Victor Basili and Scott Green. Software process evolution at the SEL. *IEEE Software*, pages 58–66, July 1994.

[10] Penny Bauersfeld, John Bennett, and Gene Lynch, editors. *Proceedings of the ACM Conference on Human Factors in Computing Systems*, Monterey, California, May 1992. ACM Press.

[11] K. H. Bennett and E. J. Younger. Model-based tools to record program understanding. In Fadini and Rajlich [35], pages 87–95.

[12] Heinz-Dieter Böcker, Gerhard Fischer, and Helga Nieper. The enhancement of understanding through visual representations. In Marilyn Mantei and Peter Orbeton, editors, *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 44–50, Boston, Massachusetts, April 1986. ACM Press.

[13] B.W. Boehm. A spiral model for software development and enhancement. *IEEE Software*, pages 61–72, May 1988.

[14] Don Bolinger and Tan Bronson. *Applying RCS and SCCS*. O'Reilly & Associates, Sebastopol, California, 1995.

[15] Michael Brodie, John Mylopoulos, and Joachim W. Schmidt. *On Conceptual Modelling*. Springer-Verlag, New York, New York, 1984.

[16] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:543–554, 1983.

[17] Frederick P. Brooks Jr. *The Mythical Man-Month*. Addison-Wesley, Reading, Massachusetts, 1995. Anniversary Edition.

[18] Erich Buss and John Henshaw. Experiences in program understanding. Technical Report TR-74-105, IBM Canada Laboratory, Toronto, Ontario, Canada, July 1992.

[19] Lethbridge Timothy C. and Nicolas Anquetil. Architecture of a source code exploration tool: A software engineering case study. Technical Report 97-07, Department of Computer Science, University of Ottawa, 1997.

[20] Ian Campbell. PCTE proposal for a public common tool interface. In *Software Engineering Environments* [120].

[21] Minder Chen and Ronald J. Norman. A framework for integrated CASE. *IEEE Software*, pages 18–22, March 1992.

[22] P. P. Chen. The entity-relationship model: Toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[23] Richard A. Coll, Joan H. Coll, and Ganesh Thakur. Graphs and tables: A four-factor experiment. *Communications of the ACM*, 37(4):77–86, April 1994.

[24] J. Conklin. Hypertext: An introduction and survey. *IEEE Computer*, 20(9):17–41, September 1987.

[25] David Cordes and Marcus Brown. The literate-programming paradigm. *IEEE Computer*, pages 52–61, June 1991.

[26] James H. Cross II, Elliot J. Chikofsky, and Charles H. May Jr. Reverse engineering. *Advances in Computers*, 35:199–254, 1992.

[27] Consortium for Software Engineering Research. http://www.cser.ca/.

[28] Jacob L. Cybulski and Karl Reed. A hypertext based software-engineering environment. *IEEE Software*, pages 62–68, March 1992.

[29] Prem Devanbu. On a framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 21(12):1009–1010, December 1995.

[30] Susan T. Dumais and William P. Jones. A comparison of symbolic and spatial filing. In Lorraine Borman and Bill Curtis, editors, *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 127–130, San Francisco, California, April 1985. ACM Press.

[31] P. Eades and K. Zhang. *Software Visualization*, volume 7 of *Series on Software Engineering and Knowledge Engineering*. World Scientific, 1996.

[32] Eastgate Systems, Inc. http://www.eastgate.com/.

[33] Jürgen Ebert, Bernt Kullbach, and Andreas Winter. GraX: An interchange format for reengineering tools. In WCRE99 [149], pages 89–98.

[34] Stephen G. Eick and Joseph L. Steffen. Visualizing code profiling line oriented statistics. In VISUALIZATION92 [143], pages 210–217.

[35] Bruno Fadini and Vaclav Rajlich, editors. *Proceedings of the Second Workshop on Program Comprehension*, Capri, Italy, July 1993. IEEE Computer Society Press.

[36] Scott Fertig, Eric Freeman, and David Gelernter. Finding and reminding reconsidered. *ACM SIGCHI Bulletin*, 28(1):66–69, January 1996.

[37] N. V. Findler. *Associative Networks: Representation and Use of Knowledge by Computers*. Academic Press, New York, New York, 1979.

[38] Anthony Finkelstein, Michael Tauber, and Roland Traunmüller, editors. *Proceedings of the IFIP WG 8.1 Working Conference on Human Factors in Analysis and Design of Information Systems*, Schärding, Austria, June 1990. IFIP, North-Holland.

[39] P.J. Finnigan, R. Holt, I. Kalas, S. Kerr, Kostas Kontogiannis, Hausi A. Müller, John Mylopoulos, S.G. Perelgut, M. Stanley, and Kenny Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, 1997.

[40] Nigel T. Fletton and M. Munro. Redocumenting software systems using hypertext technology. In *Proceedings of the Conference on Software Maintenance*, pages 54–59. IEEE Computer Society Press, 1988.

[41] P. Freeman. Software engineering: Strategies for technology transfer. In *Proceedings of the International Computing Symposium on Applications Systems Development*, pages 333–351, Berichte, Germany, 1983. ACM Press.

[42] Pankaj K. Garg and Walt Scacchi. A hypertext system to manage software life-cycle documents. *IEEE Software*, pages 90–98, May 1990.

[43] Ann Gawman, M. Gentleman, Evelyn Kidd, Per-Ake Larson, and Jacob Slonim, editors. *Proceedings of the 1993 CAS Conference*, Toronto, Ontario, Canada, October 1993. IBM Centre for Advanced Studies.

[44] Robert L. Glass. The software-research crisis. *IEEE Software*, pages 42–47, November 1994.

[45] Robert L. Glass and Iris Vessey. Contemporary application-domain taxonomies. *IEEE Software*, pages 63–76, July 1995.

[46] Robert B. Grady. *Practical Software Metrics for Project Management and Process Improvement*. Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[47] Raymonde Guindon. Requirements and design of DesignVision, an object-oriented graphical interface to an intelligent software design assistant. In Bauersfeld et al. [10], pages 499–506.

[48] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[49] José R. Hilera, León A. González, José A. Gutiérrez, and J. M. Martinez. Software documentation as an engineering process. *ACM SIGSOFT Software Engineering Notes*, 23(5):61–64, September 1998.

[50] Michael Himsolt. Graph Modeling Language (GML) format, July 1997. http://www.fmi.uni-passau.de/Graphlet/GML/.

[51] Richard C. Holt. Tuple-Attribute (TA) language, November 1998. http://plg.uwaterloo.ca/ holt/papers/ta.html.

[52] Clifford C. Huff. Elements of a realistic CASE tool adoption budget. *Communications of the ACM*, 35(4):45–54, April 1992.

[53] Watts S. Humphrey. *Managing the Software Process*. Addison-Wesley, 1989.

[54] Watts S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley, 1995.

[55] *Hypertext '89 Proceedings*, Pittsburgh, Pennsylvania, November 1989. ACM Press.

[56] Juhani Iivari. Why are CASE tools not used? *Communications of the ACM*, 39(10):94–103, October 1996.

[57] Stan Jarzabek and Riri Huang. The case for user-centered CASE tools. *Communications of the ACM*, 41(8):93–99, August 1998.

[58] Brian Johnson. TreeViz: Treemap visualization of hierarchically structured information. In Bauersfeld et al. [10], pages 369–370.

[59] Brian Johnson and Ben Shneiderman. Tree-maps: A space-filling approach to the visualization of hierarchical information structures. In *Proceedings Visualization '91*, pages 284–291, San Diego, California, October 1991. IEEE Computer Society Press.

[60] J. Howard Johnson. Identifying redundancy in source code using fingerprints. In Gawman et al. [43], pages 171–183.

[61] Daniel S. Jordan, Daniel M. Russell, Anne-Marie S. Jensen, and Russell A. Rogers. Facilitating the development of representations in hypertext with IDE. In HYPERTEXT89 [55], pages 93–104.

[62] Rick Kazman and S. Jeromy Carrière. Playing detective: Reconstructing software architecture from available evidence. Technical Report CMU/SEI-97-TR-010, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, October 1997.

[63] Joseph F. Kess. *Psycholinguistics: Introductory Perspectives*. Academic Press, New York, New York, 1976.

[64] Barbara Kitchenham, Lesley Pickard, and Shari Lawrence Pfleeger. Case studies for method and tool evaluation. *IEEE Software*, pages 52–62, July 1995.

[65] Michael F. Kleyn and Paul C. Gingrich. GraphTrace—Understanding object-oriented systems using concurrently animated views. In *OOPSLA'88 Conference on Object-Oriented Programming Systems, Languages, and Applications Proceedings*, pages 191–205, San Diego, California, September 1988. ACM/SIGPLAN, ACM Press.

