A Customization Framework for the SVG Graph Visualization Engine

by

Feng Zou

B.Eng., Tongji University 1995

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

MASTER OF SCIENCE

In the Department of Computer Science

University of Victoria

A Customization Framework for the SVG Graph Visualization Engine

by

Feng Zou
B.Eng., Tongji University 1995

Supervisory Committee

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

Dr. William Wadge, Departmental Member (Department of Computer Science)

Dr. Bruce Kapron, Departmental Member (Department of Computer Science)

Supervisory Committee

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

Dr. William Wadge, Departmental Member (Department of Computer Science)

Dr. Bruce Kapron, Departmental Member (Department of Computer Science)

Abstract

The Rigi research group has built an SVG (Scalable Vector Graphics) Graph Visualization Engine to visualize, interactively explore and annotate software structures. The biggest difficulty we experienced during the customization of this engine for different domains is that we need to create idiosyncratic generators every time. Each generator is created by different developers using different approaches. If the subject information model does not contain layout information, developers of the generator are also responsible for writing algorithms to calculate layout.

In this thesis, we present a customization framework for our SVG Graph Visualization Engine to provide flexible customization using third party libraries to construct specific SVG Graph Generators. The customization framework consists of documentation for the existing graph engine and a componentized abstract generator that can be extended for a variety of information domains. We also created two reference

implementations for the abstract generator and included them in the template solution project to exemplify the usage of the customization framework. We also validated the template solution with an end user to build a generator for a new domain. Our customization framework greatly eases the process of building SVG Editor Generators for domain-specific visualization engine.

Table of Contents

List of Figures

Acknowledgements

I would like to thank my supervisor, Dr. Hausi Müller, for giving me the opportunity to pursue graduate studies under your supervision, and especially for your patience, support, guidance and encouragement throughout this research. I would also like to thank the committee members for taking the time to read and comment on my thesis.

A special thank-you goes to Shimin Li from University of Waterloo for the invaluable advice and help on this research. I am also grateful to all the members of the Rigi research group for their contribution to this research project. In particular, I would like to acknowledge the help I received from Holger Kienle, Qin Zhu and Yingyun Lin.

I acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) and the IBM Toronto Center for Advanced Studies (CAS).

Finally, I would like to thank my friends and family for helping me through this long process with your care and love. Mom and grandma, your love goes beyond space and time. And, to Yingyun and Aaron, thank you for your patience and understanding, and for filling my life with more than school and work.

Dedication

To my family

Chapter 1 Introduction

Software visualization can visualize information such as software architecture, software evolution history and software metrics [Die05, SDBP98, Zha03]. The field of software visualization has enjoyed tremendous research interest over the years [Kos03]. According to visualization researchers at the University of Bern, Switzerland, there are over fifty different commercial and research software visualization tools available [SCG07].

1.1 Motivation

Our research group has developed a special-purpose graph visualization engine built with SVG and ECMAScript to visualize, interactively explore and annotate software structures [W3C03, MWK03, KWM02]. Throughout this thesis this existing visualization engine is referred to as SVG Editor.

SVG, which stands for Scalable Vector Graphics, is platform independent and, hence, SVG documents can be used and displayed in many different environments. It also means that SVG Editor runs on many different platforms and applications. However, the current implementation of our graph visualization engine requires a specific generator for each information domain; as a result, it is difficult to re-target SVG Editor to visualize another information space.

Hitherto, SVG Editor has been used to visualize software structures generated by Rigi [Mül, MK88], ReGoLive [GKM05a, GKM05b] and EMF [EMF] models. The problem is that, for different domains, a specific generator has to be created in order to create an SVG Editor to visualize the information from the subject domain. Also, the generator is responsible for layout calculations. Since computing layout of graphs is not an easy task, the layout is calculated based on simple algorithms such as Breadth-First Tree. Also, there is no flexibility in changing or choosing appropriate algorithms for layouts. Thus, a worthwhile objective is to develop a customization framework that generates customized SVG visualization engines for various information visualization needs including a variety of domains and graph layout algorithms.

1.2 Approach

First, we investigate customization techniques to be able to tailor the engine to different visualization needs. Customization techniques applicable for our application include generative programming and OMG's MDA (Model Driven Architecture) [CE00, OMG]. Two recent University of Victoria theses have also investigated the customization

frameworks for information visualization. Bull's dissertation employed a model-driven engineering approach for information visualization [Bul08]. In his M.Sc. thesis, Michaud developed a software customization framework and introduced the three customization roles used in this thesis (i.e., Designer, Customizer and End User) [Mic03].

Before one can modify a program—let alone use the program to derive other programs, one must thoroughly understand the program. Since there was no detailed documentation available for SVG Editor, we had to reverse engineer it—that is, analyze the existing system and identify its components and the relationships between the components.

After gaining an understanding of customization and the existing system, we tried to determine the scope of the customization framework by answering the following questions:

1) What should be included in the customization framework?
2) What types of customization should be supported?

Once we clarified the scope of the research, we elicited requirements for our customization framework. To determine what should be included in the framework and to investigate useful types of customizations, we considered the different roles of framework users—Designer, Customizer, and End User.

For Designers, we need abstract architecture diagrams and detailed documentation of the existing SVG Editor to facilitate understanding and future enhancements. For Customizers, we need to develop a flexible approach to specify and create domain-specific SVG Editor generators. The framework solution should also guide the

Customizer in the process of generating customized SVG visualization engines for different application domains. Finally, End Users should be able to customize the look-and-feel of the visualization engine.

The actual implementation includes two reference SVG Editor Generator implementations—SVG Editor Generators for Java applications and organization charts. Furthermore, we provide a sample Eclipse project to exhibit the template solution and reference implementations.

We then evaluate the developed solution by checking whether it satisfies the requirements for the customization framework. We also validate the template solution project by letting an end-user build a generator for goal models using our customization solution.

1.3 Thesis outline

This thesis is organized as follows. Chapter 1 introduces the problem, motivates the customization framework for SVG Editor, and outlines an approach towards creating such a framework. Chapter 2 provides background information on customization frameworks, Scalable Vector Graphics, SVG Editor, and other related work. Chapter 3 discusses the scope and requirements for the customization framework. Chapter 4 describes the high-level architectural design of our customization framework and the design for the core component of our approach. Chapter 5 presents two case studies implementing generators for SVG Editor and integrates all components into an Eclipse project. Chapter 6 evaluates our customization framework and describes its limitations.

Chapter 7 summarizes the thesis, lists the contributions and outlines potential future work.

Chapter 2 Background and Related work

This chapter provides background information on important concepts and terms for the customization framework and discusses related work.

2.1 Background

In this section, we introduce relevant background information on the following topics: Scalable Vector Graphics (SVG), SVG Editor, customization techniques and frameworks.

2.1.1 SVG

SVG is a W3C standard and a language based on XML (eXtensible Markup Language) [W3C06] to represent two-dimensional vector graphics [W3C03]. A markup language is a set of annotations to text that give instructions regarding the structure of text or how it

is to be displayed. Markup languages have been in use for centuries, and in recent years have also been used for computer typesetting. XML is said to be extensible because it lets users define their own elements. Vector graphics uses basic geometric elements, such as points, lines, curves, circles, triangles and polygons, to describe two-dimensional graphics based on mathematical equations [FDFH97, p.9]. The SVG language incorporates three graphic object types: shapes, images and text. A program written in SVG is typically referred to as an SVG document and can be rendered on various platforms including web browsers. In contrast to HTML (Hypertext Markup Language) documents, SVG documents are scalable and zoomable without loss of details.

Since SVG is XML-based plain text, SVG texts are fully selectable and searchable. SVG content is enclosed in well-formatted XML tags with the root tag <svg>. An SVG document can be interpreted and rendered as a standalone, self-contained graphics file or be embedded in other documents such as web pages [Ado].

SVG provides declarative and scripting animations. Declarative animation elements are used to describe how document elements change over time (e.g., increase the radius of a circle over time to animate the size of a circle object). Scripting animation uses DOM (Document Object Model) scripting—SVG conforms to the DOM standard, which is a standard that defines document object models for HTML, XML or other similar formats. SVG DOM Scripting provides more flexibility than declarative animation and can be used for a variety of applications. For example, it can be used to program event-based user interaction. Developers can write functions to handle user events such as mouse and keyboard events. Besides animation, SVG elements can be

added, modified or deleted on the fly using a scripting language to access and manipulate the contents of the DOM tree structure. Therefore, we can produce highly dynamic and interactive SVG documents [W3C03].

SVG is platform independent and, hence, SVG documents can be used and displayed in many different environments. It also means that our graph visualization engine runs on many different platforms and applications. For example, the Adobe SVG Viewer [Ado] is available as a plug-in for browsers under Mac, Windows, Linux and Solaris. Selected Microsoft Office (e.g., Visio) and Adobe tools (e.g., Illustrator) are able to import and export SVG documents. Web browsers, such as Firefox and Opera, provide native support for SVG (i.e. they can render SVG without installing any extensions or plug-ins [Rus04]).

2.1.2 SVG Editor

SVG Editor is shorthand for the graph visualization engine developed by our research group. SVG Editor can be used as a software visualization tool to visualize, interactively explore, and annotate software structures. Users can not only explore objects in a graph by zooming and panning, but also select objects to view their attributes, group objects, change objects' location, color, shape, take snapshots, and so on [Lin08]. As depicted in Figure 1, the user interface of SVG Editor has five main components:

1) Main visualization window,
2) Menus,
3) History views,

4) Hierarchical view, and

5) RCL (Rigi Command Library) command input.

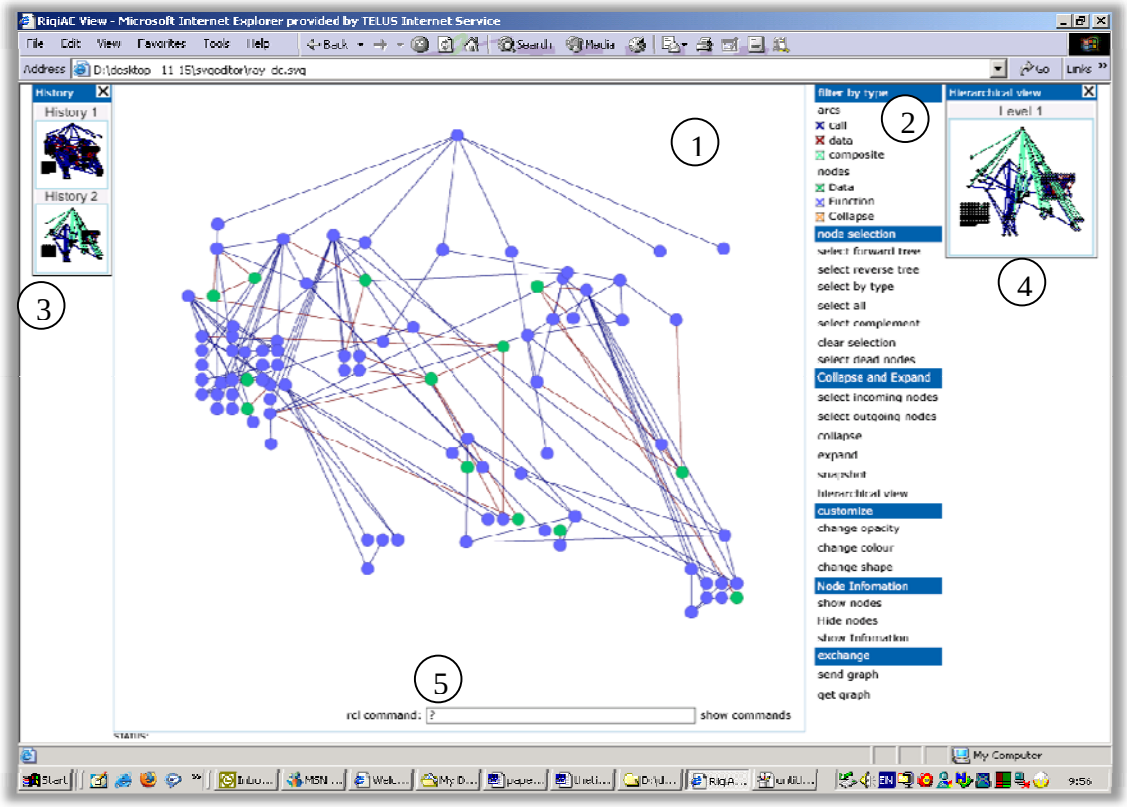

Figure 2-1: SVG Editor Interface

The main visualization window displays the node and arc information of a graph: nodes represent entities of the information system under study and arcs represent relationships between entities. The main window listens to mouse events so that users can perform a series of interactive activities on nodes, such as dragging, selecting, collapsing and expanding of nodes and arcs.

There are two types of menus: on-screen menus and pop-up menus. The on-screen menu window provides a set of command buttons so that users can click them to perform all kinds of tasks, including filtering certain node and arc types, selecting outgoing nodes. The pop-up menu window shows up when users right-click on the main window. It provides similar menus as the on-screen menus. In addition, it provides a menu item to invoke a control panel that has control buttons to pan the image in the main window, take snapshots for the main window and so on.

The history window looks like a filmstrip that keeps the snapshots of the main window taken by users. Snapshots can be added or deleted in the history window, and they are stored in time order of creation. Users can restore the state of SVG Editor to a specific point in history by clicking its snapshot.

Users can explore the information contained in a subsystem (i.e., a collapsed node) by selecting the hierarchical view command. The view from which users request detailed subsystem information will be added to the hierarchical view window, so that users can go back to it by clicking the specific hierarchical view.

SVG Editor provides another option for users to interactive with the graph information: RCL command line input. For each menu command, there is an RCL command created for automating tasks, such as *rcl_select_all*, *rcl_collapse* and *rcl_expand*. All available RCL commands can be displayed in a separate window by clicking the *show commands* button.

Since SVG is platform independent, SVG Editor is available to users under different platforms and environments for free. As mentioned above, SVG Editor has been

used with software reverse engineering tools such as Rigi and ReGoLive and EMF-based Eclipse software models. The following section provides more details on the custom SVG Editors for ReGoLive and EMF (Eclipse Modeling Framework); details about a custom SVG Editor for Rigi's RSF (Rigi Standard Format) format are discussed in Section 2.2.2 below.

2.1.2.1 SVG Editor for ReGoLive

ReGoLive is a reverse engineering tool developed by Grace Gui of the Rigi group and built as part of her M.Sc. thesis on top of the Adobe GoLive web authoring tool. It supports comprehension of Web sites that can be read by or have been developed with Adobe GoLive by offering three distinct viewpoints of the Web site under analysis: developer-view, server-view, and client-view [GKM05a, GKM05b]. REGoLive uses the SVG Editor to show each viewpoint, and also allows users to explore and navigate mappings between them interactively.

The process used to generate SVG Editor for REGoLive is as follows: fact extractors produce flat text files in GXL (Graph eXchange Languge) format. A generator implemented in JavaScript transforms GXL files into customized SVG Editors. The tree layout of the graphs for a particular SVG Editor is calculated by this generator using breadth-first tree traversal [Gui05].

2.1.2.2 SVG Editor for EMF

Eclipse is an open, universal tool platform that is designed to serve as the common basis for diverse IDE (integrated Development Environment)-based products [Ecl]. A plug-in

is a component that provides a certain type of service within the context of the Eclipse workbench. The Eclipse Modeling Framework (EMF) is a modeling framework and code generator for building tools and applications according to a data model description [BSM+03]. All EMF models adhere to a common meta-model, called Ecore.

The Eclipse SVG Visualization Engine plug-in generates a customized SVG Editor for EMF models by transforming an EMF meta-model description (Ecore file) into a suitable graph representation. Graphic layout is calculated by the generator. It first finds out parent-child relationships between all nodes, and then uses a simple tree layout algorithm to divide nodes into layers based on the hierarchical relationships. Finally it evenly distributes the nodes and assigns locations to the nodes [LZKM07].

2.1.3 Software Customization and Customization Techniques

According to Cambridge Dictionary, “customize” means “to change something to make it suitable for a particular person or purpose.” Customization means tailoring something existing to meet users’ special needs or requirements.

The field of customization has attracted a lot of research interest in disciplines such as Information System and Human-Computer-Interaction [YTH06]. There is some confusion about customization and personalization, an emerging research area that has received a lot of attention in the 21st century. Even when we search for “customization” in Wikipedia, the search result is redirected to the result of “personalization”. Based on the research through research profiling done by Sunikka et al. [SB08], personalization can be treated as an “umbrella” term which means both web personalization and

customization of tangible products. In the context of the World Wide Web, personalization has two meanings: when the personalization is initiated by the system, the term “personalization” is often used; and when the personalization is controlled by the users, the term “customization” is used instead.

2.1.3.1 Software Customization

Customization of software has mainly focused on providing configurable features to enable end-users to modify a subject system and studying how end-users modify a system. Mackay’s research shows that whether or not a user will make use of the customizable features does not depend on the availability of a list of customizable features—users tend to customize usage patterns, such as automating a process by a simple command [Mac91]. For example, in Microsoft Excel, users can define a Macro to record a user process and assign a shortcut key to the process. Later on, the process can be automated using the shortcut key. Mackay’s research also shows that unless a user has time to play with a system, the customization for the purpose of aesthetics is usually avoided. However, with the evolution of software systems, today’s users are becoming more and more used to software that can be tailored and personalized to satisfy their specific needs. As a result, it is common practice to include an options or preferences menu in applications for end-user customization.

Although customization has been discussed extensively, software customization is still an area that has received little attention in the literature (e.g., searching on the term returns no results in Wikipedia) [HLM03]. The Oksnøen Symposium 1992 was about software customization and triggered discussions on when, how and who should do

software customization; however, neither did it specify the notion of customization nor customization model clearly [OKS92].

Hui's paper presents a common customization model that includes analyzed goals of potential users for a specific system, cognitive skills users have and preferences for end-users [HLM03]. During the requirements analysis process, user goals, skills and preferences are analyzed and modeled to form the requirements specification and produce a set of tasks that fulfill the users' goals. Based on the requirements specification and the tasks, the software design process will produce the application that can meet user requirements in a finer-grained way. This model arguably forms a foundation for software customization from the perspective of end-users.

Michaud's M.Sc. thesis provides a broader view of software customization classifying software customization from different viewpoints [Mic03]. It categorizes three forms of customization based on what can be customized for software: data customization, presentation customization and control/behaviour customization. Data customizations can change both the content and format of data, and also the meta-data that describes data. Presentation customizations can change both the information architecture and graphical design. Information architecture determines how information is organized in a view and how users access information. Graphical design determines the look and feel of a software system. Control/behaviour customizations can change how a system behaves by selecting, adding or removing features, enhancing features, specifying options and coordinating features.