[66] Kostas Kontogiannis, Morris Bernstein, Ettore Merlo, and Renato De Mori. The development of a partial design recovery environment for legacy systems. In Gawman et al. [43], pages 206–216.

[67] M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, September 1980.

[68] John R. Levine, Tony Mason, and Doug Brown. *Lex and Yacc*. O'Reilly & Associates, Sebastopol, California, 1992.

[69] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental models and software maintenance. In *Empirical Studies of Programmers*. Ablex, Norwood, New Jersey, 1986.

[70] Gerald L. Lohse, Kevin Biolsi, Neff Walker, and Henry H. Rueter. A classification of visual representations. *Communications of the ACM*, 37(12):36–49, December 1994.

[71] James R. Lyle and Mark Weiser. Automatic program bug location by program slicing. In *Proceedings of the Second International Conference on Computers and Applications*, pages 877–, Beijing, China, June 1987. IEEE Computer Society Press.

[72] Allen D. Malony, David H. Hammerslag, and David J. Jablonowski. Traceview: A trace visualization tool. *IEEE Software*, pages 19–28, September 1991.

[73] Revital Marom and Patrick J. Finnigan. Current issues in visualization—a survey. Technical Report TR-74-100, IBM Canada Laboratory, Toronto, Ontario, Canada, June 1992.

[74] Mathematica. http://www.mathematica.com/.

[75] Alberto Mendelzon and Johannes Sametinger. Reverse engineering by visualizing and querying. *Software—Concepts and Tools*, 16:170–182, 1995.

[76] Bertrand Meyer. *Object-oriented Software Construction*. Prentice-Hall, New York, New York, 1988.

[77] Barton P. Miller. What to draw? When to draw? An essay on parallel program visualization. Technical Report 1103, Computer Sciences Department, University of Wisconsin-Madison, Madison, Wisconsin, June 1992.

[78] G. A. Miller. The magical number seven plus or minus two: Some limits on our capacity for processing information. *Psychology Review*, 63:81–97, 1956.

[79] Mirror Worlds. http://www.mirrorworlds.com/.

[80] Thomas G. Moher. PROVIDE: A process visualization and debugging environment. *IEEE Transactions on Software Engineering*, SE-14(6):849–857, June 1988.

[81] Hausi A. Müller. Rigi as a reverse engineering tool. Technical Report DCS-160-IR, Department of Computer Science, University of Victoria, Victoria, British Columbia, Canada, March 1991.

[82] Hausi A. Müller, Mehmet A. Orgun, Scott R. Tilley, and James S. Uhl. Discovering and reconstructing subsystem structures through reverse engineering. Technical Report DCS-201-IR, Department of Computer Science, University of Victoria, Victoria, British Columbia, Canada, August 1992.

[83] Hausi A. Müller, Mehmet A. Orgun, Scott R. Tilley, and James S. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5:181–204, 1993.

[84] Glenford J. Myers. *Reliable Software Through Composite Design*. Petrocelli/Charter, New York, New York, 1975.

[85] John Mylopoulos. Conceptual modelling and Telos. Technical Report DKBS-TR-91-3, University of Toronto, Toronto, Ontario, Canada, November 1991.

[86] John Mylopoulos, Martin Stanley, Kenny Wong, Morris Bernstein, Renato De Mori, Graham Ewart, Kostas Kontogiannis, Ettore Merlo, Hausi A. Müller, Scott R. Tilley, and Marijana Tomic. Towards an integrated toolset for program understanding. In John Botsford, Ann Gawman, M. Gentleman, Evelyn Kidd, Kelly Lyons, and Jacob Slonim, editors, *Proceedings of the 1994 CAS Conference*, pages 19–31, Toronto, Ontario, Canada, October 1994. IBM Centre for Advanced Studies.

[87] Bonnie A. Nardi. *A Small Matter of Programming*. MIT Press, Cambridge, Massachusetts, 1993.

[88] Object Design, Inc. http://www.objectstore.net/.

[89] David P. Olshefski and Alan Cole. A prototype system for static and dynamic program understanding. In Richard C. Waters and Elliot J. Chikofsky, editors, *Proceedings of the Working Conference on Reverse Engineering*, pages 93–106, Baltimore, Maryland, May 1993. IEEE Computer Society Press.

[90] Object Management Group. http://www.omg.org/.

[91] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[92] John K. Ousterhout. Scripting: Higher-level programming for the 21st century. *IEEE Computer*, pages 23–30, March 1998.

[93] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[94] T. J. Parr, H. G. Dietz, and W. E. Cohen. Purdue Compiler Construction Tool Set (PCCTS) reference manual. *ACM SIGPLAN Notices*, pages 88–165, February 1992.

[95] Santanu Paul and Atul Prakash. Source code retrieval using program patterns. In Gene Forte, Nazim H. Madhavji, and Hausi A. Müller, editors, *Proceedings of the Fifth International Workshop on Computer-Aided Software Engineering*, pages 95–105, Montréal, Québec, Canada, July 1992. IEEE Computer Society Press.

[96] Dewayne E. Perry. Software interconnection models. In *Proceedings of the 9th International Conference on Software Engineering*, pages 61–69, Monterey, California, March 1987. IEEE Computer Society Press.

[97] Marian Petre. Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44, June 1995.

[98] M. C. Pong and N. Ng. PIGS: A system for programming with interactive graphical support. *Software—Practice and Experience*, 13(9):847–855, September 1983.

[99] 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, 1993.

[100] Jan A. Purchase and Russel L. Winder. Debugging tools for object-oriented programming. *Journal of Object-Oriented Programming*, pages 10–27, June 1991.

[101] Sridhar A. Raghavan and Donald R. Chand. Diffusing software-engineering methods. *IEEE Software*, pages 81–90, July 1989.

[102] Balasubramaniam Ramesh. Factors influencing requirements traceability practice. *Communications of the ACM*, pages 37–44, December 1998.

[103] Resource Description Framework (RDF). http://www.w3.org/RDF/.

[104] Reasoning Systems, Inc. http://www.reasoning.com/.

[105] Steven Reiss. *The FIELD Programming Environment: A Friendly Integrated Environment for Learning and Development*. Kluwer Academic, Boston, Massachusetts, 1995.

[106] Steven P. Reiss. Pecan: Program development systems that support multiple views. *IEEE Transactions on Software Engineering*, SE-11(3):276–285, March 1985.

[107] Steven P. Reiss. On the use of annotations for integrating the source in a program development environment. In Finkelstein et al. [38], pages 25–36.

[108] Marc Rettig. Hat racks for understanding. *Communications of the ACM*, 35(10):21–24, October 1992.

[109] George G. Robertson, Stuart K. Card, and Jock D. Mackinlay. Information visualization using 3D interactive animation. *Communications of the ACM*, 36(4):57–71, April 1993.

[110] Marc J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, 1(4):364–370, December 1975.

[111] E. M. Rogers. *The Diffusion of Innovations*. Free Press, New York, New York, 1983.

[112] Gruia-Catalin Roman and Kenneth C. Cox. A taxonomy of program visualization systems. *IEEE Computer*, pages 11–24, December 1993.

[113] Rosetta project. http://www.cs.queensu.ca/home/graham/rosetta.htm.

[114] Johannes Sametinger. Improving program comprehension of object-oriented software systems with object-oriented documentation. In *Proceedings of the First Workshop on Program Comprehension*, pages 48–50. IEEE Computer Society Press, 1992. Position statement.

[115] Manojit Sarkar and Marc H. Brown. Graphical fisheye views of graphs. In Bauersfeld et al. [10], pages 83–91.

[116] Bill N. Schilit, Gene Golovchinsky, and Catherine C. Marshall. As we may read: The reading appliance revolution. *IEEE Computer*, pages 65–73, January 1999.

[117] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Englewood Cliffs, New Jersey, 1996.

[118] Takao Shimomura and Sadahiro Isoda. Linked-list visualization for debugging. *IEEE Software*, pages 44–51, May 1991.

[119] Ben Shneiderman. *Software Psychology: Human Factors in Computer and Information Systems*. Winthrop, 1980.

[120] Ian Sommerville. *Software Engineering Environments*. Peter Peregrinus Ltd., London, United Kingdom, 1986.

[121] Richard M. Stallman. EMACS: The extensible, customizable, self-documenting display editor. In *Proceedings of the ACM SIGPLAN SIGOA '81 Symposium on Text Manipulation*, pages 147–156, Portland, Oregon, June 1981. ACM Press.

[122] Margaret-Anne D. Storey. *A Cognitive Framework for Describing and Evaluating Software Exploration Tools*. PhD thesis, Simon Fraser University, Burnaby, British Columbia, Canada, 1998.