Michaud summarizes two categories of customization based on how to accomplish customization [Mic03]: customizations that affect source code and customizations that do not touch source code. Non source-code customizations are given the name interaction style customizations. They can be achieved through option screens, wizards, configuration files, macros, visual builders, scripts or application generators. Source code customization can be realized by applying object-oriented design, design patterns, components and frameworks.

Michaud also identifies three roles involved in the process of customization: *Designer*, *Customizer* and *End User*. Designers are responsible for the initial design and development of a software system. They determine the functional requirements of the system as well as the system support for source-code customizations and interaction style customizations. A Customizer is both a domain expert and a person who knows the system so that he or she can modify the original system to meet specific user needs. End Users of a system actively use the system. They can customize the look and feel of the user interface or perform other types of customization based on their skills and interest.

2.1.3.2 Software Customization Techniques

There are several different kinds of software customization techniques. For example, parameterization during system startup and configuration or modification and extension of an application's behavior through scripts and APIs. Among the various approaches available, Generative Programming [CE00] and OMG's (Object Modeling Group) MDA (Model-Driven Architecture) [OMG] are two areas that have drawn much research interest.

2.1.3.2.1 Generative Programming

Generative Programming [CE00] is a software development paradigm that is concerned with defining problem domain models in DSLs (Domain Specific Languages) for software system families and using generators to automate the assembly of standard software family components into the final customized software systems. For example, web authoring tools generate websites using domain specific languages such as HTML or XHTML. The goal of Generative Programming is that "programmers state what they want in abstract terms and the generator produces the desired system or component" [CE00, p.2]. Generators can employ different generative technologies: for example, program generation, model transformation or direct interpretation [CGN04].

Although the book focused on C++ template meta-programming as a concrete example [CE00], the International Summer School on Generative Programming [ISS02] gave participants an opportunity to learn how to use common technologies such as Java and XML, in addition to C++ template meta-programming, to implement generators.

Cutumisu et al. introduced a generative and adaptive process for generating game scripting code for game story authoring developers [COM+07]. In their three-step process, high-level abstractions of a common game scenario are captured in a generative pattern; then the abstractions are customized to a specific scenario using a set of standard adaption operations; and finally a generator automates the generation of the scripting code based on the adapted pattern. This is an attempt to apply generative design patterns, which can be automatically translated into executable code, for one of the most difficult content creation problems in the computer games industry.

2.1.3.2.2 Model Driven Architecture

Model driven architecture (MDA) promotes a model-centric approach to software engineering that shifts the focus of software development from code to model [OMG]. MDA uses a platform independent model (PIM) to specify requirements of a software system. A PIM is then transformed to one or more platform specific models (PSMs) using a transformation engine. Finally, a generator analyzes the PSMs and automatically transforms PSMs into running applications. The primary goal of MDA is to achieve platform independence, improve productivity, interoperability and maintainability [KWB03].

MDA and Generative Programming are both model based. Although MDA and Generative Programming use different terminologies, MDA can be considered a subset of Generative Programming. Essentially, MDA uses a particular technology of the generative approach, that is, model transformation to create applications from a platform independent model. Here is a useful way of describing these techniques at once: MDA realizes the generative approach and popularizes it in practice. For example, MDA has been applied in the construction of embedded software [KNS08], medical service software [MBK07], and IT solutions [KR06].

Bull's Ph.D. dissertation [Bul08] discusses a model based approach called model driven visualization (MDV) to design customizable information visualization tools. In addition to the platform independent data model, MDV includes several platform independent models for common viewers (e.g., Tree, Node-Link and Bar Chart). For each target platform, there is a concrete platform specific model that conforms to each of the

common viewers as described in the platform independent data model. In order to visualize instance data that conform to the data model in a particular viewer, the designer builds the transformation rules that describe what target elements in the viewer should be generated for each matching element in the data model. The view is then generated by applying the transformation rules to the instance data. MDV uses model transformation languages such as ATL (Atlas Transformation Language) [JABK08] to facilitate the transformation from data model to view model. MDV uses JET (Java Emitting Template) [JET] to generate platform specific models from platform independent models of abstract viewers.

The research reported in this thesis adopts the generator concept from Generative Programming and uses a model driven approach to create SVG Editors. Information under study is modeled in a platform independent model, and a generator translates the platform independent model into a customized SVG Editor. Section 4.1.3 discusses this process in detail.

2.1.4 Framework

The notion of a framework has many different meanings. For the purpose of software customization as discussed in this thesis we use the following definitions: From a structure perspective, *a framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact;* from purpose viewpoint, *a framework is the skeleton of an application that can be customized by an application developer* [Joh97].

Our customization framework for SVG Editor provides a reusable design of an SVG Editor generator that can be customized by a Customizer to generate specific SVG Editors for different information domains. Concretely, our framework consists of a set of abstract classes and reference instances.

2.2 Other Related Work

In general, visualization tools provide better usability if they have been customized for particular user needs. So far, there are few visualization tools that provide sufficient and effective customization support for creating tailorable interfaces [BF05].

2.2.1 MetaView

Software projects involve many activities: requirements engineering, high-level design, detailed designed, development, integration, testing and maintenance. The information produced by these activities about software projects (i.e., artifacts about the software system), is referred software documents. The MetaView system developed by Sorenson and his team was one of the first customization frameworks for software tools. Its goal was to generate the user interface for computer-aided software engineering (CASE) tools for a wide variety of software specification environments [STM88].

They divided the view-based MetaView architecture into three levels: the meta, environment and user level. At the meta-level, a meta-model EARA/GE is defined and some necessary generic software components are created. EARA/GE stands for entity-aggregate-relationship-attribute with graphical extension and is designed to model

a variety of software development methods. At the environment level, the required environment is configured by describing the environment definitions using an environment definition language and environment constraints expressed in an environment constraints language. Also, generic tools created at the meta level are configured to become specific tools for the environment. At the user level, the users develop a software specification using the configured tools. Users have different requirements on how to view the important software artifacts of interest and, thus, demand different views. This architecture provides a customization framework for building an integrated specification environment. The meta level provides facilities needed to accommodate the requirements of multiple views and the environment level deals with specific environments defined in terms of multiple views [Fin94].

2.2.2 Rigi

Rigi, developed by the Rigi group at the University of Victoria, is a reverse engineering tool and environment for program understanding and software analysis [MK88, MOT93]. Reverse engineering is the process of analyzing an existing system to identify its components and their interrelationships and to create representations of the system in another form or at a higher level of abstraction. The process of program understanding is the process of developing mental models of a software system's intended architecture, meaning, and behavior [Mül].

Rigi is a highly customizable and extensible research tool that can be tailored to accommodate different needs of software reverse engineering, exploration, visualization

and re-documentation [Til94, TMWW93]. Rigi has three important components: Rigireverse, Rigiserver, and Rigiedit. Rigireverse is the parsing system, Rigiserver is the underlying repository, and Rigiedit is the graphical user interface. The extracted information from Rigireverse is encoded in the Rigi standard format (RSF) exchange format, stored in Rigiserver, and presented and explored in Rigiedit.

Rigi provides a generic data model for Rigi graphs, which can be customized for various domains described by specification files. Users can define various configuration parameters to customize the Rigi environment. They can even design their own menus and widgets to customize the working environment for different visualization needs. Also, Rigi has built-in scripting capabilities. With Tcl/Tk scripting, users can extend Rigi to support their own specific requirements. Rigi comes with a library of over 400 predefined scripts as part of the Rigi command library (RCL) and is, thus, highly end-user programmable [TWS94]. Users can write their own commands for manipulating artifacts.

Rigi graphs can be exported into SVG Editors. Through Tcl, information contained in Rigi internal graph data structure can be accessed, manipulated and output to an intermediary Rigi view graph (RVG) file. This file stores graphic layout information about the nodes. An RVG file is then transformed to an SVG document using a Perl script [Lin08].

2.2.3 SHriMP

The simple hierarchical multi-perspective (SHriMP) environment comprises a set of highly successful and polished tools for visualizing and exploring software structures and

other information spaces [Sto]. Best discussed the reengineering of the SHriMP tool into a component-based framework that could be customized to explore information for different information spaces [BSM02]. The wrapper (i.e., data and display components) can be substituted with a new data model or graphical engine for a specific domain. Other components, categorized as support components, support wrapper components by providing particular services to them. Generic support components can also be replaced by specific ones for the domain under investigation. Through the framework of reusable components, SHriMP has been customized to visualize flow diagrams in Eclipse, integrated with the IBM Websphere Java development environment to achieve software visualization, and integrated with a knowledge management tool, Protégé [Pro], to achieve knowledge visualization.

Ernst proposed a customizable visual framework (CVF) and implemented the CVF in Jambalaya, the SHriMP plug-in for Protégé, to provide cognitive support in knowledge engineering [Ern04]. The techniques he employs are MDA and script-based environments. A Designer creates a model of the CVF for Jambalaya using an ontology—a formal specification of a conceptualization. The ontology can be modified by a Customizer to realize a domain-specific ontology. Customizers select a suitable initial layout, available layouts and action tools needed by the domain. Jambalaya accesses the model, as specified in the ontology, and loads the tool according to the options a Customizer states. Also, a Customizer can create customized scripting actions for End Users to perform functions they may need.

2.2.4 GMF

The graphical editor framework (GEF) is a general-purpose graphical editing framework that can be customized for specific application models [GEF]. The resulting diagrams can be exported as SVG graphs. Its customization, however, requires the developer to write a lot of code. Also, exported SVG graphs are static—users can use built-in SVG support for panning and zooming, but domain-specific interactions, such as filtering and finding outgoing nodes, are not available.

The graphical modeling framework (GMF) project provides a generative approach for a domain model expressed in EMF to add diagramming capabilities where a visual editor is desired [GMF]. Essentially, GMF forms a generative bridge between GEF and EMF. GMF is an important project that adopts MDA. As depicted in Figure 2-2: GMF uses models to define diagram editors from different aspects: domain model, graphical definition model and tool definition model. A mapping model binds the graphical and tool definition models to the domain model. The mapping model is then transformed to a generator model—a set of parameters for templates used to generate diagramming code. GMF Runtime is a set of frameworks that help the development of graphical editors. It contains reusable components for editors such as common menu commands and toolbars. It uses a standardized model for describing diagram elements. Also, it provides a set of extensible services by means of declared extension points at execution time, which enable editors to be extended and customized by third-parties. [VKEH06]

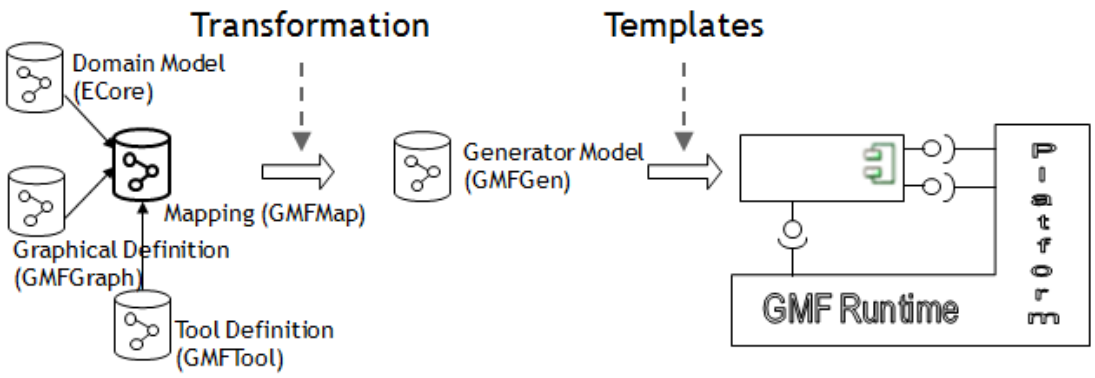

Figure 2-2: High-Level Architecture for GMF

Courtesy: Introduction to the Graphical Modeling Framework by Artem Tikhomirov and Alexander Shatalin, Borland Software Corporation [TS08]

2.2.5 Other Approaches

We are not aware of any customizable visualization engines implemented in SVG at this time. Although Domokos and Varró introduced an open, SVG-based visualization framework for modeling languages defined by meta-modeling techniques [DV02]. Their goal is to use this framework to obtain the SVG visualization of some concrete models such as Petri nets. In essence, the resulting SVG outputs do not differ from the SVG graphs exported from GEF. Users have access to standard SVG features such as panning and zooming, but domain-specific actions such as filtering and layout are unavailable. This framework does not handle user interactions either. In contrast, our customization framework can produce customizable SVG Editors that support user interaction.

2.3 Summary

This chapter presented background information on SVG and SVG Editor. It also introduced the notion of customization and software customization and discussed two important and popular software customization techniques: Generative Programming and MDA. Furthermore, it described related work on customization frameworks, such as MetaView, Rigi, SHriMP and GMF. The following chapter reverse engineers the existing SVG Editor and specifies requirements for generating customizable SVG editors.

Chapter 3 Understanding SVG Editor

Before we can even begin with the design for generating customized SVG Editors, we need to analyze and reverse engineer the existing SVG Editor in great detail. In this chapter, we document the existing implementation of SVG Editor. We then describe specific components of SVG Editor to form the foundation of our customization framework. Furthermore, we outline customization types to be supported by our customization framework. Finally, we specify requirements for the different components of the customization framework.

3.1 Reverse Engineering SVG Editor

In order to provide a customization framework for SVG Editor, the first step is to understand and document it using reverse engineering—the process involves analyzing the existing system to identify its components and relationships among those components.

In addition to the high-level understanding, we document the artifacts of the existing system in sufficient details to be able to generate SVG Editors for various applications.

The source code of SVG Editor can be divided into two components: (1) the SVG file that describes nodes and arcs of the information model under investigation, and (2) the script component composed of five ECMAScript files.

3.1.1 Main Graph SVG Document

A main graph SVG document contains all the information for rendering a particular graph (i.e., a set of nodes and arcs) using SVG. A rendered graph and part of its SVG source are depicted in Figures 2-1 and 3-1. As shown in Figure 3-1, this information includes the nodes' names, types and initial two-dimensional positions. It also describes the arcs' types and their source and destination nodes.

```
node1 = model.createNode("1", "Function");
Node.setAttribute(node1, "name", "mylistprint");
node2 = model.createNode("2", "Function");
Node.setAttribute(node2, "name", "main");

arc1 = model.createArc("1", "call", node2, node1);
var nodeview = view.nodeview;
nodeview.setInitialLocation("1", 530, 150);
nodeview.setInitialLocation("2", 450, 80);
```

Figure 3-1: Part of Source for a Main Graph SVG Document

In such a main graph SVG document, the <*defs*> tags define symbols for different node types. An SVG document is linked to another five external script files using <*script*> elements. The sixth <*script*> element defines a function *createDomain()* to create the domain model by defining the node types, arc types and their associated colors. It also defines a function called *createGraph(graph)*, which 1) creates a graph model that adds nodes, sets the attributes such as name for nodes, adds arcs between nodes, and 2) creates a graph view that defines the locations for nodes and adds node labels.

Finally, the body of the main graph SVG document defines the size and location of the graph main window and the location of the menu window.

3.1.2 ECMAScript Files

There are five ECMAScript files that provide the interactive capabilities for SVG Editors: *command.es, components.es, core.es, menuMaker.es* and *rGraph.es.*

Command.es defines all sorts of graph operations supported by SVG Editor. It contains approximately 2,000 lines of code defining commands including *select, select all, select dead node, select node by type, select forward tree, hide node, show information for selected node, collapse, expand, zoom, pan,* and *history.* Figure 3-2 depicts the hierarchical relationships between commands.

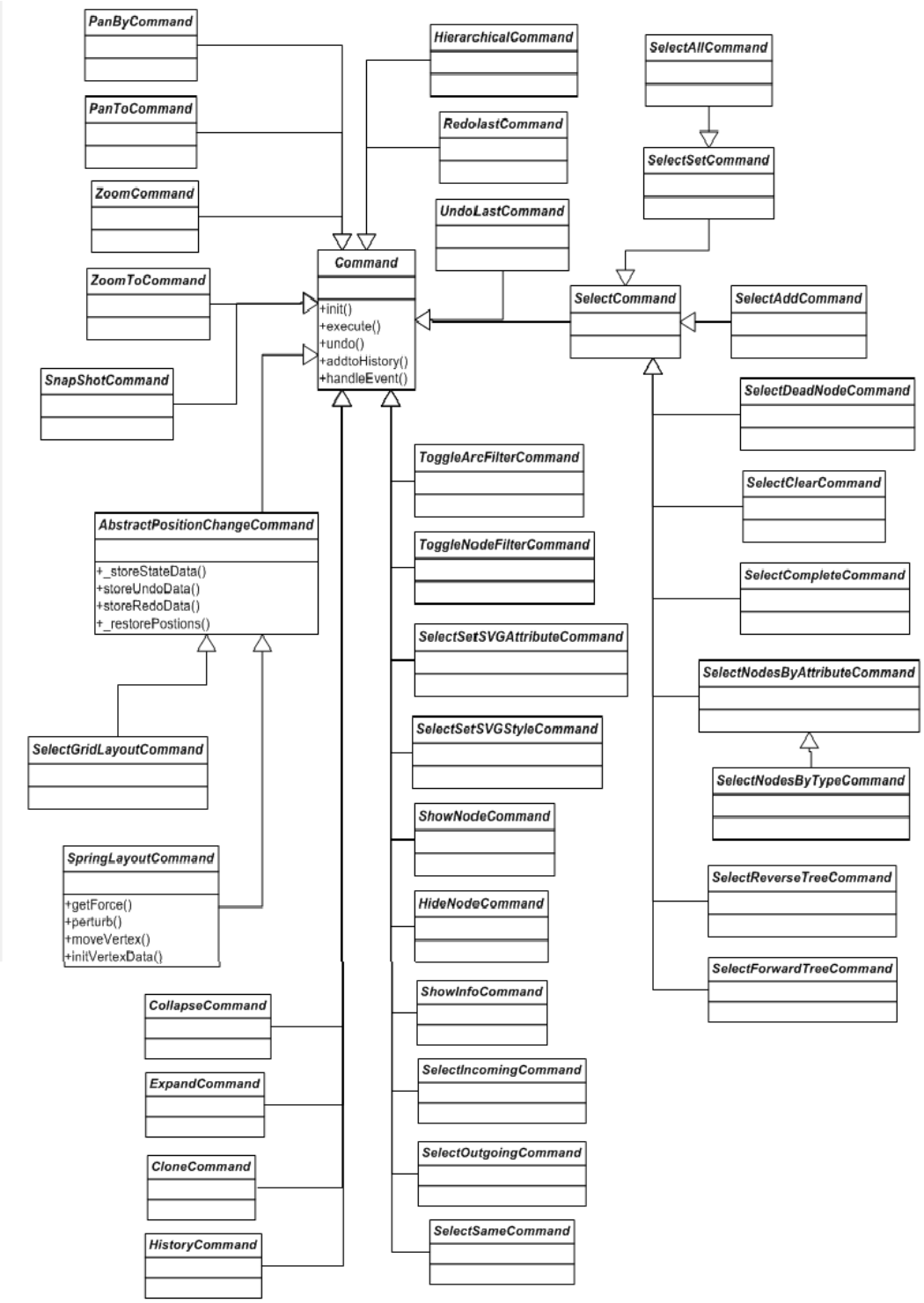

Figure 3-2: Hierarchical Relationships between SVG Editor Commands

Components.es defines visual components contained in SVG Editor. It contains approximately 1,500 lines of code defining all components included in the SVG Editor, including *container*, *composite button*, *checkbox*, *textbox*, *menu*, *submenu*, *label*, or *window*. Figure 3-3 depicts the hierarchical relationships between these visual components.

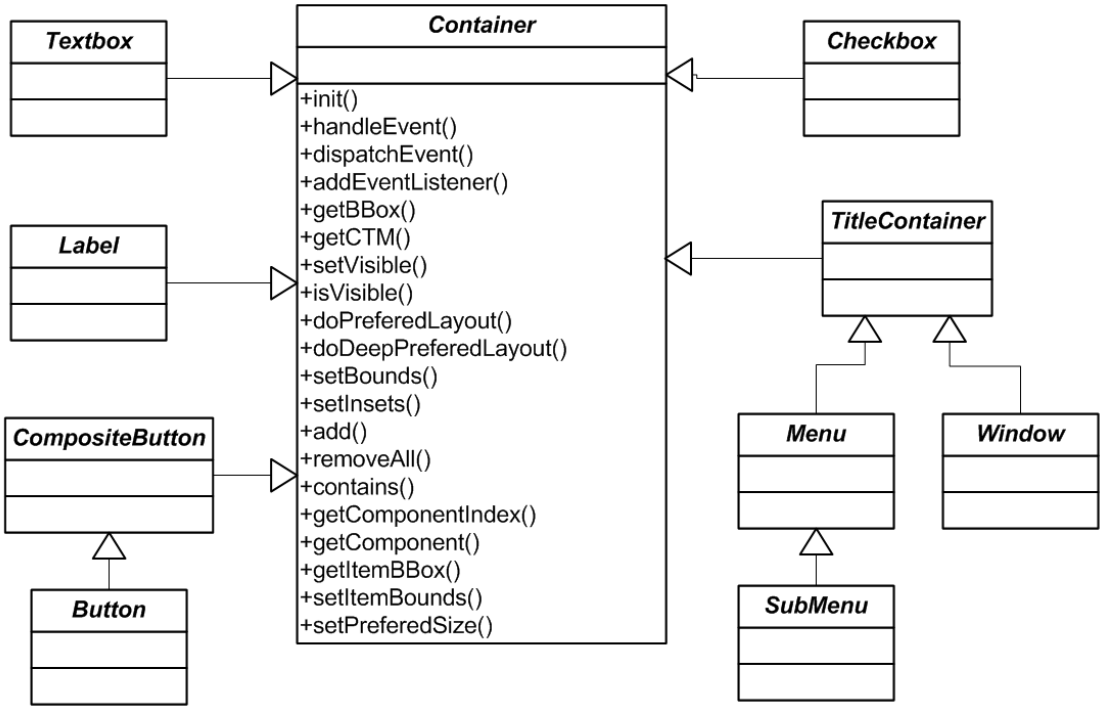

Figure 3-3: Hierarchical Relationships between SVG Editor Visual Components

MenuMaker.es contains only 58 lines of code defining a customized pop-up menu for SVG Editor.

RGraph.es defines graph model and views. It contains approximately 2,000 lines of code to define a graph with a graph model and possibly different kinds of views, including a default graph view that sets up SVG to host the graph, and provides utility

functions to return a view node based on an SVG node element, or return a view arc based on an SVG arc element. It also handles mouse events and contains a *NodeHandler* and an *ArcHandler* which act as flyweights for nodes and arcs, respectively [GHJV95, p.195]. It also defines many inner classes, including *DefaultNodeSelectionHandler, NodeSelectionHandler, Rubberband, NodeView, ArcView, NodeLabels,* and *NodeDragger.*

Core.es defines global variables and functions. It contains approximately 2,600 lines. The main method *DoOnLoad(event)* is called when the main graph SVG document is loaded. It creates the graph object for this document and initializes all components of the SVG Editor, including the *menu window, history window, hierarchy window,* and *node property window*. *Core.es* also defines other functions for manipulating the components created (e.g., methods to add history to and remove history from the history window as well as display node properties in the node information window). The script also defines an *EventListenerManager* to keep track of all the event listeners, to notify registered listeners of model changes, and modify the corresponding view [LZKM07].

3.1.3 Relationships among Components

In the previous two subsections, we introduced the main graph SVG document and five ECMAScript files. In this subsection, we summarize the relationships among them.

The main graph SVG document contains all the information for rendering a particular graph (i.e., a set of nodes and arcs) using SVG. As depicted in Figure 3-4, it directly connects with *Core.es* by calling its main method *DoOnLoad(event)* so that all

components in SVG Editor can be created. *Core.es*, as the main class of the scripts, creates instances of all other classes, and then assigns them to global variables such as the *RGraph* object *graph* for use by the SVG document.

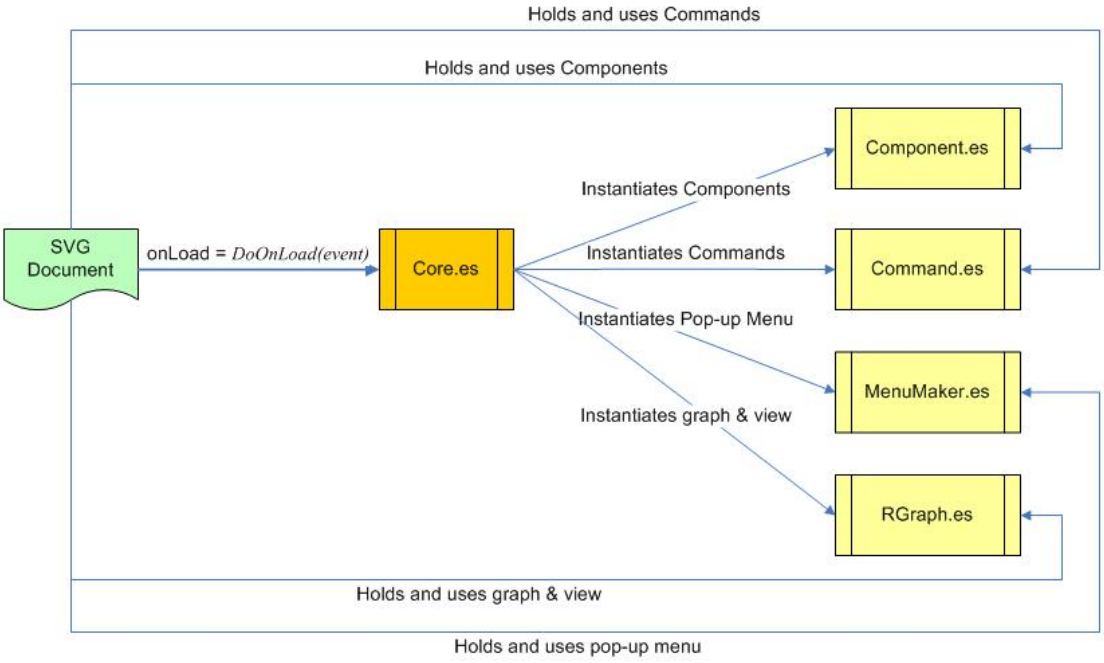

Figure 3-4: Relationships among SVG Editor Components

3.1.4 Existing Customizable Features

The existing SVG Editor provides capabilities to customize three features for nodes: opacity, color and shape. Next we describe how these customizations are realized through the function *createCustomizeMenu (menupane, submenu_root*) defined in *Core.es*.

- **Change opacity:** Create a submenu for controlling node opacity and add a list of buttons to the submenu. Each of the buttons is assigned a value between 0 and 1,

and has an event listener which listens to the button selection event. When a button is selected, the opacity style of the currently selected nodes is changed to the value assigned to this button.

- **Change color:** Create a submenu with a grid layout for controlling node color and add a list of buttons to the four-row submenu. Each of the buttons is assigned a color (e.g., red) and has a button selection event listener attached. When the button is selected, the color of the currently selected nodes is changed to the button color.
- **Change shape:** Create a submenu with a grid layout for controlling node shape and add a list of predefined shape buttons to the submenu. The shapes are not defined in the function, but defined by the <*symbol*> elem*ents in* the <*defs*> element of the SVG document. A <*symbol*> elem*ent can* include the definition of an SVG shape circle or a shape defined by an SVG path. It can even be an image (e.g., a .jpg file). Figure 3-5 illustrates the source code for adding an image to SVG *symbol* definition.

```
<symbol overflow="visible" id="flag">
  <image xlink:href="ca.jpg" x="-18" y="-12"
        width="36" height="24" />
</symbol>
```

Figure 3-5: Add an Image to SVG *Symbol* Definition

When a shape button is clicked, the button selection event listener is called and the shape of the currently selected nodes is changed to the selected shape button.

3.2 Functional Requirements for Customization Framework

Since we now have an appreciation for the benefits of customization as well as an understanding of the existing implementation of the SVG Editor, we are ready to identify the scope and specify the requirements for our new customization framework.

3.2.1 Components of Customization Framework

In Chapter 2, we introduced three roles people involved in software customization: Designers, Customizers and End Users. For the remainder of this thesis we use these roles to delineate customization tasks and to describe components of our customization framework:

1. The Designer of a customization framework may be different from the developer of the SVG Editor. One important component of the customization framework is the detailed documentation about the SVG Editor contained in Section 3.1.
2. To make the customization framework practical for different domains, as a Developer we provide a generic approach and create a generalized template (skeleton) for the Customizers so that they can modify it to fit their information needs. The template includes fixed components as well as stubs that Customizers need to substitute with the real information. To make the template easier for Customizers to understand and modify, two reference implementations are also included in customization framework in addition to the instructions on how to use the template.

3. End Users of customization framework provide instance XML data and run the specific generator to obtain a customized SVG Editor. End Users apply the customization framework to determine the colors for entity and relationship types and pick the initial layout algorithm. They are also the end-users of the generated customized SVG Editor. Although RCL command is an existing component of SVG Editor, the current customization framework does not support the customization of the end-user programming capabilities of SVG Editor. Of course, one could easily restrict the set of RCL commands available for different types of end-users. However, it is difficult to specify reasonable, useful and practical subsets.

3.2.2 Types of Customization

What type of customization should be supported by our customization framework? Data customization is needed by Customizers to generate visualizers for a variety of graph information domain models. Presentation customizations are needed by Customizers to setup the initial layout of the SVG Editor and for End Users to change the look and feel of the system. Control/behaviour customizations will be supported for Designers to add features to the system and for Customizers to select and remove features from the system.

How will customization be achieved and realized? Certainly, the customization framework must support source-code customization since this type of customization is performed by Designers. Designers can add or modify features by modifying code in appropriate locations. Customizers also perform some degree of source-code

customization by filling in stub contents by the Developer in specified components. Generator customization (i.e., interaction style customizations) will enable Customizers to provide domain model information about the information represented in a generated SVG Editor. Special generators built by Customizers enable End Users to determine what features to exclude in a customized SVG Editor and specify color or shape for entities and relationships types supported by the special domain.

3.3 Non-Functional Requirements for Customization Framework

Different requirements are needed for different components of our customization framework:

- **Reusable documents**: For Designers, the SVG Editor documentation should be a good resource. Developers should be able to find out what kind of source code change is necessary by looking at the document without digging into the source code. For Customizers, there should be some type of user manual to document the general approach to creating a special SVG Editor generator.
- **Componentized and reusable template**: For Customizers, the template should be componentized, reusable and extensible [FS97]. Components make the template easy to understand and easy to modify by Customizers.
- **Generic**: Data customization is targeted to different information models and the customization framework should guide the Customizers to generate customized

SVG visualization engines. The template should be general enough and tailorable to fit various information models represented in some special formats (e.g., in an XML format with an XML Schema or DTD or in a three-tuple format involving source, destination, and relationship).

- **Maintainable**: Our customization framework for Customizers must be easy to maintain, which means that it must provide sufficient source code documentation and also other relevant information.
- **Easy to use end-user customization**: End Users customization needs have evolved significantly. Only 16 years ago, users tended to customize “patterns of behavior”, such as automating a process by a simple command. The customization had no direct connection with the availability of customization features [Mac91]. Nowadays End Users are used to software that can be personalized, and are willing to use customizable features [Pri]. Thus, end-user customization should allow End Users to personalize the look-and-feel of a customized SVG Editor. The customization framework should provide features so that users can easily configure personal preferences (e.g., color scheme).

3.4 Summary

This chapter documented the existing SVG Editor in sufficient detail to facilitate the development of a customizable framework for it. We also discussed the scope and requirements for the customization framework.

Chapter 4 Architecture and Template Solution Design

This chapter describes the high-level architectural design of our customization framework and discusses the design for our template solution.

4.1 Architectural Design

A customization framework should be componentized for ease of adoption, maintenance and evolution. Figure 4-1 depicts a high-level design of our customization framework. The core component is called the SVG Editor Generator. It takes a list of XML files as input and uses a graphic layout framework to compute the layout for the generated output—the customized SVG Editor.

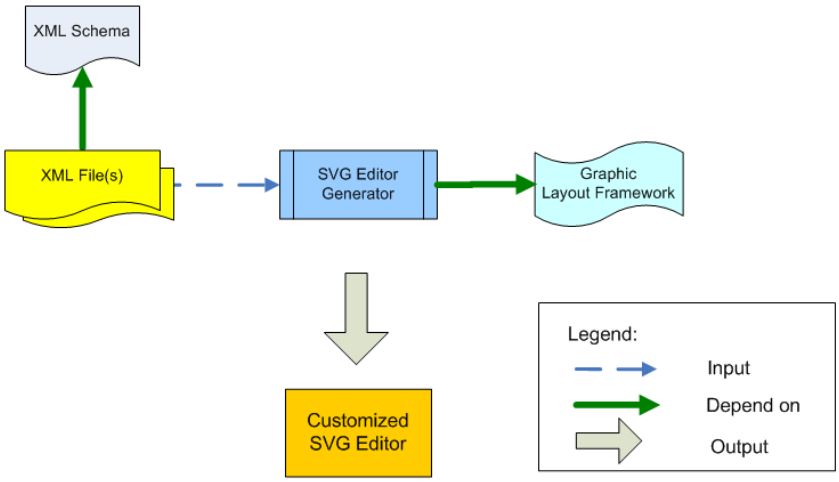

Figure 4-1: High-Level Design of our Customization Framework

4.1.1 Input Format for SVG Editor Generator

The input to our SVG Editor Generator is a set of files formatted according to an XML Schema. The XML Schema definition language provides a type system for XML documents. Similar to the traditional object-oriented relationship between classes and objects, an XML document that complies with an XML Schema type is often referred to as an instance document [Sko03].

The reasons for picking XML as the carrier to represent the input information domain are as follows:

- XML is a general-purpose markup language.
- XML is plain text; therefore, it is platform independent.
- XML is extensible: since it allows users to create their own tags, any information that users want to represent can be described in XML.

- XML separates content and presentation: it stores only content, no presentation information such as layout; therefore it is suitable for storing domain model information.
- XML is an effective format for being a source for multiple outputs: based on an input XML document, users can easily transform it to multiple target XML formats using XSLT (eXtensible Stylesheet Language Transformation) [W3C99] or other techniques.
- XML has been heavily used for data exchange/storage format: for example, XML is the language for describing web services, as used in WSDL (Web Service Definition Language) [Cer02]. Also, XML is a type in Oracle database systems for storing information [Sch01].
- XML has been used in configuration files (e.g., `struts-config.xml` in the Struts framework [HDFW02], an open source framework for building web-based applications).

The building blocks and structure of an XML document are defined by either a document type definition (DTD) or an XML Schema [W3Sa, W3Sb]. The notion of XML Schema is newer and more powerful. Thus, it is advantageous to use XML Schema definitions instead of DTD definitions for graph domain descriptions:

- DTD definitions are not expressed in XML, whereas XML Schema definitions are XML documents can therefore be validated and edited using XML editors (e.g., XMLSpy [Alt]).

- The DTD language supports limited number of data types (e.g., CDATA, PCDATA, ID), whereas XML Schema supports many more and allows complex type hierarchies.
- DTD does not support namespaces whereas Schema does. With namespaces, XML Schema definitions can be refactored into multiple files and namespaces, which can be included and/or imported to increase reuse and simplify maintenance [Sko03].

An XML Schema defines:

- Elements that can appear in a document
- Attributes that can appear in a document
- Which elements are child elements
- The order of child elements
- The number of child elements
- Whether an element is empty or can include text
- Data types for elements and attributes
- Default and fixed values for elements and attributes

In summary, XML with XML Schema domain descriptions provides significant flexibility for representing many different information models.

4.1.2 Graph Layout Frameworks

A customizable graph visualization engine clearly needs a flexible and tailorable graph layout framework to accommodate the layout needs of different information and application domains. The layout framework should support a variety of existing and common graph layout algorithms, be extensible to support the integration of new or modified layout algorithms, and be compatible with the technologies used in our implementation (i.e., XML, XML Schema, SVG, ECMAScript).

There are many graph layout libraries available and many of them are proprietary (e.g., yFiles [yWo], aiSee [aiS] and Tom Sawyer Layout C++ / Java [TOS]). Moreover, the free layout packages are often unsupported (e.g., ffGraph—a C++ layout library). Fortunately, we found two solid open-source layout frameworks: Java Universal Network/Graph framework (JUNG) [JUN] and Zest [BBS04, CHI], which are more complete and versatile than most frameworks. Both of them are written in Java, the former uses BSD (Berkeley Software Distribution) license and the latter uses Eclipse Public License. JUNG is a powerful framework for working with graphs with rich attribute structure [JUN]. It provides a few layout algorithms, including spring layout, tree layout, radial layout, balloon layout. More algorithms are added to the framework continually. Zest is discussed in more detail in Section 4.2.3 below.

4.1.3 SVG Editor Generator

SVG Editor Generator is the central component of our customization framework and corresponds to the notion of generator as described in Generative Programming and

MDA [CE00, OMG]. Thus, SVG Editor Generator has the following functionality and responsibilities:

- Read the data from XML files; retrieve the platform independent information model stored in an XML file and specified by XML Schema definitions; interpret the data using the Schema definitions; and extract entities and relationships between entities from data file;
- Calculate the layout for the extracted entities and relationships using layout algorithms of the graph layout framework; and
- Automatically generate a customized SVG Editor to explore and manipulate the graph structures (i.e., nodes and arcs representing the entities and relationships as described in the information model).