[123] Margaret-Anne D. Storey, H. A. Müller, and Kenny Wong. Manipulating and documenting software structures. In *Manipulating and Documenting Software Structures* [31], pages 255–263.

[124] Margaret-Anne D. Storey, Kenny Wong, P. Fong, D. Hooper, K. Hopkins, and Hausi A. Müller. On designing an experiment to evaluate a reverse engineering tool. In Linda Wills, Ira Baxter, and Elliot J. Chikofsky, editors, *Proceedings of the Third Working Conference on Reverse Engineering*, pages 31–40, Monterey, California, November 1996. IEEE Computer Society Press.

[125] Margaret-Anne D. Storey, Kenny Wong, and Hausi A. Müller. How do program understanding tools affect how programmers understand programs? In Ira Baxter, Alex Quilici, and Chris Verhoef, editors, *Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 12–21, Amsterdam, The Netherlands, October 1997. IEEE Computer Society Press.

[126] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, second edition, 1991.

[127] Tarja Systä. On the relationships between static and dynamic models in reverse engineering java software. In WCRE99 [149], pages 304–313.

[128] TakeFive Software, Inc. http://www.takefive.com/.

[129] Harold Thimbley. *User Interface Design*. ACM Press Frontier Series. ACM Press, New York, New York, 1990.

[130] Ian Thomas and Brian A. Nejmeh. Definitions of tool integration for environments. *IEEE Software*, pages 29–35, March 1992.

[131] Walter F. Tichy. RCS: A system for version control. *Software—Practice and Experience*, 15(7):637–654, July 1985.

[132] Walter F. Tichy. Should computer scientists experiment more? *IEEE Computer*, pages 32–40, May 1998.

[133] Scott R. Tilley. *Domain-Retargetable Reverse Engineering*. PhD thesis, University of Victoria, Victoria, British Columbia, Canada, 1995.

[134] Scott R. Tilley and Hausi A. Müller. INFO: A simple document annotation facility. In *SIGDOC '91: Proceedings of the 9th Annual International Conference on Systems Documentation*, pages 30–36, Chicago, Illinois, October 1991. ACM Press.

[135] Scott R. Tilley, Hausi A. Müller, and Mehmet A. Orgun. Documenting software systems with views. In *SIGDOC '92: Proceedings of the 10th Annual International Conference on Systems Documentation*, pages 211–219, Ottawa, Ontario, Canada, October 1992. ACM Press.

[136] Scott R. Tilley, Michael J. Whitney, Hausi A. Müller, and Margaret-Anne D. Storey. Personalized information structures. In *SIGDOC '93: Proceedings of the 11th Annual International Conference on Systems Documentation*, pages 325–337, Waterloo, Ontario, Canada, October 1993. ACM Press.

[137] Scott R. Tilley, Kenny Wong, Margaret-Anne D. Storey, and Hausi A. Müller. Programmable reverse engineering. *International Journal of Software Engineering and Knowledge Engineering*, 4(4):501–520, December 1994.

[138] Together/J. http://www.togetherj.com/.

[139] Michael Travers. A visual representation for knowledge structures. In HYPERTEXT89 [55], pages 147–158.

[140] Joel Troster, John Henshaw, and Erich Buss. Filtering for quality. In Gawman et al. [43], pages 429–449.

[141] David Turo and Brian Johnson. Improving the visualization of hierarchies with treemaps: Design issues and experimentation. In VISUALIZATION92 [143], pages 124–131.

[142] Amjad Umar. *Application Reengineering*. Prentice-Hall, Upper Saddle River, New Jersey, 1997.

[143] *Proceedings Visualization '92*, Boston, Massachusetts, October 1992. IEEE Computer Society Press.

[144] A. von Mayrhauser and A. M. Vans. From program comprehension to tool requirements for an industrial environment. In Fadini and Rajlich [35], pages 78–86.

[145] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *IEEE Computer*, pages 44–55, August 1995.

[146] Lawrence G. Votta and Adam Porter. Experimental software engineering: A report on the state of the art. In *Proceedings of the 17th International Conference on Software Engineering*, pages 277–279, Seattle, Washington, April 1995. IEEE Computer Society Press.

[147] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, Sebastopol, California, 1996.

[148] Anthony I. Wasserman. Tool integration in software engineering environments. In Fred Long, editor, *Software Engineering Environments, International Workshop on Environments Proceedings*, number 467 in Lecture Notes in Computer Science, pages 137–149, Chinon, France, September 1989. Springer-Verlag.

[149] *Proceedings of the Sixth Working Conference on Reverse Engineering*, Atlanta, Georgia, October 1999. IEEE Computer Society Press.

[150] Michael J. Whitney, Kostas Kontogiannis, J. Howard Johnson, Morris Bernstein, Brian Corrie, Ettore Merlo, James G. McDaniel, Renato De Mori, Hausi A. Müller, John Mylopoulos, Martin Stanley, Scott R. Tilley, and Kenny Wong. Using an integrated toolset for program understanding. In Karen Bennet, Dennis Bockus, Morven Gentleman, Howard Johnson, Evelyn Kidd,

Jacob Slonim, and Anne Stilman, editors, *Proceedings of the 1995 CAS Conference*, pages 262–274, Toronto, Ontario, Canada, November 1995. IBM Centre for Advanced Studies.

[151] Kenny Wong. On inserting program understanding technology into the software change process. In *Proceedings of the Fourth Workshop on Program Comprehension*, pages 90–99, Berlin, Germany, March 1996. IEEE Computer Society Press.

[152] Kenny Wong. *Rigi User's Manual*, June 1998. Version 5.4.4.

[153] Kenny Wong, Scott R. Tilley, Hausi A. Müller, and Margaret-Anne D. Storey. Structural redocumentation: A case study. *IEEE Software*, pages 46–54, January 1995.

[154] XML Metadata Interchange (XMI). http://www.software.ibm.com/ad/ features/xmi.html.

[155] Ralph Zainlinger. Building interfaces for case environments: An object oriented interaction model and its application. In Finkelstein et al. [38], pages 65–80.

[156] Marvin V. Zelkowitz and Dolores R. Wallace. Experimental models for validating technology. *IEEE Computer*, pages 23–31, May 1998.

[157] Polle T. Zellweger. Scripted documents: A hypermedia path mechanism. In HYPERTEXT89 [55], pages 1–14.