4.2 Design of Template Solution

Having outlined the high-level architecture of our customization framework, we can now describe the design of our template solution. Our template solution is a software template containing reusable components and stubs that must be substituted with real implementations to generate SVG Editors for different subject information models.

4.2.1 Java as a Language for Template Solution

We choose Java [Sun08a] as the programming language for constructing our template solution. Java is a platform independent object-oriented programming language. Since its release in 1995, Java technologies have become ubiquitous. In the opening keynote of the

2007 JavaOne conference, Sun estimated there are over 6 million Java developers world-wide. Also, Sun open-sourced the core Java code as of May 2007. Not only is the core Java source open-sourced, but also a lot of other third-party Java libraries are available as open source (e.g., the Apache Commons libraries [Apa08a]).

Another important reason for choosing Java as the preferred language for developing our template solution is that we are more familiar with Java than any other programming language. Given the abundant Java-related resources, coding in Java is rather convenient and pleasant.

4.2.2 Apache XMLBeans

Assuming that we use Java for our implementation, we need to retrieve information from XML input files using Java. Although the phrase “Java and XML—portable code and portable data” has been a familiar slogan since Java developers started using XML, developers have experienced difficulties dealing with these technologies since manipulating XML content is different from handling Java objects [ME06].

A lot of libraries are available for parsing XML files (e.g., Apache Xerces [Apa05] or Oracle XML parser [Ora]). Basically, there are two types of XML parsers: SAX (Simple API for XML) parsers and DOM (Document Object Model) parsers. Both aforementioned parsers provide implementations for SAX and DOM compliant parsers. SAX parsers use a “callback model” to interact with the code users write and require users to provide implementations for handler interfaces defined in the SAX API. For example, *ContentHandler* and *ErrorHandler* are callback methods to process XML

elements they parse and then give users a piece by piece view of an XML document [ME06]. In order to use an SAX parser, users have to write a significant amount of code to implement interfaces for all kinds of handlers.

In contrast to SAX parsers, DOM parsers parse an XML document, retrieve information and store it in a tree-based domain object model to provide a complete view of an XML document in memory. However, to retrieve the underlying XML information from a DOM tree, a lot of code has to be written.

Apache XMLBeans eases the use of XML with Java significantly [Apa08b]. It is a technology for accessing XML by binding it to Java types. A common usage of XMLBeans is to compile an XML schema to generate a Java Archive or JAR file that contains Java interfaces and classes representing schema types. Using the generated JAR file, users can access and modify XML instance data based on the structure described in the XML schema. For example, if there is an element “customer” in the XML schema, the JAR file compiled by XMLBeans will contain a Java class called Customer.

The steps on how to setup a programming environment to work with XMLBeans is available in Appendix A, which also includes the details on problems encountered during the setup process and how to resolve them.

Not every XML document has an associated constraint model to define its document type. A tool called *Relaxer* comes in handy just for this kind of cases. It allows to generate an XML Schema based on a set of XML instance documents. The more XML instances are provided, the more accurate the generated XML Schema will be. The command for generating an XML Schema is as follows [ME06]:

```
relaxer –xsd [input xml files 1… n separated by space]
```

Figure 4-2: *Relaxer* Command for Generating an XML Schema From XML Files

Since there are many XML files that are constrained using DTDs, there is a need to convert a DTD file to XML Schema constraint model. An open-source tool called *DTD2XD* can be used to accomplish this. The following command converts a DTD file to an XML Schema [ME06]:

```
java dtd2xsd [DTD file] > [output XML Schema file]
```

Figure 4-3: *DTD2XD* Command for Converting a DTD File to an XML Schema

Thus, it does not matter whether XML files have a constraint model or not, or whether the constraint model is written in DTD or XML Schema, we can always provide an XML Schema file for most XML files to be used as input to our SVG Editor Generator.

Figure 4-4 depicts XML Schema Compiler—the graphical user interface for compiling an XML schema file into a JAR file using XMLBeans (i.e., to specify the XML Schema input file to be compiled into a JAR output file). If the input passes the validation, the appropriate JAR file is output.

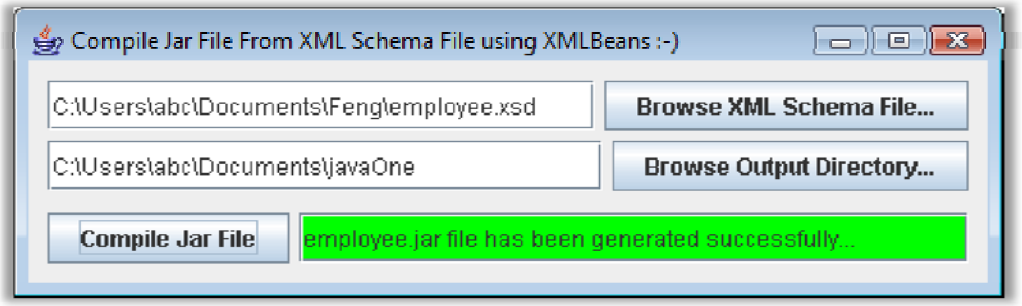

Figure 4-4: Widget for Compiling an XML Schema file into a JAR File

4.2.3 Graph Layout Package Zest

Zest, the Eclipse Visualization Toolkit is a set of visualization components built for Eclipse. The entire Zest library has been developed in SWT / Draw2D and integrates seamlessly within Eclipse [CHI]. Zest contains a layout package that includes a set of example layouts, including horizontal tree layout, vertical tree layout, radial layout, spring layout, grid layout. The Zest layout package can be used independently. By implementing node and edge interfaces, any graph model can be adapted to use this package. Zest also provides sample Java Swing [Sun08b] applications to show how layouts can be reused easily. According to Bull's dissertation, Zest uses existing layout algorithms from Jung and SHriMP, and is distributed under the Eclipse Public License agreement [Bul08]. Thus, the Zest layout algorithms are ideal for our Eclipse-based template solution to calculate graph layouts for our generated SVG Editors.

4.2.4 SVG Editor Generator

The SVG Editor Generator has three main components:

1) SVG template,
2) ECMAScript files, and
3) Extensible abstract Java classes.

These components are all part of the template solution.

4.2.4.1 SVG Template

The SVG template solution includes components obtained from disassembling the SVG document of an existing SVG Editor. The following components are assembled with the SVG Editor Generator to form the SVG document of a new customized SVG Editor:

- `head.svg:` This component contains the first part of the SVG document. It includes a substitute string for the title of SVG Editor whose value will be provided by Customizers through the Generator's GUI. It also contains the *`createDomain()`* method, which defines the types of the nodes and arcs used in the subject information domain. As domain experts, Customizers can manually define it, or automate its generation using the default template method in the Generator Java class. The following code fragment shows how to create a sample domain:

```
function createDomain() {
  var domain = new Domain("javaAppSVG");
  var type = domain.addNodeType("Class");
  type.setAttr("rgb", "rgb(102,102,255)");
  type = domain.addNodeType("Package");
  type.setAttr("rgb", "rgb(255,0,0)");
  type = domain.addArcType("containment");
  type.setAttr("rgb", "rgb(99,179,244)");
  return domain;
}
```

Figure 4-5: ECMAScript Code Fragment for Domain Creation

- `middle.svg:` This component contains the second part of the SVG document, which creates a graph view and sets relevant view attributes (e.g., the radius of the node drawing used in the view). It also contains width and height substitute strings to customize the dimensions of the main window.
- `rear.svg:` This component renders the view and finalizes the SVG document.

4.2.4.2 ECMAScript Files

The ECMAScript files of the SVG Editor Generator are the script files introduced in Chapter 3.

4.2.4.3 Abstract Java Classes

There are four abstract Java classes created for the SVG Editor Generator. The Generator class `SVGEditorGenerator.java` and the abstract Generator GUI class `SVGEditorGeneratorGUI.java` are super classes that any particular Generator or Generator GUI class must extend, respectively. `DomainModel.java` is the super class

that any type of a specific domain class must extend to provide methods to retrieve entity and relationship types included in the domain. `EntityNode.java` is the super class that any type of entity nodes must extend to provide constructors.

Through the Generator's GUI, a Customizer can configure the following parameters in the Generator class template to generate an SVG Editor:

- The input folder of XML files,
- The target output directory,
- A name for SVG Editor,
- A title for SVG Editor,
- A default size of the main window of SVG Editor,
- A default configuration of the popup menu,
- Color schemes for node and arc types, and
- A default layout algorithm.

Given a specific set of template parameters, an instantiated Generator GUI requests a specific Generator to generate a customized SVG Editor. Figure 4-6 illustrates the generation process. The specific Generator

1. Analyzes the input XML files to extract entities and relationships topology from the input XML files;
2. Computes the layout for entities according to the selected layout algorithm from the Zest layout package;
3. Annotates in `middle.svg` the entities and relationships topology with the computed coordinate information;

4. Generates a customized SVG Editor by gluing the components of SVG template together to form a complete view.

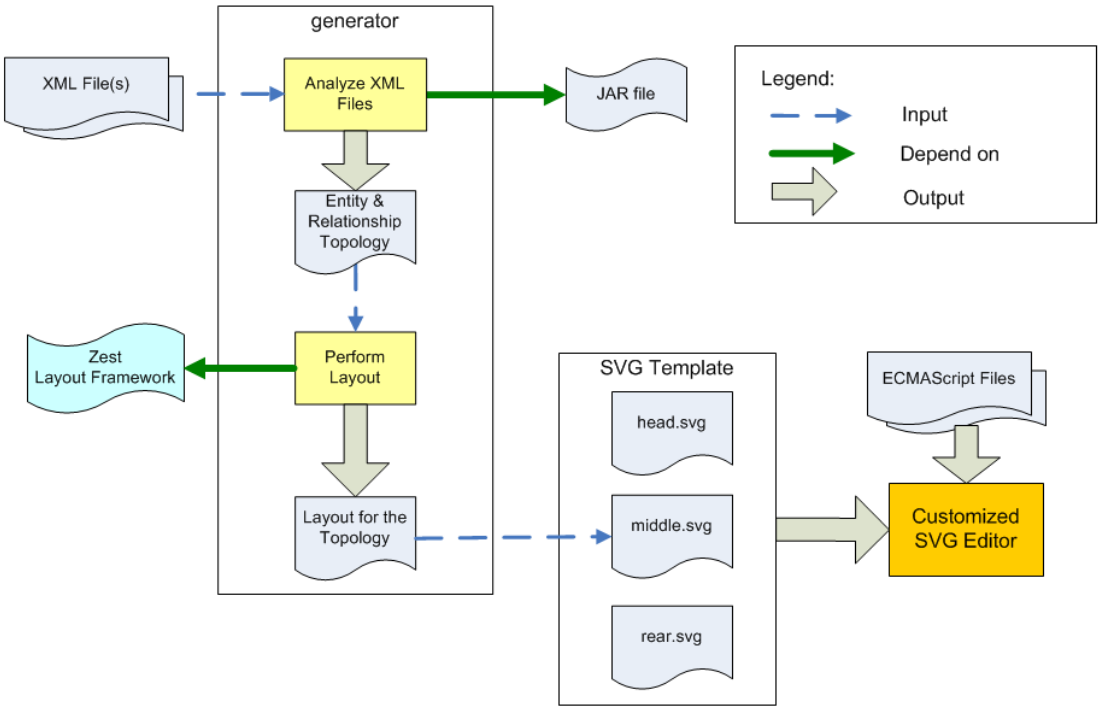

Figure 4-6: Generation Process of SVG Editor Generator

The `SVGEditorGeneratorGUI.java` class defines constants for Generator GUIs (e.g., default dimensions for the SVG Editor main window or warning messages). It also provides default behaviour for constructing graphical user interfaces by defining folder-browsing buttons, text input fields, and the SVG Editor Creator button. The following code fragment shows the abstract methods declared in the `SVGEditorGeneratorGUI.java` class that its subclass must provide to implementations:

```
public abstract class SVGEditorGeneratorGUI {
  // other fields and methods are omitted here …
  // generate SVG Editor; it passes the parameters to
  // the specific SVGEditorGenerator class to
  // generate an SVG Editor
  public abstract void generateSVGEditor (
    String inputDir,
    String outputDir,
    String editorName,
    String title,
    int width,
    int height,
    boolean useCustomPopupMenu,
    LayoutAlgorithm layoutAlgorithm,
    List<Color> entityColors,
    List<Color> relationshipColors,
    DomainModel model);

  // returns the domain model associated with Generator
  public abstract DomainModel getDomainModel();
}
```

Figure 4-7: Abstract Methods Declared in `SVGEditorGeneratorGUI.java`

The `SVGEditorGenerator.java` class defines constants for Generator classes (e.g., layout algorithms, names of ECMAScript files, or substitute strings used in SVG template components) and defines common attributes used by its subclasses (e.g., a list of entities and relationships managed by a specific Generator). It also provides default operations for Generators. For example:

- Rendering an entity given its entity type;

- Rendering a relation given its source, destination and relation type;
- Finding a layout entity from the list of entities given an entity name;
- Performing layout calculations; and
- Generating an SVG Editor by outputting all file components for SVG Editor.

The following code fragment shows the abstract methods in the `SVGEditorGenerator.java` class for which any specific Generator class must provide implementations:

```
public abstract class SVGEditorGenerator {
  // other fields and methods are omitted here …
  // analyze the input XML files; the implementation
  // must include calls to createNode(node),
  // createRelationship(src, dest, type) and
  // getNodeById() methods.
  public abstract void analyzeInputXMLFiles() ;
  // generates an SVG Editor; the implementation must
  // include calls to analyzeInputXMLFiles(),
  // performLayout(), and outputFiles() methods.
  public abstract void generateSVGEditor();
}
```

Figure 4-8: Abstract Methods Declared in `SVGEditorGenerator.java`

4.2.5 Creating a Specific SVG Editor

Figure 4-3 summarizes interdependencies of the template components and in our customization framework.

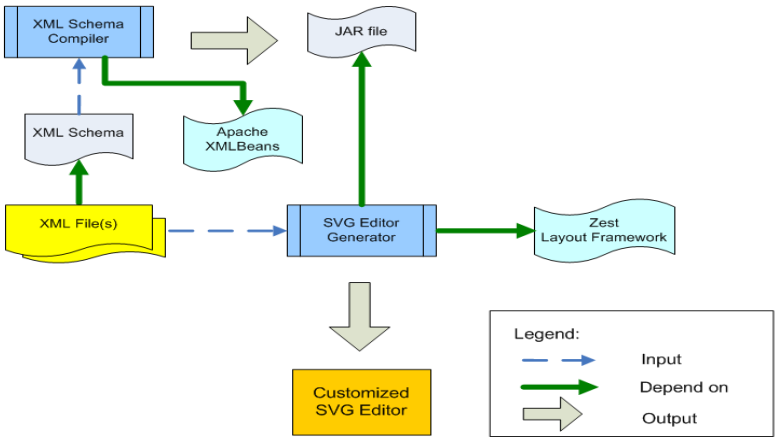

Figure 4-9: Template Solution Design for our Customization Framework

The steps used to generate a specific SVG Editor can be summarized as follows:

1. Describe the subject domain model and its data in XML (i.e., XML Schema and XML data files).
2. Use the XML Schema Compiler to compile an XML Schema into a JAR file. (i.e., XML Schema type definitions are mapped to Java interfaces and classes). Thus, the structure of the XML Schema can now be accessed from Java through the JAR file.
3. Create a new Java project and include the newly created JAR file in its build path;
4. Create a customized Generator by extending the abstract methods as declared in four abstract classes (i.e., `DomainModel.java`, `EntityNode.java`, `SVGEditorGenerator.java`, and `SVGEditorGeneratorGUI.java`).
5. Instantiate the generic calls to layout algorithms with calls to concrete layout algorithms from the Zest layout package;

6. Run the customized Generator GUI, which in turn requests preferences from the user for the generation process, and then invokes the SVG Editor Generator to produce the customized SVG Editor. Executing the SVG Editor Generator extracts the entity and relationship topology from the XML files, calculates a layout for this topology, and annotates the computed geometric location information with the topology information.
7. Finally, run the generated SVG Editor to explore and manipulate the entities and their relationships interactively.

4.3 Summary

This chapter described the high-level architecture and the detailed design of the various components for our customization framework. It also discussed the template solution to generate SVG Editor Generators for different information domains.

Chapter 5 SVG Editor Generator Reference Implementations

To demonstrate and illustrate how the template solution of our customization framework can readily be adapted to create SVG Editor Generators for different information domains, we present two reference SVG Editor Generator implementations—SVG Editor Generators for Java applications and organization charts. Furthermore, we provide a sample Eclipse project to exhibit the template solution and reference implementations.

5.1 SVG Editor Generator for Java Applications

Since there are many Java applications, an SVG Editor for Java projects is a very useful and practical visualization tool for exploring, manipulating and analyzing Java software structures. We use JavaBlueTooth as a subject system to illustrate the construction of an SVG Editor Generator for Java.

5.1.1 JComp and JavaBluetooth Project

JComp is a reuse-driven componentization framework for Java applications developed at the University of Waterloo [LT06]. The Java parser from the JComp toolkit extracts facts from an existing Java system. The output of the JComp Java parser is a set of XML files.

We use the JavaBlueTooth project as our subject Java system. Parsing the source code of JavaBlueTooth using JComp's parser reveals that this open-source Java application comprises 49 interfaces and classes and results into 49 XML files. The resulting XML Schema domain model includes the entity types of *interface* and *class* as well as the relationship types of *aggregation*, *association*, *composition*, *inheritance*, *realization* and *usage*.

5.1.2 Implementation

To create a specific SVG Editor for a Java application we follow the generation instructions discussed in Section 4.2.5:

1. Use the XML files resulting from parsing the Java Bluetooth project using the JComp parser as input for the specific SVG Editor Generator;
2. Compile the XML Schema domain model using the XML Schema Compiler to generate a JAR file `rclass.jar`;
3. Create a new Eclipse Java project; put the `rclass.jar` file in its `lib` folder and update the class build path to include the `rclass.jar` file;

4. Create a specific domain model class `JavaAppDomainModel.java`, an entity node class called `JavaEntityNode.java` class, a concrete generator class called `JavaSVGEditorGenerator.java`, and a GUI generator class called `JavaSVGEditorGeneratorGUI.java` to create a specific SVG Editor Generator; and finally
5. Run the `JavaSVGEditorGeneratorGUI.java` class; configure the template with preferences; and generate a customized SVG Editor for the Java Bluetooth application.

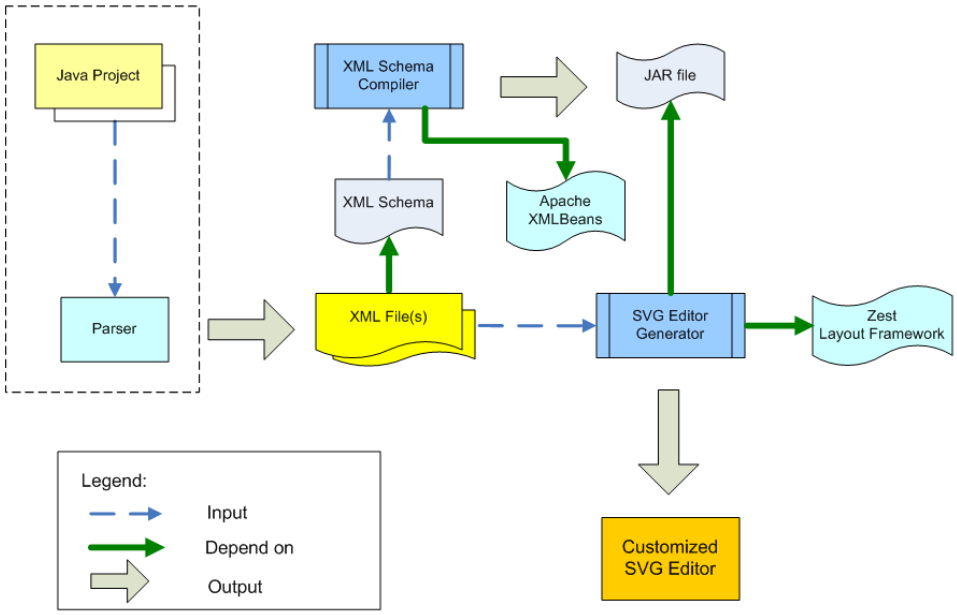

Figure 5-1: Creating a Customized SVG Editor for a Java Application

Figure 5-1 illustrates pictorially the different stages on how to create a customized SVG Editor for a Java application. Figure 5-2 shows the user interface for a specific SVG Editor Generator for Java applications. Using this interface, an End User can choose the

template parameter information. Default values are provided for the layout algorithm and the dimensions of the main window. Random colors are assigned to entity and relationship types. Once the user input passes validation, clicking the *Generate SVG Editor* button will generate the SVG Editor as specified by the End User.

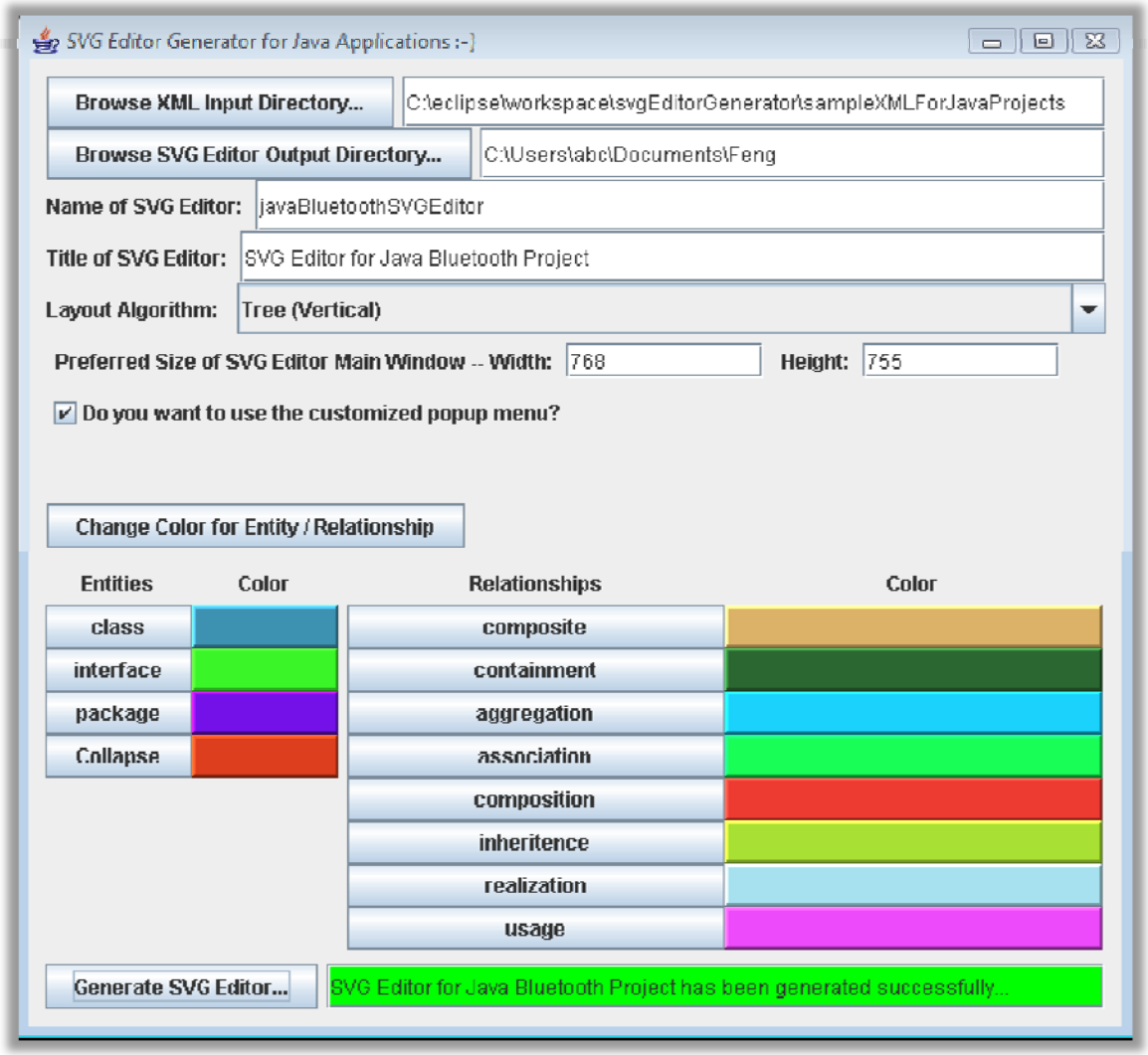

Figure 5-2: Specific SVG Editor Generator for Java Applications

5.1.3 Creating Esthetically Pleasing Layouts

The default strategy for computing the graph layout takes all relationships of the domain model into account. However, software structure visualizations often feature a specific relationship (e.g., call relationship for call graphs or dependency relationship for class diagrams). Thus, the initial layout exhibited when a specific SVG Editor is invoked might look like the graph depicted in Figure 5-3.

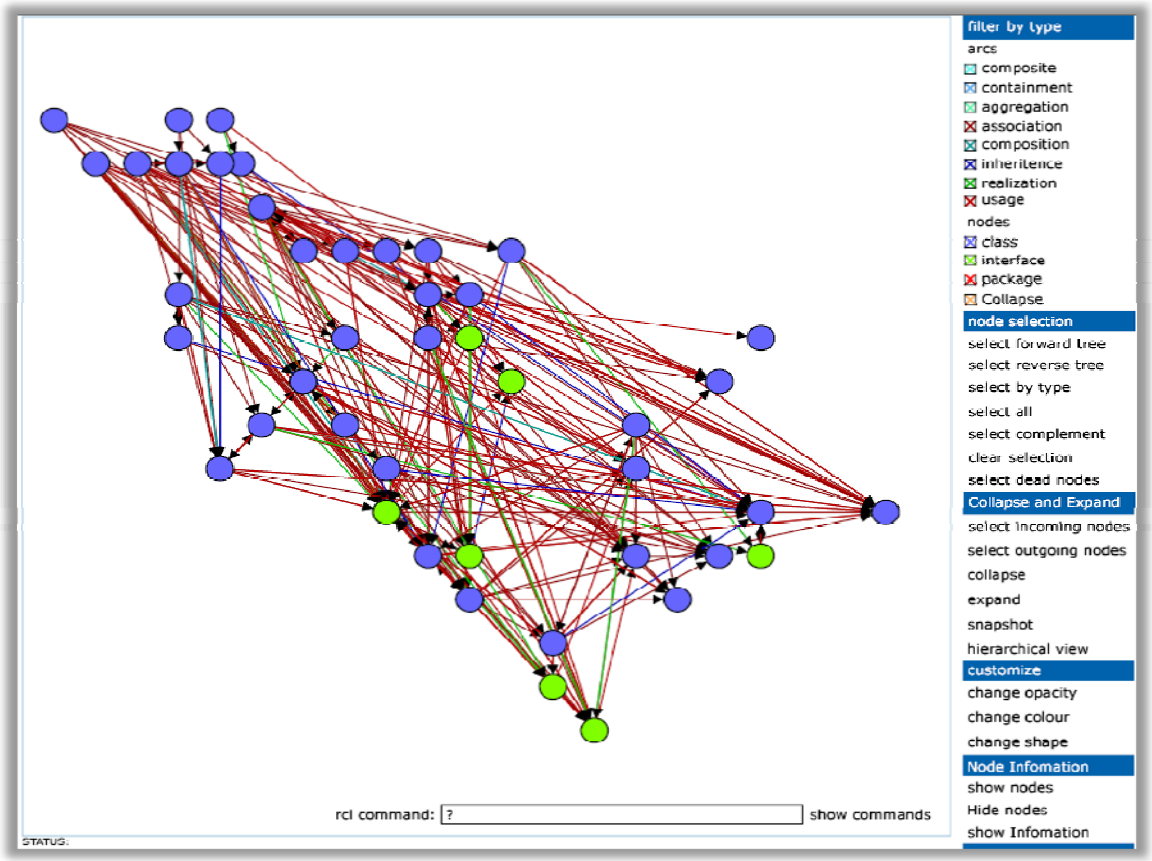

Figure 5-3: Layout using all Relationships—Vertical Tree Layout

Since all Java classes/interfaces have hierarchical relationships with their package, we add a new node type called *package* and new relationship called *containment* to solve the problem. Initially, only the instances of the *containment* relationship are added to the instance variable *relationships* in SVG Editor Generator. The layout based on the *containment* relationship only is esthetically pleasing. After the layout calculation, we add other types of relationships to the instance variable *relationships* of SVG Editor Generator. Figures 5-4 and 5-5 show the resulting vertical tree layout and radial layout, respectively.

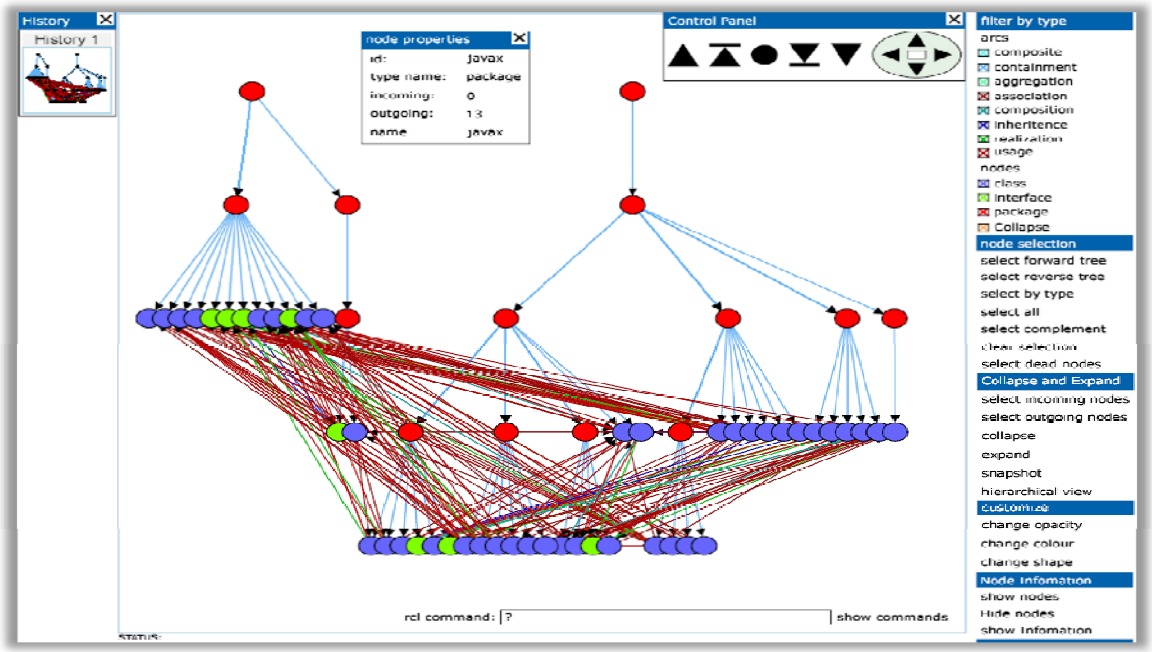

Figure 5-4: Vertical Tree Layout using *Containment* Relationship only

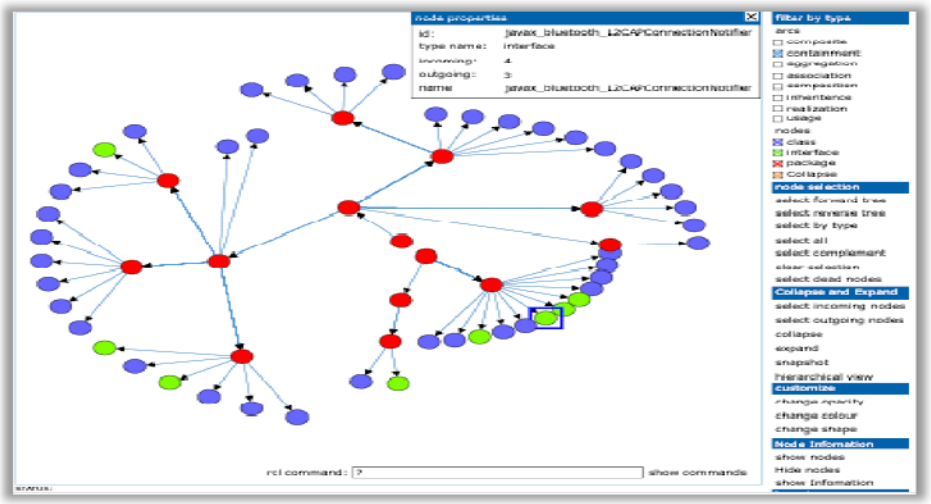

Figure 5-5: Radial Layout using Containment Relationship only

5.2 SVG Editor Generator for Organization Charts

Many graph editors, for example Microsoft Visio [Mic07], can generate organization charts based on organizational models. To demonstrate the flexibility of our customization framework with respect to input information domains, we apply it to the domain of organization charts.

Again, to create a special Generator for organization chart SVG Editors we follow the generation approach summarized in Section 4.2.5:

1. Define an XML Schema `organization.xsd` (see Appendix B) for the organization chart domain: the root element *organization* contains a number of *person* elements; each person element in turn has four sub-elements: *id*, *name*,

title and *supervisor*. The persons in an organization are linked through *supervise* relationships.

2. Create instance files which conform to the XML Schema file as the input:

```
<?xml version="1.0" encoding="UTF-8"?>
<organization xmlns="http://rigiuvic/organization">
  <person>
    <id>d1</id>
    <name>Sir. Topham Hat</name>
    <title>Director</title>
  </person>
  <person>
    <id>ad1</id>
    <name>Thomas</name>
    <title>AssistantDirector</title>
    <supervisor>d1</supervisor>
  </person>
</organization>
```

Figure 5-6: an Instance XML Data File for an Organization Chart

3. Compile the XML schema produced in Step 1 above using the XML Schema compiler GUI to generate a JAR file `organization.jar`;
4. Include the `organization.jar` file in the `lib` folder of the project created in Step 3 above and update the class build path to include the file;
5. Create a specific domain model class `OrgDomainModel.java`, an entity node class called `OrgChartEntityNode.java` class, a concrete generator class called `OrgChartSVGEditorGenerator.java` and GUI generator class

called `OrgChartSVGEditorGeneratorGUI.java` to create a specific SVG Editor Generator; and finally

6. Run the `OrgChartSVGEditorGeneratorGUI.java` class; configure the template with preferences; and generate a customized SVG Editor for an organization chart.

Figure 5-7 depicts the user interface of the special Generator for creating organization chart SVG Editors.

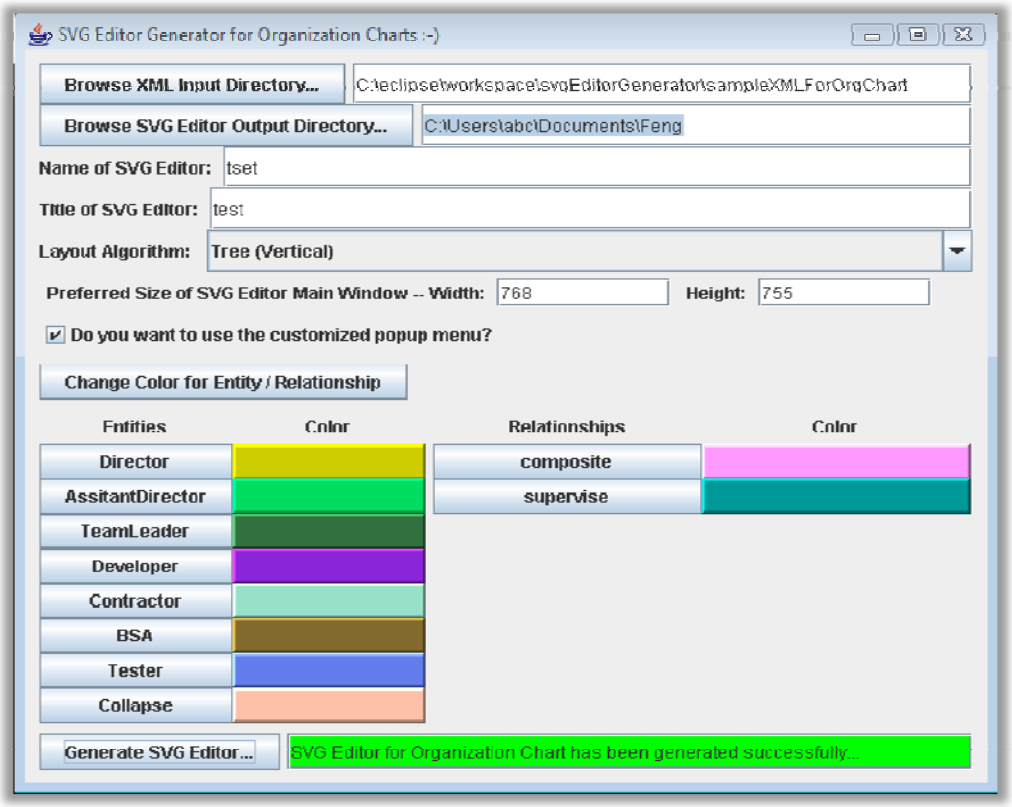

Figure 5-7: Specific SVG Editor Generator for Organization Charts

Figure 5-8 shows an SVG Editor generated by the specific Generator using the sample XML files mentioned above.