Appendix A

Scripting Examples

A.1 Constructive abstraction script

The following is a recorded script (with some manual editing) to group nodes of various types into related subsystems for a Monopoly game program [125]:

```
rcl_select_name "cards.c" 0
rcl_select_name "execute.c" 1
rcl_select_name "getinp.c" 1
rcl_select_name "houses.c" 1
rcl_select_name "initdeck.c" 1
rcl_select_name "jail.c" 1
rcl_select_name "misc.c" 1
rcl_select_name "monop.c" 1
rcl_select_name "morg.c" 1
rcl_select_name "print.c" 1
rcl_select_name "prop.c" 1
rcl_select_name "rent.c" 1
rcl_select_name "roll.c" 1
rcl_select_name "spec.c" 1
rcl_select_name "trade.c" 1
rcl_select_name "deck.h" 1
rcl_select_name "monop.h" 1
rcl_select_name "pathnames.h" 1
rcl_select_name "brd.dat" 1
rcl_select_name "mon.dat" 1
rcl_select_name "monop.def" 1
rcl_select_name "monop.ext" 1
```

```
rcl_select_name "prop.dat" 1
rcl_filter_selection

rcl_select_type "File"
rcl_collapse Collapse "System Files"
rcl_filter_unfilter

rcl_select_name "dk_st" 0
rcl_select_name "DECK" 1
rcl_select_name "own_st" 1
rcl_select_name "OWN" 1
rcl_select_name "plr_st" 1
rcl_select_name "PLAY" 1
rcl_select_name "prp_st" 1
rcl_select_name "PROP" 1
rcl_select_name "RR_S" 1
rcl_select_name "UTIL_S" 1
rcl_select_name "sqr_st" 1
rcl_select_name "SQUARE" 1
rcl_select_name "trd_st" 1
rcl_select_name "MON" 1
# rcl_select_name "TRADE" 1
rcl_select_name "monop.h" 1
rcl_select_name "deck.h" 1
rcl_collapse Collapse "Data Structures"

# rcl_select_type "Unknown" 0
# rcl_collapse Collapse "Unknown"

rcl_filter_nodetype [rcl_get_nodetype Memory]
rcl_filter_nodetype [rcl_get_nodetype Var]
rcl_filter_apply

rcl_select_name "abs" 0
rcl_select_name "brk" 1
rcl_select_name "calloc" 1
rcl_select_name "cfree" 1
rcl_select_name "close" 1
rcl_select_name "creat" 1
rcl_select_name "ctime" 1
rcl_select_name "exit" 1
rcl_select_name "fflush" 1
rcl_select_name "fopen" 1
rcl_select_name "fread" 1
rcl_select_name "fseek" 1
rcl_select_name "fstat" 1
rcl_select_name "getpid" 1
rcl_select_name "getw" 1
rcl_select_name "isalpha" 1
```

```
rcl_select_name "isdigit" 1
rcl_select_name "isspace" 1
rcl_select_name "isupper" 1
rcl_select_name "open" 1
rcl_select_name "perror" 1
rcl_select_name "printf" 1
rcl_select_name "rand" 1
rcl_select_name "read" 1
rcl_select_name "sbrk" 1
rcl_select_name "signal" 1
rcl_select_name "sprintf" 1
rcl_select_name "srand" 1
rcl_select_name "strcasecmp" 1
rcl_select_name "strcpy" 1
rcl_select_name "strlen" 1
rcl_select_name "stat" 1
rcl_select_name "time" 1
rcl_select_name "tolower" 1
rcl_select_name "write" 1
rcl_collapse Collapse "System Procs"

rcl_select_name "play" 0
rcl_select_name "play.loc" 1
rcl_select_name "play.money" 1
rcl_select_name "play.own_list.sqr.name" 1
rcl_select_name "play.num_rr" 1
rcl_select_name "play.num_util" 1
rcl_select_name "player" 1
rcl_select_name "cur_p" 1
rcl_select_name "cur_p.in_jail" 1
rcl_select_name "cur_p.loc" 1
rcl_select_name "cur_p.money" 1
rcl_select_name "cur_p.num_gojf" 1
rcl_select_name "cur_p.own_list.sqr.name" 1
rcl_select_name "name_list" 1
rcl_select_name "num_play" 1
rcl_collapse Collapse "Player Globals"

rcl_select_name "comlist" 0
rcl_select_name "func" 1
rcl_select_name "yn" 1
rcl_collapse Collapse "Command Globals"

rcl_select_name "lucky_mes" 0
rcl_select_name "num_luck" 1
rcl_collapse Collapse "Lucky Messages"

rcl_select_name "board" 0
rcl_select_name "board.cost" 1
```

```
rcl_select_name "board.desc" 1
rcl_select_name "board.name" 1
rcl_select_name "board.owner" 1
rcl_select_name "board.type" 1
rcl_select_name "brd.dat" 1
rcl_select_name "copyright" 1
rcl_select_name "deck" 1
rcl_select_name "deck.gojf_used" 1
rcl_select_name "deck.num_cards" 1
rcl_select_name "fixing" 1
rcl_select_name "mon" 1
rcl_select_name "mon.dat" 1
rcl_select_name "num_doub" 1
rcl_select_name "prop" 1
rcl_select_name "prop.dat" 1
rcl_select_name "rr" 1
rcl_select_name "spec" 1
rcl_select_name "told_em" 1
rcl_select_name "trading" 1
rcl_select_name "util" 1
rcl_select_name "monop.def" 1
rcl_select_name "monop.ext" 1
rcl_collapse Collapse "General Globals"

rcl_select_name "init_decks" 0
rcl_select_name "set_up" 1
rcl_collapse Collapse "Card Decks"

rcl_select_name "Card Decks" 0
rcl_select_name "get_card" 1
rcl_select_name "printmes" 1
rcl_select_name "cards.c" 1
rcl_select_name "pathnames.h" 1
rcl_collapse Collapse "Cards"

rcl_select_name "card" 0
rcl_select_name "move_jail" 1
rcl_select_name "pay" 1
rcl_select_name "printturn" 1
rcl_select_name "ret_card" 1
rcl_select_name "jail.c" 1
rcl_collapse Collapse "Jail"

rcl_select_name "do_move" 0
rcl_select_name "execute" 1
rcl_select_name "move" 1
rcl_select_name "next_play" 1
rcl_select_name "show_move" 1
rcl_select_name "execute.c" 1
```

```
rcl_collapse Collapse "Normal Moves"

rcl_select_name "cc" 0
rcl_select_name "chance" 1
rcl_select_name "goto_jail" 1
rcl_select_name "inc_tax" 1
rcl_select_name "lux_tax" 1
rcl_select_name "spec.c" 1
rcl_collapse Collapse "Special Moves"

rcl_select_name "Cards" 0
rcl_select_name "Jail" 1
rcl_select_name "Normal Moves" 1
rcl_select_name "Special Moves" 1
rcl_collapse Collapse "Moving"

rcl_select_name "restore" 0
rcl_select_name "rest_f" 1
rcl_select_name "save" 1
rcl_collapse Collapse "Load/Save Game"

rcl_select_name "comp" 0
rcl_select_name "get_int" 1
rcl_select_name "getinp" 1
rcl_select_name "getyn" 1
rcl_select_name "getinp.c" 1
rcl_collapse Collapse "Input"

rcl_select_name "list" 0
rcl_select_name "list_all" 1
rcl_select_name "notify" 1
rcl_select_name "printboard" 1
rcl_select_name "printhold" 1
rcl_select_name "printmorg" 1
rcl_select_name "printsq" 1
rcl_select_name "where" 1
rcl_select_name "print.c" 1
rcl_collapse Collapse "Misc Output"

rcl_select_name "buy_h" 0
rcl_select_name "buy_houses" 1
rcl_select_name "sell_h" 1
rcl_select_name "sell_houses" 1
rcl_select_name "list_cur" 1
rcl_select_name "houses.c" 1
rcl_collapse Collapse "Houses"

rcl_select_name "fix_ex" 0
rcl_select_name "force_morg" 1
```

```
rcl_select_name "m" 1
rcl_select_name "mortgage" 1
rcl_select_name "set_mlist" 1
rcl_select_name "set_umlist" 1
rcl_select_name "unm" 1
rcl_select_name "unmortgage" 1
rcl_select_name "morg.c" 1
rcl_collapse Collapse "Mortgage"

rcl_select_name "add_list" 0
rcl_select_name "bid" 1
rcl_select_name "buy" 1
rcl_select_name "del_list" 1
rcl_select_name "prop_worth" 1
rcl_select_name "value" 1
rcl_select_name "prop.c" 1
rcl_collapse Collapse "Properties"

rcl_select_name "rent" 0
rcl_select_name "rent.c" 1
rcl_collapse Collapse "Rent"

rcl_select_name "do_trade" 0
rcl_select_name "get_list" 1
rcl_select_name "move_em" 1
rcl_select_name "resign" 1
rcl_select_name "set_list" 1
rcl_select_name "summate" 1
rcl_select_name "trade" 1
rcl_select_name "trade.c" 1
rcl_collapse Collapse "Trade"

rcl_select_name "cpy_st" 0
rcl_select_name "is_monop" 1
rcl_select_name "isnot_monop" 1
rcl_select_name "set_ownlist" 1
rcl_select_name "misc.c" 1
rcl_collapse Collapse "Util"

rcl_select_name "Houses" 0
rcl_select_name "Mortgage" 1
rcl_select_name "Properties" 1
rcl_select_name "Rent" 1
rcl_select_name "Trade" 1
rcl_select_name "Util" 1
rcl_collapse Collapse "Buying & Selling"

rcl_select_name "roll" 0
rcl_select_name "roll.c" 1
```

```
rcl_collapse Collapse "Rolling Dice"

rcl_select_name "Misc Output" 0
rcl_select_name "Moving" 1
rcl_select_name "Buying & Selling" 1
rcl_select_name "Rolling Dice" 1
rcl_collapse Collapse "Play"

rcl_select_name "getplayers" 0
rcl_select_name "init_monops" 1
rcl_select_name "init_players" 1
rcl_collapse Collapse "Setup"

rcl_select_name "quit" 0
rcl_collapse Collapse "Finish"

rcl_select_name "main" 0
rcl_select_name "monop.c" 1
rcl_collapse Collapse "Control"

rcl_select_all
rcl_collapse Collapse "Monopoly Program"
```