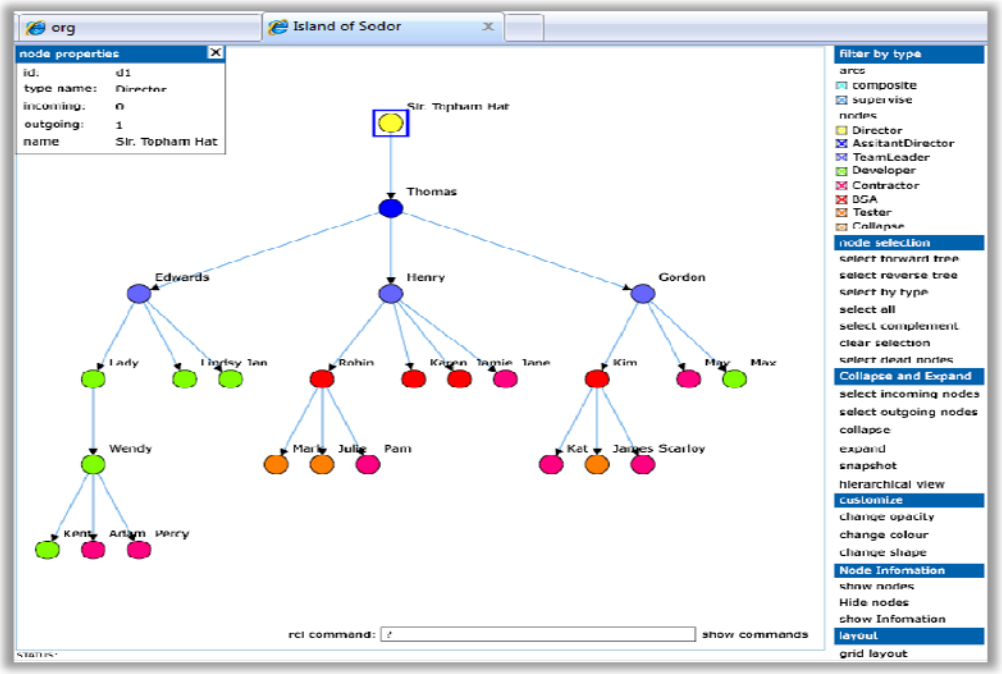

Figure 5-8: Generated Organization Chart SVG Editor

5.3 Template Solution Project

We provide an Eclipse Java project called `svgEditorGenerator` for the template solution of our customization framework. This project includes all components of the Generator, the XML Schema compiler class and associated GUI class, the `.classpath` and `.project` files, and a `lib` folder holding the Apache XMLBeans library files and

the Zest layout library. The `lib` folder should be updated to include the XMLBeans compiled JAR file for the domain described in XML Schema. This project also contains all the files for the reference implementations including sample XML files, XML Schema files, and subclasses that extend the aforementioned abstract classes. Figure 5-9 depicts the hierarchical dependencies among the components of the template solution of our customization framework in an Eclipse project.

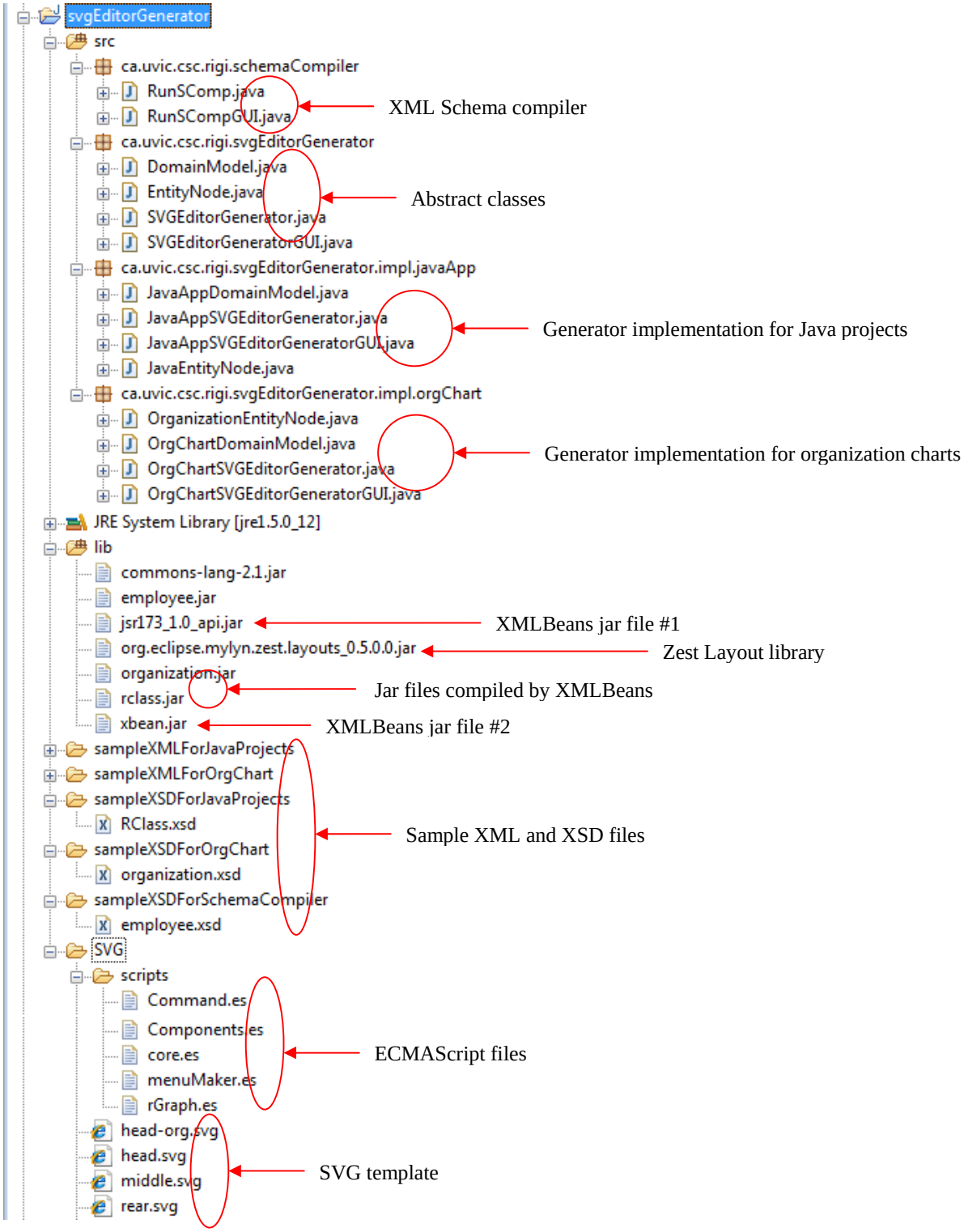

Figure 5-9: Components of Template Solution `svgEditorGenerator` Eclipse Project

5.4 Summary

This chapter presented two SVG Editor Generators reference implementations for Java applications and organization charts. We also composed an Eclipse project that includes all components of our SVG Editor Generator customization framework including necessary libraries and reference implementations.

Chapter 6 Evaluation

Chapters 4 and 5 described the design and implementation of our customization framework for the SVG visualization engine. This chapter evaluates the framework against its functional and non-functional requirements, introduces use cases for the framework, discusses its limitations, and finally documents lessons learned.

6.1 Evaluating Customization Framework Against Requirements

In this section, we assess our customization framework according to the functional and non-functional requirements as outlined in Chapter 3.

6.1.1 Evaluating Customization Framework Against FRs

In Section 3.2, we specified the functional requirements (FRs) of the customization framework and articulated that there should be components for each of its stakeholders. We now summarize how our customization framework meets the criteria:

- For **Designers**: we reverse engineered the existing SVG Editor and produced the high-level architecture diagram. In addition to the high-level understanding, we documented the existing system in sufficient detail to be able to design a customizable SVG Editor Generator suitable for various domains.
- For **Customizers**: we started from the architectural design and decided how to structure and componentize the customization framework for Customizers. We then used a template design approach to provide general steps and a skeleton project to allow Customizers to extend constructed SVG Editor Generators. Instructions on how to use this template project are summarized as follows:

    1) Copy the project `svgEditorGenerator` to the Eclipse workspace.

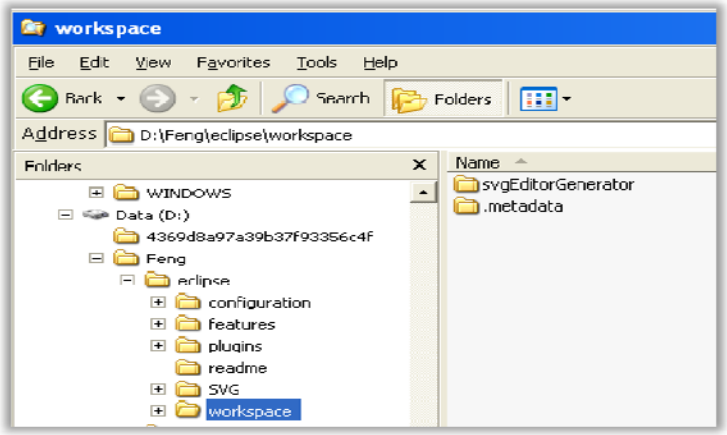

Figure 6-1: Copy `svgEditorGenerator` Project to Eclipse Workspace

2) Start Eclipse and import `svgEditorGenerator` project into the Eclipse workspace:

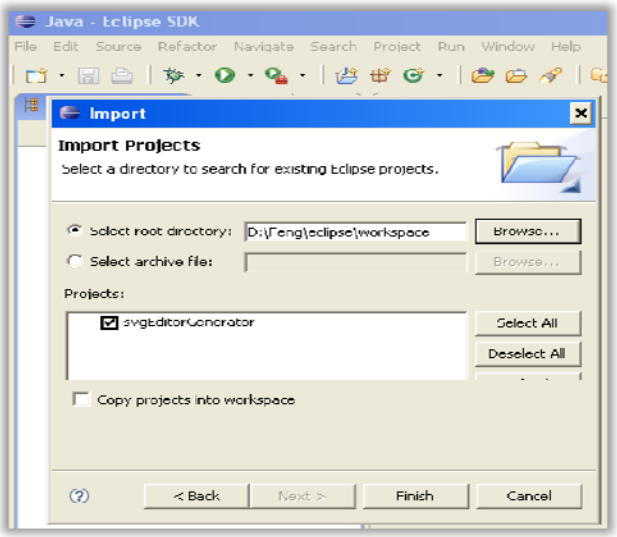

Figure 6-2: Import `svgEditorGenerator` Project into Eclipse Workspace

3) Define the information domain model in an XML Schema.

4) Create XML files to store the data to be visualized.

5) Run the XML Schema Compiler as illustrated in Figure 6-3. Click the *Browser XML Schema File* button to locate the XML Schema file to be compiled, and click *Browse Output Directory* button to specify the project `lib` folder as the output directory, as illustrated in Figure 6-4.

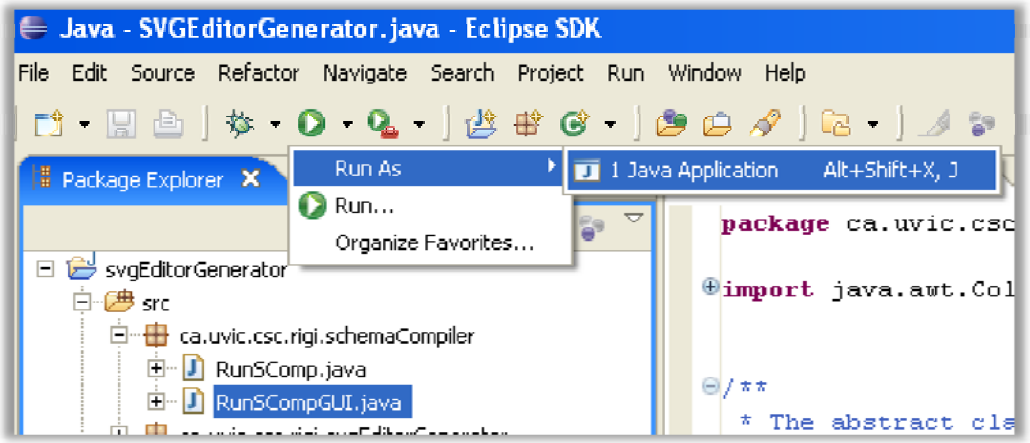

Figure 6-3: Run the XML Schema Compiler

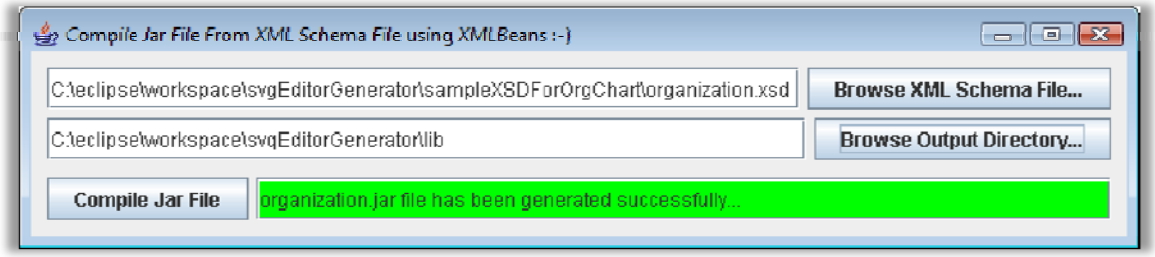

Figure 6-4: Compile an XML Schema into a JAR File

6) Update the project Java build path to include the newly created JAR file.

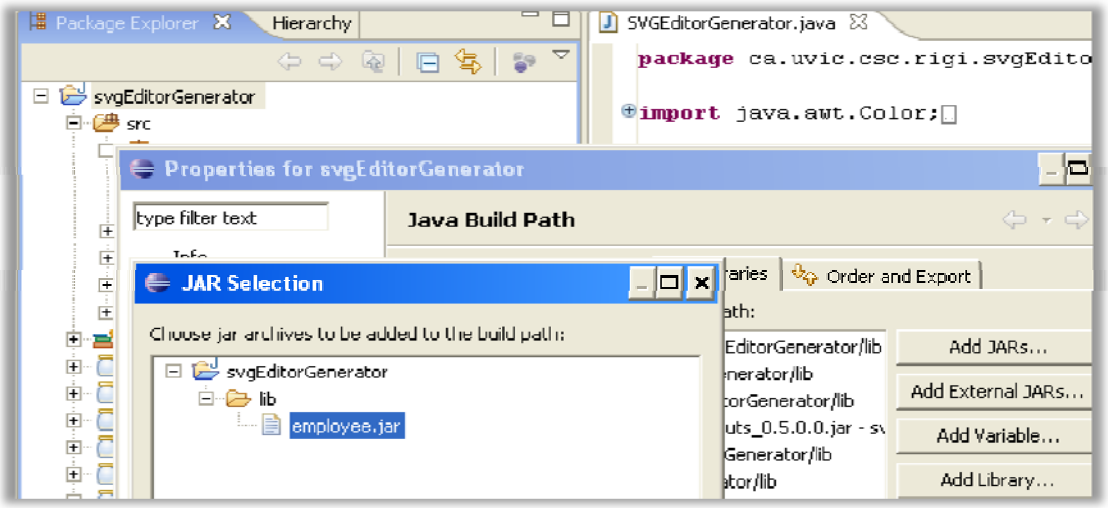

Figure 6-5: Update the Project Java Build Path

7) Follow the two reference implementations included in the packages called `ca.uvic.csc.rigi.svgEditorGenerator.impl.javaApp` & `ca.uvic.csc.rigi.svgEditorGenerator.impl.orgChart` to build a specific generator by providing concrete implementations for the four abstract classes defined in the package called `ca.uvic.csc.rigi.svgEditorGenerator`.
8) Run the specific generator GUI class to generate a customized SVG Editor.

- For **End Users**: The two reference implementations of the template solution project demonstrate that End Users can customize the look-and-feel of the resulting SVG Editor by specifying preferences.

Software customizations can be effected by customizing data, presentation, and control/behavior. Our customization framework meets these requirements as follows:

- **Data Customization**: Customizers use XML Schema (i.e., a platform independent meta-model) to describe the subject domain model. Entities and relationships described in the instance XML data that conform to the model are then transformed to nodes and arcs in the generated SVG Editor.
- **Presentation Customization**: We accomplished this as described in above mentioned customizations for End Users.
- **Control/Behavior Customization**: The template project allows End Users to use a default pop-up menu over the customized one provided by SVG Editor while generating the SVG Editor.

6.1.2 Evaluating Customization Framework Against NFRs

The following describes how the non-functional requirements (NFRs) are satisfied:

- **Reusable documents**: The document for Designers of an SVG Editor is a good reference for future Designers. Also, the document on how to use the template solution is useful to Customizers interested in changing SVG Editor as illustrated in Section 6.1.1.
- **Componentized and reusable template**: The template solution of our customization framework has been designed as a composite of third-party libraries and extensible components that can be reused to create a customized SVG Editor Generators.

- **Generic**: Customizers follow the generic steps to create particular SVG Editor Generators. Also, the template project is general enough so that Customizers can easily add new implementations to it. As long as a Customizer knows how to model a subject information domain as an XML Schema definition, our customization framework is able to produce customized SVG visualization engines for a variety of information models. However, the customization framework cannot be readily applied to information domains described in other formats, such as the common 3-tuple format (i.e., source, destination and relationship) used for example in RSF or GXL without translating the data to XML Schema and XML data.
- **Maintainable**: Both the componentized architecture and the template solution contribute to the maintainability of the customization framework. Also, we decomposed and encapsulated the domain of SVG Editor Generator into four abstract Java classes and included sufficient documentation in the source code of our customization framework to help Customizers retarget it to a new domain.
- **Easy to use end-user customization**: Through the generator GUIs, End Users can easily customize the resulting SVG Editor by selecting different preferences. However, the customization of the overall look-and-feel of the SVG Editor is limited.

6.2 Validating Template Solution Project

After demonstrating the customization framework to our research group, one of the group members immediately expressed interest in building an SVG Editor for his own research project which involves the visualization of goal models—models that contain entity types of *goal* and *soft goal* and relationship types of *and, or, make, help, hurt* and *break* [MCN92, DLF93]. Using the source code of the template solution and general instructions for creating a specific generator, he readily created an SVG Editor for visualizing goal models.

Following the instructions of Section 6.1.1, our customer imported the template project into his Eclipse workspace, defined the goal model in an XML schema, created instance XML files to store the information under study, compiled the XML Schema file into a JAR file, configured the Eclipse class path, and provided implementations for the four abstract classes defined in the `ca.uvic.csc.rigi.svgEditorGenerator` package. Figure 6-6 illustrates the updated `svgEditorGenerator` Java project, including the implementation for the special Generator for goal models. Implementations for the above mentioned four abstract classes are included in the package `ca.uvic.csc.rigi.svgEditorGenerator.impl.goalModel`. Figures 6-7 to 6-10 document the implementation of this SVG Editor Generator in detail.

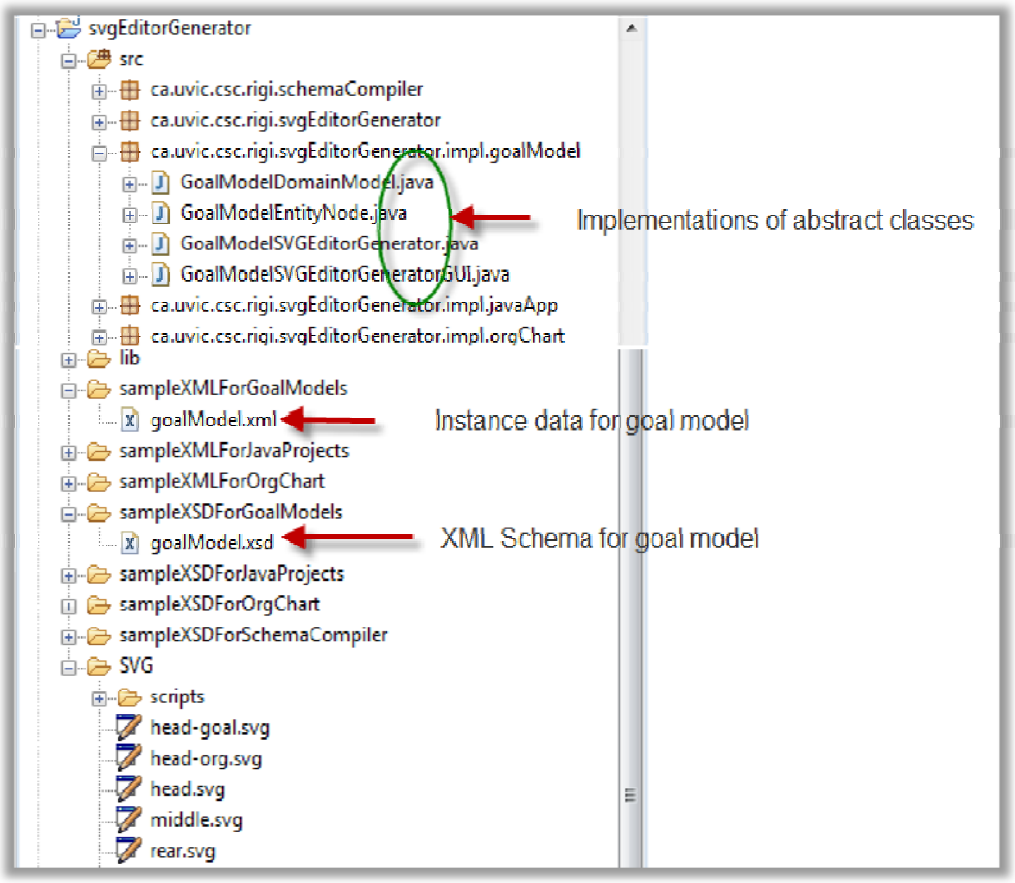

Figure 6-6: Updated `svgEditorGenerator` Project with Goal Models

GoalModelDomainModel.java

```
package ca.uvic.csc.rigi.svgEditorGenerator.impl.goalModel;

import ca.uvic.csc.rigi.svgEditorGenerator.DomainModel;

/**
 * This class defines entity and relationship types for goal model domain model.
 *
 * @author qin
 * @version 1.0, 2008-08-31
 */
public class GoalModelDomainModel extends DomainModel {

    public static final String AND = "and";
    public static final String OR = "or";
    public static final String MAKE = "make";
    public static final String HELP = "help";
    public static final String HURT = "hurt";
    public static final String BREAK = "break";

    private static final String[] ENTITY_TYPES = {"Goal", "SoftGoal", "Collapse"};

    private static final String[] RELATIONSHIP_TYPES = {"composite",
                                                        AND, OR, MAKE, HELP, HURT, BREAK};

    /**
     * @return the entity types handled in the domain.
     */
    public String[] getEntityTypes () {
        return ENTITY_TYPES;
    }

    /**
     * @return the relationship types handled in the domain.
     */
    public String[] getRelationshipTypes () {
        return RELATIONSHIP_TYPES;
    }
}
```

Figure 6-7: Implementation of `DomainModel.java`

GoalModelEntityNode.java

```
package ca.uvic.csc.rigi.svgEditorGenerator.impl.goalModel;

import ca.uvic.csc.rigi.svgEditorGenerator.EntityNode;

/**
 * The class represents an entity node for goal model. Node type can be user defined types
 * such as goal and softGoal.
 *
 * @author qin
 * @version 1.0, 2008-08-31
 */
public class GoalModelEntityNode extends EntityNode {

    /**
     * GoalModelEntityNode constructor.
     *
     * @param name the name
     * @param id the id
     * @param title the job title
     * @param imgURL the link to the image of the entity
     */
    public GoalModelEntityNode(String name, String id, String title, String imgURL) {
        this.name = name;
        this.id = id;
        this.type = title;
    }
}
```

Figure 6-8: Implementation of `EntityNode.java`

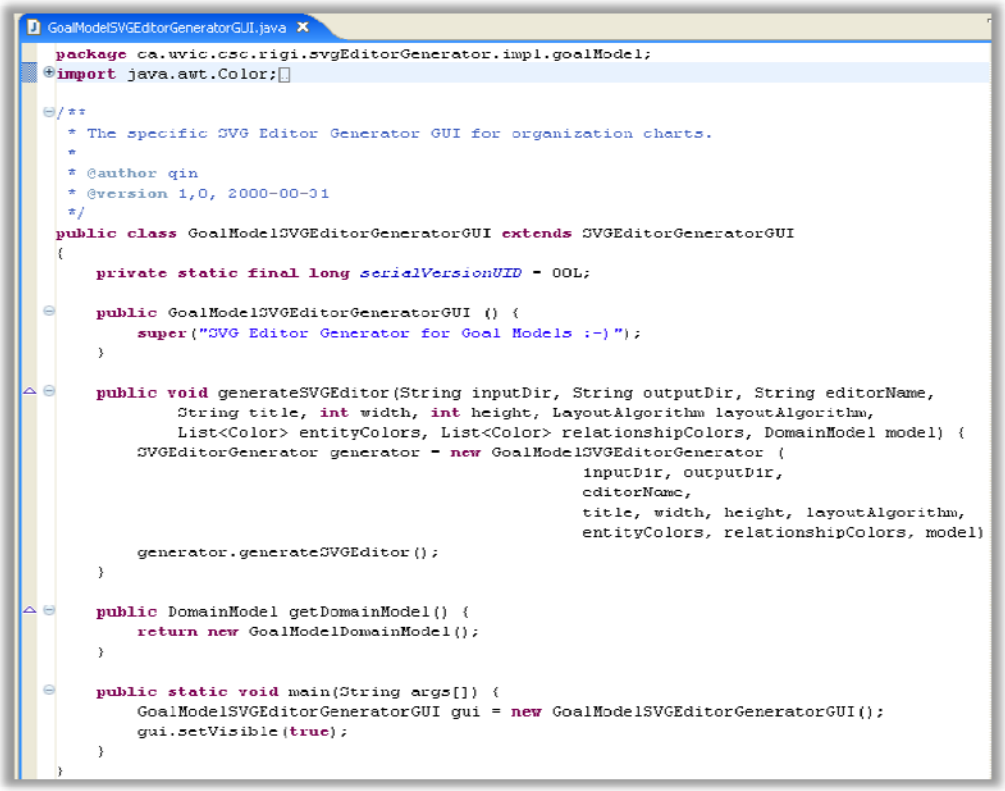

Figure 6-9: Implementation of `SVGEditorGeneratorGUI.java`

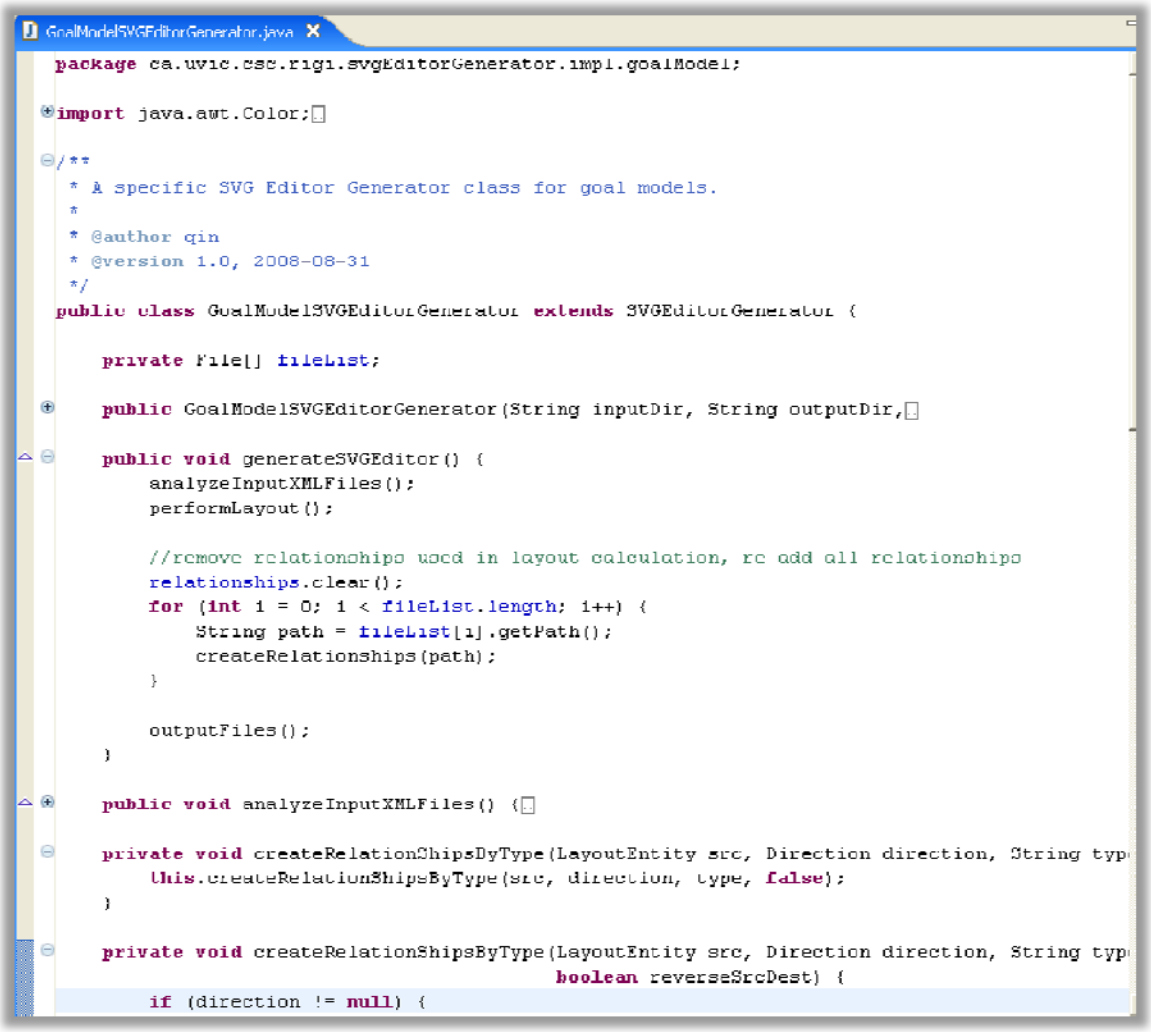

Figure 6-10: Implementation of `SVGEditorGenerator.java`

With little clarification, our customer produced his desired SVG Editor Generator using the widget depicted in Figure 6-11.

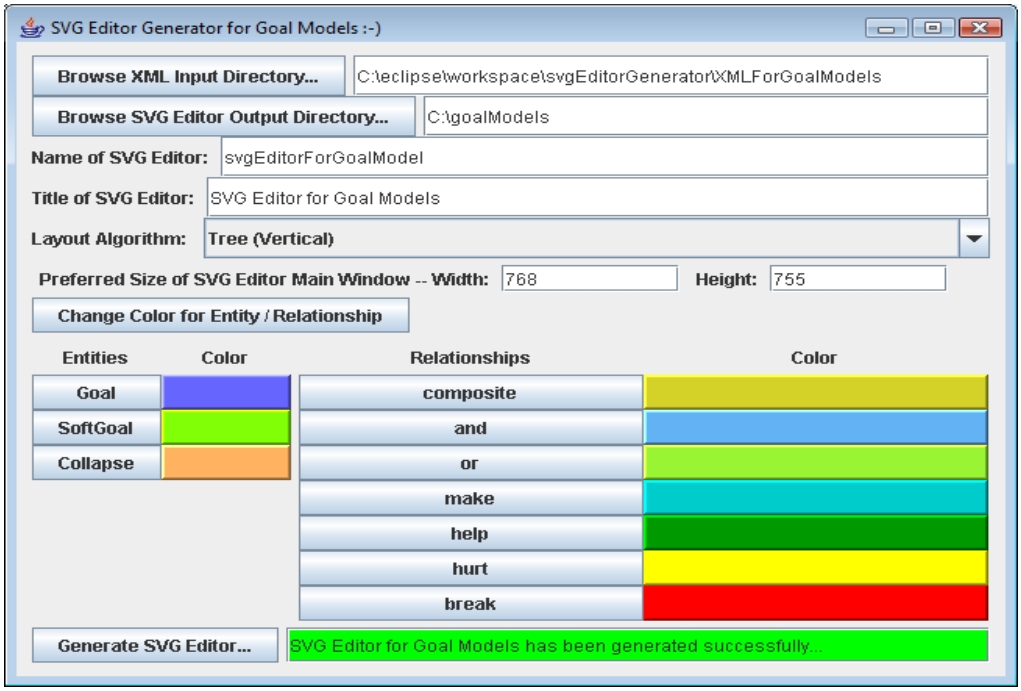

Figure 6-11: SVG Editor Generator for Goal Models

Figure 6-12 depicts the specific SVG Editor generated by the goal model SVG Editor Generator.

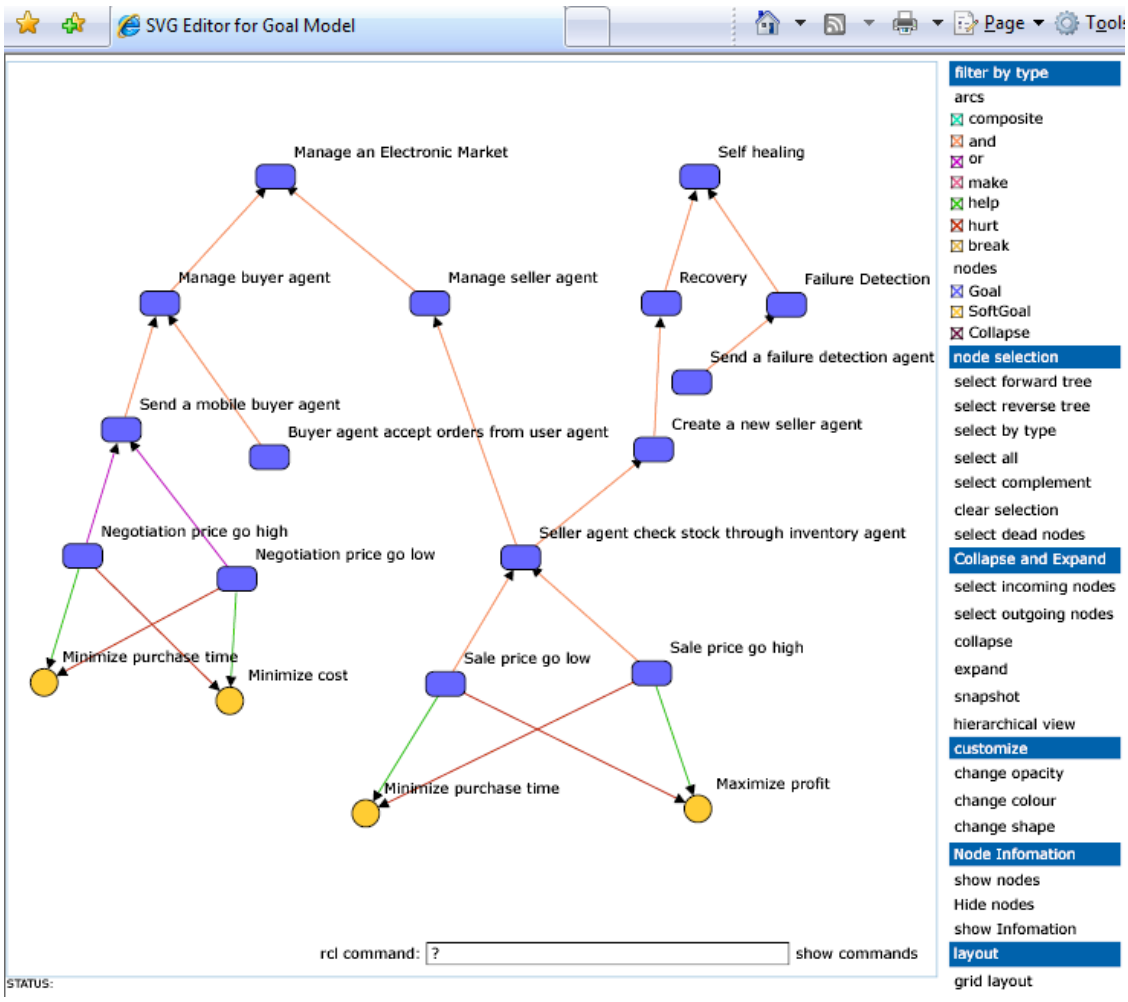

Figure 6-12: Specific SVG Editor for Goal Models

6.3 Experience and Lessons Learned

Although our customization framework has greatly simplified the process of creating domain-specific SVG Editor Generators, we experienced difficulties in defining an XML Schema for a new domain. One typical reason is insufficient knowledge of the target domain (e.g., organization charts) and, hence, the inability to model the subject domain effectively. Another important reason is that we did not use any modeling tools or XML editors to create XML Schemas (i.e., we used text editors).

Documenting while implementing a software system is certainly good practice. For example, without the information recorded in Appendix A, it is rather challenging to set up Apache XMLBeans on another computing system.

Moreover, it is good practice to provide reference implementations (e.g., as in our template solution project) so that potential users can refer to sample implementations readily. For example, in addition to the instruction manual on how to create a specific Generator, our initial customizer used the sample projects extensively, while he built the SVG Editor Generator for goal models.

6.4 Limitations

We have adopted Java and XML to achieve portable code and portable data. The techniques and tools that come with these languages are easy to understand for people who are familiar with them; however, using these languages effectively is surprisingly challenging for people who are unfamiliar with them and their associated concepts and

tools. Also, the data format is limited to XML at this point. For example, the customization framework is not yet able to parse data expressed in software artifact exchange formats (e.g., GXL or RSF).