Appendix B

Web Integration

B.1 Startup Script

This startup script sets Rigi in a polling state for an input file of commands.

```
# startup.rcl

# ----------
# start and stop processing ...

proc web_start {} {
    global parms
    set parms(go) 1
}

proc web_stop {} {
    global parms
    set parms(go) 0
}

# ----------
# main polling routine ...

proc web_update {} {
    global parms

    if {$parms(go) == 0} return
```

```
    set dir $parms(dir)
    set lockfile ${dir}lock

    if {[file exists $lockfile]} return
    exec touch $lockfile

    set inputfile ${dir}input
    if {[file size $inputfile]} {
        source $inputfile
        exec /bin/rm $inputfile
        exec touch $inputfile

        exec /bin/rm $lockfile

        switch $parms(op) {
            doview { web_doview }
        }
    } else {
        exec /bin/rm $lockfile
    }
}

# ----------
# dispatch views ...

proc web_doview {} {
    global parms

    set op              $parms(op)
    set cookie          $parms(cookie)
    set userid          $parms(userid)
    set project         $parms(project)
    set domain          $parms(domain)
    set servername      $parms(servername)
    set scriptname      $parms(scriptname)

    set view            $parms(view)
    set db              $parms(db)
    set context         $parms(context)
    set comment         $parms(comment)

    switch $view {
        auto {}
        grid { rcl_grid_all }
        selectname { rcl_select_name $context }
        enum { rcl_load_view [rcl_env_get DBDIR]/$context }

        # CSER stuff ...
        default { cser_dispatcher }
```

```
    }
}

# ----------
# get things going ...

web_stop

proc rcl_poll_proc {} {
    web_update
}

web_start
```

B.2 Server side CGI script

This script handles view requests from a web page.

```
#!/public/bin/perl
# rigiserver - server-side CGI script - KW

# ----------
# configurable parameters ...

# directory containing server-side template files
# (absolute, resolved path)
$gTemplateDir = "/rigi/proj/webrigi/server-tmpl/";
$gTemplateDir =~ s+([^/])$+\1/+;

# directory containing project-specific server-side files
$gProjectDir = ${gTemplateDir} . "project/";

# ----------
# some sanity checks ...

unless (-d $gTemplateDir) {
    &SendMessage( 500, "Server Error",
        "Cannot find server templates: $gTemplateDir", "exit(1)" );
}

unless (-d $gProjectDir) {
    &SendMessage( 500, "Server Error",
        "Cannot find project files: $gProjectDir", "exit(1)" );
}

# ----------
# dispatcher ...

&ParseForm( *gFormData );

# server operations jumptable
%gServerOps = (
    "startpage"         => "&StartPage",
    "opspage"           => "&OpsPage",
    "viewpage"          => "&ViewPage",
    "scriptpage"        => "&ScriptPage",
    "testpage"          => "&TestPage",
    "showenv"           => "&ShowEnv",
    "startup"           => "&StartUp",
    "doview"            => "&DoView",
```

```
    "doaview"           => "&DoView",
    "doscript"          => "&DoScript",
    "dotest"            => "&DoTest",
    "selectname"        => "&SelectName"
);
# "changecookie"        => "&ChangeCookie",
# "chooseview"          => "&ChooseView",

$op = $gFormData{'op'} || "startpage";

if ($gServerOps{$op}) {
    eval $gServerOps{$op};
} else {
    &SendMessage( 500, "Server Error",
        "Unknown server operation: $op", "exit(1)" );
}

exit( 0 );

# ----------
# HTML pages ...

# ?op=startpage
sub StartPage
{
    my( $seed, $cookie );

    for (1..4) {
        srand( time + $$ + $seed );
        $cookie = $cookie .
            sprintf( "%2.2x", ($seed = int( rand( 65536 ))));
    }
    # $cookie = "wnek"; # hardcoded cookie

    &SendHTML( "startpage.html", $cookie );
}

# ?op=opspage
# &cookie=<cookie>&userid=<userid>
# (set by startpage)
sub OpsPage
{
    &SendHTML( "opspage.html", $gFormData{'cookie'} );
}

# ?op=viewpage
# &cookie=<cookie>&userid=<userid>&project=<project>
# (passed thru opspage)
# &domain=<domain>
```

```
# (set by opspage)
sub ViewPage
{
    &SendHTML( "viewpage.html", $gFormData{'cookie'} );
}

# ?op=scriptpage
# &cookie=<cookie>&userid=<userid>&project=<project>
# (passed thru opspage)
# &domain=<domain>
# (set by opspage)
sub ScriptPage
{
    &SendHTML( "scriptpage.html", $gFormData{'cookie'} );
}

# ?op=testpage
# &cookie=<cookie>&userid=<userid>&project=<project>
# (passed thru opspage)
# &domain=<domain>
# (set by opspage)
sub TestPage
{
    &SendHTML( "testpage.html", $gFormData{'cookie'} );
}

# ----------
# server ops (the real work) ...

# use the indicated arguments in URLs

# show the CGI environment variables
# ?op=showenv
sub ShowEnv
{
    print "Content-type: text/html\n\n";

    print "<HTML>";
    print "<TITLE>CGI Environment Variables</TITLE>";
    print "<H1><HEAD>CGI Environment Variables</HEAD></H1>";
    print "<BODY>";

    print "<P><PRE>";
    while (($var, $info) = each %ENV) {
        print "$var = $info\n";
    }
    print "</PRE>";

    print "<HR>";
```

```
    print "</BODY>";
    print "</HTML>";
}

# launch rigiedit
# ?op=startup
# &cookie=<cookie>&userid=<userid>&project=<project>
# (spec by startpage)
sub StartUp
{
    &SendBasicParms( "startup", $gFormData{'cookie'} );

    # rigiedit is launched by Netscape through
    # rigiclient (client-side MIME handler)

    # both rigiedit and Netscape are on the same host, but
    # don't depend on it
}

# show view with rigiedit
# ?op=doview
# &cookie=<cookie>&userid=<userid>&project=<project>
# (passed thru viewpage)
# &domain=<domain>
# (also set by viewpage)
# &view=<view>&db=<database>&context=<context>
# (set by viewpage)
# &comment=<comment>
# (set by viewpage)
sub DoView
{
    my( $view ) = $gFormData{'view'};
    my( $db ) = $gFormData{'db'};
    my( $context ) = $gFormData{'context'};
    my( $comment ) = $gFormData{'comment'};

    # output checking ...
    $context = &WashedContext( $context );

    &SendBasicParms( "doview", $gFormData{'cookie'} );
    &SendParm( "view", $view );
    &SendParm( "db", $db );
    &SendParm( "context", $context );
    &SendParm( "comment", $comment );
}

# run Tcl script with rigiedit
# ?op=doscript
# &cookie=<cookie>&userid=<userid>&project=<project>
```

```
# (passed thru scriptpage)
# &domain=<domain>
# (also set by scriptpage)
# &script=<script>
# (set by scriptpage)
sub DoScript
{
    &SendScript( $gFormData{'script'}, $gFormData{'cookie'} );
}

# run test with rigiclient
# ?op=dotest
# &cookie=<cookie>&userid=<userid>&project=<project>
# (passed thru testpage)
# &domain=<domain>
# (also set by testpage)
sub DoTest
{
    &SendBasicParms( "dotest", $gFormData{'cookie'} );
}

# select by name
# ?op=selectname
# &cookie=<cookie>&userid=<userid>&project=<project>
# &name=<name>
sub SelectName
{
    &SendScript(
        "rcl_select_name $gFormData{'name'}",
        $gFormData{'cookie'}
    );
}

# ----------
# complete responses (sent back to browser) ...

# respond with nothing
sub SendNoResponse
{
    print "Content-type: text/plain\n";
    print "Status: 204 No Response\n\n";
}

# respond with information message (usually for errors)
sub SendMessage
{
    my( $status, $keyword, $message, $action ) = @_;

    print "Content-type: text/html\n";
```

```
    print "Status: $status $keyword\n\n";

    print "<HTML>\n";
    print "<TITLE>$keyword</TITLE>\n";
    print "<H1><HEAD>$keyword</HEAD></H1>\n";

    print "<BODY>\n";
    print "$message\n";
    print "<HR>";
    print "</BODY>\n";
    print "</HTML>\n";

    eval $action;
}

# respond with dynamic page by completing HTML file template
sub SendHTML
{
    my( $htmlFile, $cookie ) = @_;

    my( $userID ) = $gFormData{'userid'};
    my( $project ) = $gFormData{'project'};
    my( $domain ) = $gFormData{'domain'};
    # some of these may be unspecified

    my( $serverHost ) = $ENV{'SERVER_NAME'};
    my( $scriptName ) = $ENV{'SCRIPT_NAME'};

    # output checking ...
    $cookie = &WashedCookie( $cookie );
    $userID = &WashedUserID( $userID );
    $project = &WashedProject( $project );
    $domain = &WashedDomain( $domain );
    $scriptName = &WashedScriptName( $scriptName );

    print "Content-type: text/html\n";
    print "Pragma: no-cache\n\n";

    open( FSH, "< $gTemplateDir$htmlFile" ) ||
        &SendMessage( 500, "Server Error",
        "Cannot read file: $gTemplateDir$htmlFile", "exit(1)" );
    while (<FSH>) {
        s/#COOKIE#/$cookie/g;
        s/#USERID#/$userID/g;
        s/#PROJECT#/$project/g;
        s/#DOMAIN#/$domain/g;
        s/#SERVERHOST#/$serverHost/g;
        s/#SCRIPTNAME#/$scriptName/g;
        s+#SCRIPTURL#+http://$serverHost$scriptName+g;
```

```
        if (s/^#INCLOPTS ([^#]+)#.*/\1/s) {
            &SendOptions( $_ );
        } else {
            print;
        }
    }
    close( FSH );
}

# respond with dynamic options
sub SendOptions
{
    my( $dataFile ) = @_;
    my( $value, $name );

    open( FSD, "< $gProjectDir$dataFile" ) ||
        &SendMessage( 500, "Server Error",
        "Cannot read file: $gProjectDir$dataFile", "exit(1)" );
    while (<FSD>) {
        next if (/^\s*#/);
        chop;
        ($value, $name) = split( /#/ );
        $name = $value unless ($name);
        print "<OPTION VALUE=\"$value\">$name\n";
    }
    close( FSD );
}

# ----------
# complete responses (sent back to browser,
# then passed to MIME handler) ...

# send script as Tcl code
sub SendScript
{
    my( $script, $cookie ) = @_;

    &SendBasicParms( "doscript", $cookie );
    print "# Tcl\n";
    print $script;
}

# send parameters as Perl code:
# op, cookie, userid, project, domain, serverhost, scriptname
sub SendBasicParms
{
    my( $clientOp, $cookie ) = @_;
```

```
    my( $userID ) = $gFormData{'userid'};
    my( $project ) = $gFormData{'project'};
    my( $domain ) = $gFormData{'domain'};

    my( $serverHost ) = $ENV{'SERVER_NAME'};
    my( $scriptName ) = $ENV{'SCRIPT_NAME'};

    # output checking ...
    $cookie = &WashedCookie( $cookie );
    $userID = &WashedUserID( $userID );
    $project = &WashedProject( $project );
    $domain = &WashedDomain( $domain );
    $scriptName = &WashedScriptName( $scriptName );

    print "Content-type: application/x-rigiclient\n\n";

    print "# Perl\n";
    &SendParm( "op", $clientOp );
    &SendParm( "cookie", $cookie );
    &SendParm( "userid", $userID );
    &SendParm( "project", $project );
    &SendParm( "domain", $domain );
    &SendParm( "serverhost", $serverHost );
    &SendParm( "scriptname", $scriptName );
}

# ----------
# partial responses ...

# send a parameter
sub SendParm
{
    my( $name, $value ) = @_;

    print q/$gParms{'/ . $name . qq/'} = "/ . $value . qq/";\n/;
}

# start server push of a multipart message
sub SendMultiPartHead
{
    print "Content-type: multipart/x-mixed-replace;boundary=End\n\n";
    # no prologue
    print "--End\n";
}

# end each part of the multipart message
sub SendMultiPartFoot
{
    print "--End\n";
```

```
}

# ----------
# input/output "washing" (useful when forms are not used) ...

# allow only alphanumeric and _ in cookie
sub WashedCookie
{
    my( $cookie ) = @_;

    # cookie is required input
    unless ($cookie) {
        &SendMessage( 500, "Server Error",
            "No cookie specified", "exit(1)" );

        # $cookie = "ChocoChip";
    }

    $cookie =~ s/[^\w]//g;
    return $cookie;
}

# allow only alphanumeric and _
sub WashedUserID
{
    my( $userID ) = @_;

    # userID is optional input
    # $userID = "nouserid" unless ($userid);
    $userID = "rigiuser" unless ($userid);
    # might see nouserid in response

    $userID =~ s/[^\w]//g;
    return $userID;
}

# allow only alphanumeric, _, space, and parens
sub WashedProject
{
    my( $project ) = @_;

    $project = "noproject" unless ($project);
    # should not see noproject in response

    $project =~ s/[^\w ()]//g;
    return $project;
}

# allow only alphanumeric and _
```

```
sub WashedDomain
{
    my( $domain ) = @_;

    $domain = "nodomain" unless ($domain);
    # should not see nodomain in response

    # $domain =~ s/[^+\w]//g;
    return $domain;
}

# add leading slash if not present, just in case
sub WashedScriptName
{
    my( $scriptName ) = @_;

    $scriptName =~ s+^([^/])+/\1+;
    return $scriptName;
}

# replace commas with carets
sub WashedContext
{
    my( $context ) = @_;

    $context =~ s/,/^/g;
    return $context;
}

# ----------
# form parsing ...

sub ParseForm
{
    local( *FORMDATA ) = @_;
    my( $requestMethod, $queryString );
    my( @pairs, $pair, $key, $value );

    $requestMethod = $ENV{'REQUEST_METHOD'};

    if ($requestMethod eq "GET") {
        $queryString = $ENV{'QUERY_STRING'};
    } elsif ($requestMethod eq "POST") {
        read( STDIN, $queryString, $ENV{'CONTENT_LENGTH'} );
    } else {
        &SendMessage( 500, "Server Error",
            "Unsupported request method: $requestMethod", "exit(1)" );
    }
```

```
    @pairs = split( /&/, $queryString );
    foreach $pair (@pairs) {
        ($key, $value) = split( /=/, $pair );
        $value =~ tr/+/ /;
        $value =~ s/%([\dA-Fa-f][\dA-fa-f])/pack("C",hex($1))/eg;

        if (defined( $FORMDATA{$key} )) {
            $FORMDATA{$key} = join( "\0", $FORMDATA{$key}, $value );
        } else {
            $FORMDATA{$key} = $value;
        }
    }
}
```

B.3 Client side helper script

This helper script passes the view request to Rigi.

```
#!/public/bin/perl
# rigiclient - KW

$RIGI = $ENV{'RIGI'} ||
    die "$0: RIGI environment variable not set\n";

# ----------
# configurable parameters ...

# directory containing client-side template files
$gTemplateDir = "/rigi/proj/webrigi/client-tmpl/";
$gTemplateDir =~ s+([^/])$+\1/+;

# directory for temporary files
# (probably should be same as TMPDIR with slash)
$gTempDir = "/tmp/";
$gTempDir =~ s+([^/])$+\1/+;

# computational host machine
$gCompHost = "logan";

# display host machine
$gDispHost = "logan";

# ----------
# some sanity checks ...

unless (-d $gTemplateDir) {
    die "$0: Cannot find client templates: $gTemplateDir\n";
}

unless (-d $gTempDir) {
    die "$0: Cannot find temp directory: $gTempDir\n";
}

# ----------
# dispatcher ...

use Sys::Hostname;

$gThisHost = hostname();
```

```
if ($#ARGV == -1) {
    $gInStream = STDIN;
} else {
    $gFile = shift;

    unless (-f $gFile) {
        die "$0: Cannot find parameter file: $gFile\n";
    }

    # read and evaluate Perl code
    open( FS, "< $gFile" ) ||
        die "$0: Cannot read parameter file: $gFile\n";
    $gInStream = FS;
}

while (<$gInStream>) {
    last if (/# Tcl/);
    eval;
    die "$0: Bad parameter input: $_\n" if ($@);
    # print
}

$gCompHost = $gParms{'comphost'} || $gCompHost;

if ($gCompHost ne $gThisHost) {
    close( $gInStream );

    # run rigiclient remotely on computational host
    unless (-f $gFile) {
        die "$0: Unexpectedly missing parameter file\n";
    }

    open( FS, "< $gFile" ) ||
        die "$0: Cannot reread parameter file: $gFile\n";
    open( FSP,
        "| (rsh $gCompHost rigiclient -l $gParms{'userid'})" ) ||
        die "$0: Unable to establish remote connection\n";

    while (<FS>) {
        print FSP;
    }

    # wait for rsh to finish
    close( FSP );
    die "$0: Error with remote connection\n" if ($?);
    close( FS );
} else {
    # run locally
    $op = $gParms{'op'};
```

```
    # client operations jumptable
    %gClientOps = (
        "startup"               => "&StartUp",
        "doscript"              => "&DoScript",
        "doview"                => "&DoView",
        "dotest"                => "&DoTest"
    );

    if ($gClientOps{$op}) {
        eval $gClientOps{$op};
        print $@ if ($@)
    } else {
        die "$0: Unknown client operation: $op\n";
    }
}

exit( 0 );

# ----------
# client ops (the real work) ...

# start Rigi
sub StartUp
{
    my ( $dir ) = "$gTempDir$gParms{'cookie'}/";

    close( $gInStream );

    &PrepStart( $dir );

    # no need to lock file
    open( FSI, ">> ${dir}input" ) ||
        die "$0: Cannot append to file: ${dir}input\n";

    # acknowledgment action
    $timeStamp = localtime( time );
    print FSI "# Tcl $timeStamp\n";
    print FSI "exec netscape -remote openURL(" .
        "http://$gParms{'serverhost'}$gParms{'scriptname'}" .
        "?op=opspage&cookie=$gParms{'cookie'}&" .
        "userid=$gParms{'userid'}&" .
        "project=$gParms{'project'})\n";
    close( FSI );

    &RealStart( $dir );
}

# show view request
```

```
sub DoView
{
    my( $dir ) = "$gTempDir$gParms{'cookie'}/";
    my( $autoStart ) = 0;
    my( $timeStamp );

    unless (-d $dir) {
        $autoStart = 1;
        &PrepStart( $dir );
    }

    &Lock( $dir );

    open( FSI, ">> ${dir}input" ) ||
        die "$0: Cannot append to file: ${dir}input\n";

    # write globals
    $timeStamp = localtime( time );
    print FSI "# Tcl $timeStamp\n";
    &WriteBasicParms( FSI );
    &WriteParm( FSI, "dir", $dir );
    &WriteParm( FSI, "view", $gParms{'view'} );
    &WriteParm( FSI, "db", $gParms{'db'} );
    &WriteParm( FSI, "context", $gParms{'context'} );
    &WriteParm( FSI, "comment", $gParms{'comment'} );
    close( FSI );

    # system( "cat ${dir}input" );

    &UnLock( $dir );

    if ($autoStart == 1) {
        &RealStart( $dir );
    }
}

# run script request
sub DoScript
{
    my( $dir ) = "$gTempDir$gParms{'cookie'}/";
    my( $autoStart ) = 0;
    my( $timeStamp );

    unless (-d $dir) {
        $autoStart = 1;
        &PrepStart( $dir );
    }

    &Lock( $dir );
```

```
    open( FSI, ">> ${dir}input" ) ||
        die "$0: Cannot append to file: ${dir}input\n";

    # write globals
    $timeStamp = localtime( time );
    print FSI "# Tcl $timeStamp\n";
    &WriteBasicParms( FSI );
    &WriteParm( FSI, "dir", $dir );

    # write script
    while (<$gInStream>) {
        print FSI;
    }
    print FSI "\n";
    close( FSI );
    close( $gInStream );

    &UnLock( $dir );

    if ($autoStart == 1) {
        &RealStart( $dir );
    }
}

sub DoTest
{
    close( $gInStream );
}

# ----------
# Preparing and starting Rigi ...

sub PrepStart
{
    my( $dir ) = @_;
    my( $configFile, $timeStamp );

    mkdir( $dir, 0777 ) ||
        die "$0: Cannot create directory: $dir\n";

    # find a configuration file
    if (-f ($configFile = "$ENV{'RIGIUSER'}/rigicfg.env")) {
    } elsif (-f ($configFile = "$RIGI/rigicfg.env")) {
    } else {
        die "$0: Cannot find a configuration file\n";
    }

    # generate configuration file ...
```

```
    open( FST, "< $configFile" ) ||
        die "$0: Cannot read file: $configFile\n";
    open( FSE, "> ${dir}rigicfg.env" ) ||
        die "$0: Cannot write file: ${dir}rigicfg.env\n";

    while (<FST>) {
        s+^WEBROOT.*+WEBROOT = http://$gParms{'serverhost'}/+;
        print FSE;
    }
    close( FSE );
    close( FST );

    # generate startup script ...
    open( FST, "< ${gTemplateDir}startup.rcl" ) ||
        die "$0: Cannot read file: ${gTemplateDir}startup.rcl\n";
    open( FSR, "> ${dir}startup.rcl" ) ||
        die "$0: Cannot write file: ${dir}startup.rcl\n";

    # write globals
    $timeStamp = localtime( time );
    print FSR "# Tcl $timeStamp\n";
    &WriteBasicParms( FSR );
    # during an autostart, op!=startup
    &WriteParm( FSR, "dir", $dir );

    while (<FST>) {
        print FSR;
    }
    close( FSR );
    close( FST );

    # create input file (sourced by Tcl if non-empty) ...
    open( FSI, "> ${dir}input" ) ||
        die "$0: Cannot create file: ${dir}input\n";
    close( FSI );
}

sub RealStart
{
    my( $dir ) = @_;
    my( $prog );

    # adjust display ...
    $gDispHost = $gParms{'disphost'} || $gDispHost;

    if ($gDispHost ne $gThisHost) {
        $ENV{'DISPLAY'} = $gDispHost . ":0.0";
    } else {
        $ENV{'DISPLAY'} = ":0";
```

```
    }

    # launch rigiedit ...
    $ENV{'TCL_LIBRARY'} = "$RIGI/lib/tcl";
    $ENV{'TK_LIBRARY'} = "$RIGI/lib/tk";

    $prog = "$RIGI/bin/" . &ArchOS . "/rigiedit";

    if (-x $prog) {
        system "/usr/bin/nice -127 $prog " .
            "-i ${dir}startup.rcl -env ${dir}rigicfg.env -poll " .
            "-- -nowb " .
            "1>/dev/null 2>/dev/null";
        die "$0: Error with program launch\n" if ($?);
    } else {
        die "$0: Command not found\n";
    }

    # cleanup ...
    unlink(
        "${dir}startup.rcl",
        "${dir}rigicfg.env",
        "${dir}lock",
        "${dir}input"
    );
    rmdir( "$dir" ) ||
        die "$0: Cannot remove directory: $dir\n";
}

# ----------
# writing ...

# write parameters as Tcl code
sub WriteBasicParms
{
    my( $fileHandle ) = @_;

    &WriteParm( $fileHandle, "op", $gParms{'op'} );
    &WriteParm( $fileHandle, "cookie", $gParms{'cookie'} );
    &WriteParm( $fileHandle, "userid", $gParms{'userid'} );
    &WriteParm( $fileHandle, "project", $gParms{'project'} );
    &WriteParm( $fileHandle, "domain", $gParms{'domain'} );
    &WriteParm( $fileHandle, "serverhost", $gParms{'serverhost'} );
    &WriteParm( $fileHandle, "scriptname", $gParms{'scriptname'} );
}

# write parameter as a Tcl assignment
sub WriteParm
{
```

```
    my( $fileHandle, $name, $value ) = @_;

    print $fileHandle qq/set parms(/ . $name . qq/) "/ .
        $value . qq/"\n/;
}

# ----------
# utilities ...

sub Lock
{
    my( $dir ) = @_;

    while (-e "${dir}lock") {
        sleep( 1 );
    }
    open( FSL, "> ${dir}lock" ) ||
        die "$0: Cannot create file: ${dir}lock\n";
    close( FSL );
}

sub UnLock
{
    my( $dir ) = @_;

    unlink( "${dir}lock" );
}

# determine architecture
sub ArchOS
{
    my( $os, $ver1, $ver2, $mach ) =
        (split( /\s+/, '/usr/bin/uname -a' ))[0,2,3,4];

    if ($os eq "AIX") {
        return "rs6000-aix$ver2";
    } elsif ($os eq "SunOS") {
        $ver1 =~ s/([0-9]).*/\1/;
        $mach =~ s/(sun4).*/\1/;
        return "$mach-sunos$ver1";
    } else {
        die "$0: Unknown architecture\n";
    }
}
```

B.4 MIME type

The MIME type of the view request forwarded by the server and handled through the helper script:

```
application/x-rigiclient; rigiclient %s
```

Vita

Surname:	Wong	Given Name:	Kenny
Place of Birth:	Victoria, British Columbia, Canada		

Educational Institutions Attended:

University of Victoria	1984 to 1999

Degrees Awarded:

B.Sc. (Honours)	University of Victoria	1989
M.Sc.	University of Victoria	1991

Honours and Awards:

IBM Canada Research Fellowship	1993–1994
NSERC Postgraduate Scholarship	1989–1991, 1992–1994
UVic President's Research Scholarship	1989, 1990, 1993
Petch Award	1992
NSERC Undergraduate Student Research Award	1989
Digital Equipment of Canada Ltd. Award of Merit	1988
HTS Hi-Tech Systems Ltd. Scholarship	1988
Software Research Corporation Scholarship	1987
UVic President's Scholarship	1987, 1988
BC Post Secondary Scholarship	1986, 1987, 1988
Kiwanis Club of Victoria Scholarship	1985
Woods Trust Scholarship	1985, 1987
UVic Lindsay Award	1985
Provincial Government of BC Entrance Scholarship	1984
UVic President's Entrance Scholarship	1984
BC Forest Products Ltd. Entrance Scholarship	1984
IWA Local 1-180 Academic Scholarship	1984

Publications:

Articles in Refereed Publications

Journal Articles Published or Accepted

1 P.J. Finnigan, R. 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, 1997.

2 K. Wong, S.R. Tilley, H.A. Müller, and M.-A.D. Storey. Structural redocumentation: A case study. *IEEE Software*, pages 46–54, January 1995.

3 S.R. Tilley, K. Wong, M.-A.D. Storey, and H.A. Müller. Programmable reverse engineering. *International Journal of Software Engineering and Knowledge Engineering*, 4(4):501–520, December 1994.

4 E. Buss, R. De Mori, M. Gentleman, J. Henshaw, H. Johnson, K. Kontogiannis, H.A. Müller, J. Mylopoulos, S. Paul, A. Prakash, M. Stanley, S.R. Tilley, J. Troster, and K. Wong. Investigating reverse engineering technologies for the CAS program understanding project. *IBM Systems Journal*, 33(3):477–500, 1994.

Other Refereed Contributions

Conference Proceedings Articles

5 K. Kontogiannis, J. Martin, K. Wong, R. Gregory, H.A. Müller, and J. Mylopoulos. Code migration through transformations: An experience report. In *Proceedings of the 1998 CAS Conference*, pages 1–13, Toronto, Ontario, Canada, December 1998. IBM Centre for Advanced Studies. *Awarded Best Practical Paper*.

6 M.-A.D. Storey, K. Wong, and H.A. Müller. How do program understanding tools affect how programmers understand programs? In *Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 12–21, Amsterdam, The Netherlands, October 1997. IEEE Computer Society Press.

7 M.-A.D. Storey, K. Wong, and H.A. Müller. On integrating visualization techniques for effective software exploration. In *INFOVIS '97: Proceedings of IEEE Symposium on Information Visualization*, pages 38–45, Phoenix, Arizona, October 1997. IEEE Computer Society Press.

8 M.-A.D. Storey, K. Wong, and H.A. Müller. Rigi: A visualization environment for reverse engineering. In *Proceedings of the International Conference on Software Engineering*, pages 606–607, Boston, Massachusetts, May 1997. IEEE Computer Society Press.

9 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. In *Proceedings of the Third Working Conference on Reverse Engineering*, pages 31–40, Monterey, California, November 1996. IEEE Computer Society Press. *Awarded Outstanding Paper Contribution.*

10 K. Wong. On inserting program understanding technology into the software change process. In *Proceedings of the Fourth Workshop on Program Comprehension*, pages 90–99, Berlin, Germany, March 1996. IEEE Computer Society Press.

11 M.J. Whitney, K. Kontogiannis, J.H. Johnson, M. Bernstein, B. Corrie, E. Merlo, J.G. McDaniel, R. De Mori, H.A. Müller, J. Mylopoulos, M. Stanley, S.R. Tilley, and K. Wong. Using an integrated toolset for program understanding. In *Proceedings of the 1995 CAS Conference*, pages 262–274, Toronto, Ontario, Canada, November 1995. IBM Centre for Advanced Studies.

12 J. Mylopoulos, M. Stanley, K. Wong, M. Bernstein, R. De Mori, G. Ewart, K. Kontogiannis, E. Merlo, H.A. Müller, S.R. Tilley, and M. Tomic. Towards an integrated toolset for program understanding. In *Proceedings of the 1994 CAS Conference*, pages 19–31, Toronto, Ontario, Canada, October 1994. IBM Centre for Advanced Studies.

13 S.R. Tilley, H.A. Müller, M.J. Whitney, and K. Wong. Domain-retargetable reverse engineering. In *Proceedings of the Conference on Software Maintenance*, pages 142–151, Montreal, Quebec, Canada, September 1993. IEEE Computer Society Press.

14 H.A. Müller, S.R. Tilley, and K. Wong. Understanding software systems using reverse engineering technology: Perspectives from the Rigi project. In *Proceedings of the 1993 CAS Conference*, pages 217–226, Toronto, Ontario, Canada, October 1993. IBM Centre for Advanced Studies.

15 K. Wong. Managing views in a program understanding tool. In *Proceedings of the 1993 CAS Conference*, pages 244–249, Toronto, Ontario, Canada, October 1993. IBM Centre for Advanced Studies.

Non-Refereed Contributions

Book Chapters

16 M.-A.D. Storey, H.A. Müller, and K. Wong. Manipulating and documenting software structures. In P. Eades and K. Zhang, editors, *Software Visualization*, volume 7 of *Series on Software Engineering and Knowledge Engineering*, pages 255–263. World Scientific, 1996. Refereed in Proceedings of the International Conference on Software Maintenance 1995.

17 H.A. Müller, K. Wong, and S.R. Tilley. Understanding software systems using reverse engineering technology. In V. S. Alagar and R. Missaoui, editors, *Object-Oriented Technology for Database and Software Systems*, pages 240–252. World Scientific, 1995. Refereed in Proceedings of the 1993 CAS Conference.

Invited Papers

18 H.A. Müller, K. Wong, and S.R. Tilley. Understanding software systems using reverse engineering technology. In *Proceedings of the 62nd Congress L'Association Canadienne Francaise pour l'Avancement des Sciences, Colloquium on Object Orientation in Databases and Software Engineering*, pages 41–48, Montreal, Quebec, Canada, May 1994.

Workshop Reports

19 S.R. Tilley and K. Wong. Workshop report of the seventh international conference on computer-aided software engineering, July 1995. 42 pages.

20 S.R. Tilley and K. Wong. Report on NWSEE '93: The 1993 National Workshop on Software Engineering Education. Technical Report TR 74-131, IBM Canada Ltd. Laboratory, Toronto, Ontario, Canada, August 1993. 28 pages.

Position Papers

21 K. Wong. Toward reusable and evolvable web sites. In *Proceedings of the First International Workshop on Web Site Evolution*, Atlanta, Georgia, October 1999. IEEE Computer Society Press.

22 H.A. Müller, K. Wong, and M.-A.D. Storey. Reverse engineering research should target cooperative information system requirements. In *Proceedings of the Fifth Working Conference on Reverse Engineering*, Honolulu, Hawaii, October 1998. IEEE Computer Society Press.

23 H.A. Müller, K. Wong, and M.-A.D. Storey. Wrapping coarse-grained objects using standard infrastructure technology. In *Proceedings of the International Conference on Software Maintenance*, page 301, Bari, Italy, October 1997. IEEE Computer Society Press.

24 H.A. Müller, K. Wong, and S.R. Tilley. Dimensions of software architecture for program understanding. In *Proceedings of the International Workshop on Software Architecture*, Dagstuhl, Germany, October 1995.

25 S.R. Tilley and K. Wong. Software engineering education: A student vision. In *National Workshop on Software Engineering Education*, pages 155–156, Toronto, Ontario, Canada, May 1993. IBM Canada Laboratory.

Other Technical Publications

26 K. Wong. *The Reverse Engineering Notebook.* Ph.D. dissertation, University of Victoria, Victoria, British Columbia, Canada, 1999.

27 K. Wong. *Rigi User's Manual*, June 1998. Version 5.4.4, 168 pages.

28 J. Martin, J.S. Uhl, H.A. Müller, and K. Wong. Migrating PL/I code to C++: An experience report. Technical report, IBM Canada Ltd. Laboratory, Toronto, Ontario, Canada, August 1997.

29 K. Wong. An efficient implementation of Fortune's plane-sweep algorithm for Voronoi diagrams. Technical Report DCS-182-IR, Department of Computer Science, University of Victoria, Victoria, BC, Canada, October 1991.

30 K. Wong. Techniques for optimizing Fortune's plane-sweep algorithm for Voronoi diagrams. M.Sc. thesis, University of Victoria, Victoria, British Columbia, Canada, 1991.

31 K. Wong. Software engineering techniques used in designing the DCRSi support utility. Technical report, Department of Computer Science, University of Victoria, Victoria, BC, Canada, October 1988. Work Term Report for Macdonald-Dettwiler and Associates.

32 K. Wong. Better estimates of DS-0 errored second performance. Technical report, Department of Computer Science, University of Victoria, Victoria, BC, Canada, October 1987. Work Term Report for Bell Northern Research.

33 K. Wong. Service machines in on-line multi-user CMS applications. Technical report, Department of Computer Science, University of Victoria, Victoria, BC, Canada, February 1987. Work Term Report for BC Ministry of Finance.

34 K. Wong. The development of a preprocessor for the standardization of job control language. Technical report, Department of Computer Science, University of Victoria, Victoria, BC, Canada, June 1986. Work Term Report for BC Ministry of Forests.

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 thesis 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 thesis for financial gain shall not be allowed without my written permission.

Title of Dissertation:

The Reverse Engineering Notebook

Author: _____

Kenny Wong

December 24, 1999