6.5 Summary

This chapter evaluated our customization framework against the functional and non-functional requirements outlined in Chapter 3. We also validated the template solution project by successfully building a generator for goal models with one of our first customers. We then presented early experience with our customization framework and some lessons learned during the development of the framework. Finally, we discussed the limitations of our approach.

Chapter 7 Conclusions

SVG Editor was developed as a software visualization tool for users to explore and annotate software structures interactively. However, the adaptation of SVG Editor to different information domains requires significant effort—either by building an editor for a particular domain from scratch or by modifying an existing editor.

Given the need for a simplified and automated process to produce customized SVG Editors, we conducted background research on software customization and relevant related techniques. We then studied and analyzed the source code of the existing SVG Editor to obtain a high-level understanding and prepared detailed documentation of its design and implementation. First we specified the requirements for an SVG Editor Generator customization framework and then designed an architecture to construct specialized SVG visualization engines automatically. Probably, the most important

contribution of customization framework is the componentized template solution for a generic SVG Editor Generator and a general approach to creating a particular SVG Editor Generator. Following the general approach, we have realized the generator template solution by providing two reference implementations for assisting Customizers to use the SVG Editor Generators in different application domains.

7.1 Contributions

This section summarizes the main contributions of this thesis.

First, we reverse engineered the existing SVG Editor to produce abstract architecture diagrams and detailed design documentation to facilitate its understanding and design an architecture for generating customized SVG Editors.

Secondly, we created a customization framework for SVG Editor Generators that enables Customizers to generate SVG visualization engines suitable for various information visualization needs for a variety information domain models. In the process, we expanded the SVG visualization engine's application domain well beyond the software engineering realm.

Moreover, our customization framework greatly simplifies the process of constructing specific generators by leveraging an existing layout framework and providing a template solution which incorporates default behaviours.

Furthermore, we demonstrated the applicability of our customization framework by successfully applying it to build specific SVG Editor Generators which generate

customized SVG Editors for visualizing various software structures of Java applications, for viewing organization charts and exploring goal model diagrams.

Finally, in September 2008 we presented our customization framework in the Tool Demo Track at the *24th IEEE International Conference on Software Maintenance (ICSM)* in Beijing, China [LZKM08].

7.2 Future Work

While we successfully created a customization framework for our SVG visualization engine to generate SVG Editors for many application domains, there is plenty of room for improvement.

We could concentrate more effort on improving End User customizability of the generated SVG Editors. We may investigate additional End User customizable features (e.g., renaming nodes, adding nodes and relationships—desirable for a user of the organization charts SVG Editor). In addition to individual customization features, we might also consider providing a theme selector to change the overall look-and-feel of a particular SVG Editor.

Although we have produced an SVG Editor Generator for Java applications, the Generator is greatly dependent on the data obtained from the Java parser in the JComp toolkit. To make the Java structure SVG Editor Generator more versatile, the customization framework could be extended with an additional level of indirection to accommodate a variety of Java parsers and potentially parsers for other languages. For example, Rigi accomplished this by introducing a standard interchange format (i.e., RSF).

Since the input data for SVG Editor Generator is supposed to be in XML format, it might be prudent to integrate our customization framework with XML modeling framework. For example, EMF is a good candidate for a modeling language for our framework. We may accomplish this by creating an SVG Editor Generator Eclipse plug-in to integrate our customization framework seamlessly with the Eclipse environment and EMF; the generated SVG Editor would be displayed inside Eclipse, just as we did with the SVG Editor Eclipse plug-in for EMF [Lin08].

Furthermore, we might disseminate our customization framework further by applying it to build specific generators for popular information models (e.g., GXL models) and conduct user studies to evaluate the effectiveness of our customization framework for different user groups.

Bibliography

[Ado] Adobe: SVG Zone. http://www.adobe.com/svg/

[aiS] AbsInt Angewandte Informatik GmbH: aiSee Graph Layout Software. http://www.aisee.com/index.html

[Alt] Altova: XMLSpy—XML Editor for Modeling, Editing, Transforming, & Debugging XML Technologies. http://www.altova.com/products/xmlspy/xml_editor.html

[Apa05] The Apache Software Foundation: Apache Xerces. http://xerces.apache.org/, 2005.

[Apa08a] The Apache Software Foundation: Apache Commons. http://commons.apache.org/, August 2008.

[Apa08b] The Apache Software Foundation: Apache XMLBeans. http://xmlbeans.apache.org/, July 2008.

[BBS04] R. Ian Bull, Casey Best and Margaret-Anne Storey: Advanced Widgets for Eclipse. In *Proceedings 2nd Eclipse Technology eXchange (ETX 2004)*, pages 6–11, Vancouver, BC, Canada, October 2004.

[BF05] R. Ian Bull and Jean-Marie Favre: Visualization in the Context of Model Driven Engineering. In *Proceedings Workshop on Model Driven Development of Advanced User Interfaces (MoDELS 2005)*, Montego Bay, Jamaica, October 2005.

[BSM02] Casey Best, Margaret-Anne Storey and Jeffery W. Michaud: Designing a Component-Based Framework for Visualization in Software Engineering and Knowledge Engineering. In *Proceedings 14th International Conference on Software Engineering and Knowledge Engineering (SEKE 2004)*, pages 323–326, Ischia, Italy, July 2002.

[BSM+03] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick and T. Grose: *Eclipse Modeling Framework.* Addison-Wesley, August 2003.

[Bul08] R. Ian Bull: *Model Driven Visualization: Towards a Model Driven Engineering Approach for Information Visualization.* Ph.D. Dissertation, Department of Computer Science, University of Victoria, 2008.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker: *Generative Programming, Methods, Tools and Applications*. Addison-Wesley, 2000.

[Cer02] Ethan Cerami: *Web Services Essentials.* O'Reilly, 2002.

[CGN04] Code Generator Network: Interview with Krzysztof Czarnecki. http://www.codegeneration.net/tiki-read_article.php?articleId=64, Aug 2004.

[CHI] CHISEL: Zest: The Eclipse Visualization Toolkit. http://www.thechiselgroup.org/zest

[COM+07] Maria Cutumisu, Curtis Onuczko, Matthew McNaughton, Thomas Roy, Jonathan Schaeffer, Allan Schumacher, Jeff Siegel, Duane Szafron, Kevin Waugh, Mike Carbonaro, Harvey Duff and Stephanie Gillis: ScriptEase: A Generative/Adaptive Programming Paradigm for Game Scripting. *Science of Computer Programming*, Volume 67, Issue 1, pages 32–58, June 2007.

[Die05] Stephan Diehl: Software Visualization: In *Proceedings 27th ACM/IEEE International Conference on Software Engineering (ICSE 2005)*, pages 718–719, St. Louis, Missouri, USA, May 2005.

[DLF93] Anne Dardenne, Axel Van Lamsweerde and Stephen Fickas: Goal Directed Requirements Acquisition. *Science of Computer Programming*, Volume 20, Issues 1–2, pages 3–50, April 1993.

[DV02] Péter Domokos and Dániel Varró: An Open Visualization Framework for Metamodel-Based Modeling Languages. In *Proceedings of International Workshop on Graph-Based Tools (GraBaTs 2002)*, pages 78–87, Rome, Italy, October 2002.

[Ecl] Eclipse: www.eclipse.org

[EMF] Eclipse: Eclipse Modeling Framework: http://www.eclipse.org/modeling/emf/

[Ern04] Neil A. Ernst: *Towards Cognitive Support in Knowledge Engineering: An Adoption-Centred Customization Framework for Visual Interfaces*: M.Sc. Thesis, Department of Computer Science, University of Victoria, 2004.

[FDFH97] James D. Foley, Andries van Dam, Steven K. Feiner and John F. Hughes: *Computer Graphics: Principles and Practice.* Addison-Wesley, 1997.

[Fin94] Piotr Findeisen: *The Metaview System*: Technical Report TR96-13, Department of Computing Science, University of Alberta, 1994.

[FS97] Mohamed Fayad and Douglas C. Schmidt: Object-Oriented Application Frameworks. *Communications of the ACM,* Volume 40, Issue 10, pages 32–38, October 1997.

[GEF] Graphical Editor Framework (GEF) Project: http://www.eclipse.org/gef

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson and John M. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GKM05a] Grace Gui, Holger M. Kienle and Hausi A. Müller: REGoLive: Web Site Comprehension with Viewpoints. In *Proceedings 13th IEEE International Workshop on Program Comprehension (IWPC 2005),* pages 161–164, St. Louis, Missouri, USA, May 2005.

[GKM05b] Grace Gui, Holger M. Kienle and Hausi A. Müller: REGoLive: Building a Web Site Comprehension Tool by Extending GoLive: In *Proceedings 7th IEEE International Symposium on Web Site Evaluation (WSE 2005),* pages 46–53, Budapest, Hungary, September 2005.

[Gui05] Grace Qing Gui: *Extending a Web Authoring Tool for Web Site Reverse Engineering*. M.Sc. Thesis, Department of Computer Science, University of Victoria, 2005.

[GXL] GXL—Graph eXchange Language: http://www.gupro.de/GXL/

[HDFW02] Ted N. Husted, Cedric Dumoulin, George Franciscus and David Winterfeldt: *Struts in Action: Building Web Applications with the Leading Java Framework*. Manning Publications, 2002.

[HLM03] Bowen Hui, Sotirios Liaskos and John Mylopoulos: Requirements Analysis for Customizable Software—A Goals-Skills-Preferences Framework. In *Proceedings 11th IEEE International Requirements Engineering Conference (RE 2003),* pages 117–126, Monterey Bay, California, USA, September 2003.

[ISS02] International Summer School on Generative Programming: http://www.cs.tut.fi/~gp/index.html

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin and Ivan Kurtev: ATL—A Model Transformation Tool. *Science of Computer Programming,* Volume 72, Issues 1-2, 1, pages 31–39, June 2008.

[JET] JET: Java Emitting Template Tutorial. http://www.eclipse.org/articles/Article-JET2/jet_tutorial2.html

[Joh97] Ralph E. Johnson: Components, Frameworks, Patterns. In *Proceedings Symposium on Software Reusability (SSR '97)*, pages 10–17, Boston, MA, USA, May 1997.

[JUN] JUNG: Java Universal Network/Graph Framework. http://jung.sourceforge.net

[KWM02] Holger M. Kienle, Anke Weber and Hausi A. Müller: Leveraging SVG in the Rigi Reverse Engineering Tool. In *Proceedings First Annual Conference on Scalable Vector Graphics (SVG Open 2002) and Carto.net Developers Conference*, Zürich, Switzerland, July 2002.

[KNS08] Gabor Karsai, Sandeep Neema and David Sharp: Model-Driven Architecture for Embedded Software: A Synopsis and an Example. *Science of Computer Programming, June 2008.*

[Kos03] Rainer Koschke: Software Visualization in Software Maintenance, Reverse Engineering, and Re-engineering—A Research Survey. *Journal of Software Maintenance and Evolution: Research and Practice,* Volume 15, pages 87–109, 2003.

[KR06] Vinay Kulkarni and Sreedhar Reddy: Introducing MDA in a Large IT Consultancy Organization. In *Proceedings 13th Asia Pacific Software Engineering Conference (APSEC 2006)*, pages 419–426, Dec. 2006.

[KWB03] Anneke Kleppe, Jos Warmer and Wim Bast: *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[Lin08] YingYun Lin: *A Customizable SVG Graph Visualization Engine*. M.Sc. Thesis, Department of Computer Science, University of Victoria, 2008.

[LT06] Shimin Li and Ladan Tahvildari: JComp: A Reuse-Driven Componentization Framework for Java Applications. In *Proceedings 14th IEEE International Conference on Program Comprehension (ICPC 2006),* pages 264–267, Athens, Greece, June 2006.

[LZKM07] Tony Lin, Feng Zou, Holger M. Kienle and Hausi A. Müller: A Customizable SVG Graph Visualization Engine. In *Proceedings 5th Annual Conference on Scalable Vector Graphics (SVG Open 2007),* Tokyo, Japan, September 2007.

[LZKM08] Tony Lin, Feng Zou, Holger M. Kienle and Hausi A. Müller: A Domain-Customizable SVG-Based Graph Editor for Software Visualizations. In *Proceedings 24th IEEE International Conference on Software Maintenance (ICSM 2008)*, Beijing, China, September 2008.

[Mac91] Wendy E. Mackay: Triggers and Barriers to Customizing Software. In *Proceedings SIGCHI Conference on Human Factors in Computing Systems (CHI '91)*, pages 153–160, New Orleans, Louisiana, USA, 1991.

[MBK07] Amen R. Mashariki, Lee R. Bronner and Peter Kazanzides: Designing and Developing Medical Device Software Systems Using the Model Driven Architecture (MDA). In *Proceedings 2007 Joint Workshop on High Confidence Medical Devices, Software, and Systems and Medical Device Plug-and-Play Interoperability*, pages156–159, Boston, Massachusetts, USA, June 2007.

[MCN92] John Mylopoulos, Lawrence Chung and Brian Nixon: Representing and Using Nonfunctional Requirements: A Process-Oriented Approach. *IEEE Transactions on Software Engineering*, Volume 18, Issue 6, pages 483–497, Jun 1992.

[ME06] Brett McLaughlin and Justin Edelson: *Java and XML*. O'Reilly, 2006.

[Mic03] Jeffery William Michaud: *A Software Customization Framework.* M.Sc. Thesis, Department of Computer Science, University of Victoria, 2003.

[Mic07] Microsoft Corporation: Microsoft Office Visio 2007. http://office.microsoft.com/en-ca/visio/default.aspx

[MK88] Hausi A. Müller and Karl Klashinsky: Rigi—A System for Programming-in-the-large. In *Proceedings 10th ACM/IEEE International Conference on Software Engineering (ICSE '88)*, pages 80–86, April 1988.

[MKK+03] Jun Ma, Holger M. Kienle, Piotr Kaminski, Anke Weber and Marin Litoiu: Customizing Lotus Notes to Build Software Engineering Tools. In *Proceedings IBM/ACM CAS Conference 2003 (CASCON 2003)*, pages 276–287, Toronto, ON, Canada, October 2003.

[MOT93] 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*, Volume 5, Issue 4, pages 181–204, December 1993.

[Mül] Hausi A. Müller: RIGI: A Visual Tool for Understanding Legacy Systems. http://www.RIGI.csc.uvic.ca

[MWK03] Hausi A. Müller, Anke Weber and Holger M. Kienle: Leveraging Cognitive Support in SVG Applications from the Host COTS Product. In *Proceedings Second Annual Conference on Scalable Vector Graphics (SVG Open 2003)*, Vancouver, BC, Canada, July 2003.

[OMG] OMG Model Driven Architecture: http://www.omg.org/mda/

[OKS92] Oksnøen Symposium, May 1992. http://www.ul.ie/~idc/library/papersreports/LiamBannon/27/OKS1.html

[Ora] Oracle Corporation: Oracle XML Parser for Java V2. http://www.oracle.com/technology/tech/xml/xdk/doc/production/java/doc/java/parser/readme.html

[Pri] Jonathan Price: The Audience of One: Making Professional Communication Personal. In *Proceedings 2001 Professional Communication Conference*, pages 199–204, Santa Fe, New Mexico, USA, 2001.

[Pro] Protégé: http://protege.stanford.edu

[Rus04] Robert Russell: SVGBasics Tutorials. http://www.svgbasics.com/, 2004.

[SB08] Anne Sunikka and Johanna Bragge. What, Who and Where: Insights into Personalization. In *Proceedings 41st Annual Hawaii International Conference on System Sciences*, pages 283–283, Hawaii, USA, January 2008.

[SCG07] SCG Wiki: A Non-exhaustive List of Software Visualization Tools, 2007. http://smallwiki.unibe.ch/codecrawler/anon-exhaustivelistofsoftwarevisualizationtools/

[Sch01] Roger Schrag: XML and Oracle. http://www.dbspecialists.com/files/presentations/xml_and_oracle.html, 2001.

[SDBP98] John T. Stasko, John B. Domingue, Marc H. Brown and Blaine A. Price: *Software Visualization*. MIT Press, 1998.

[Sko03] Aaron Skonnard: Understanding XML Schema, 2003. http://msdn.microsoft.com/en-us/library/aa468557.aspx.

[STM88] Paul G. Sorenson, Jean-Paul Tremblay and Andrew J. McAllister: The Metaview System for Many Specification Environments. *IEEE Computer*, Volume 5, No. 2, pages 30–38, March 1988.

[Sto] Margaret-A. Storey: SHriMP Views (Simple Hierarchical Multi-Perspective). http://www.thechiselgroup.org/shrimp

[Sun08a] Sun Microsystems: Java Technology. http://www.sun.com/java/

[Sun08b] Sun Microsystems: Trail: Creating a GUI with JFC/Swing. http://java.sun.com/docs/books/tutorial/uiswing/

[TMWW93] Scott R. Tilley, Hausi A. Müller, Michaels J. Whitney and Kenny Wong: Domain-Retargetable Reverse Engineering. In *Proceedings IEEE International Conference on Software Maintenance (ICSM '93)*, pages 142–151, Montréal, Québec, Canada, September 1993.

[Til94] Scott R. Tilley: Domain-Retargetable Reverse Engineering II: Personalized User Interfaces. In *Proceedings IEEE International Conference on Software Maintenance (ICSM '94)*, pages 336–342, Seoul, Korea, September 1994.

[TOS] Tom Sawyer Software: Tom Sawyer Layout. http://www.tomsawyer.com/products/index.php

[TS08] Artem Tikhomirov and Alexander Shatalin: Introduction to the Graphical Modeling Framework. *Tutorial at EclipseCon 2008*, Santa Clara, California, USA, March 2008.

[TWS94] Scott R. Tilley, Kenny Wong, Margaret-Anne D. Storey and Hausi A. Müller: Programmable reverse engineering. *International Journal of Software Engineering and Knowledge Engineering*, pages 501-520, December 1994.

[VKEH06] Markus Voelter, Bernd Kolb, Sven Efftinge and Arno Haase: *From Front End To Code—MDSD in Practice*, June 2006. http://www.eclipse.org/articles/Article-FromFrontendToCode-MDSDInPractice/article.html

[W3C99] W3C: XSL Transformations (XSLT) 1.0, 1999: http://www.w3.org/TR/xslt.html

[W3C03] W3C: Scalable Vector Graphics (SVG) 1.1 Specification, 2003. http://www.w3.org/TR/SVG/

[W3C06] W3C: Extensible Markup Language (XML) 1.0 (Fourth Edition), 2006. http://www.w3.org/TR/REC-xml/

[W3Sa] W3Schools: DTD Tutorial. http://www.w3schools.com/dtd/

[W3Sb] W3Schools: XML Schema Tutorial. http://www.w3schools.com/Schema/default.asp

[YTH06] Stephen S. Yau, Choksing Taweponsomkiat and Dazhi Huang: A Framework for Extensible Component Customization for Component-based Software Development. In *Proceedings Sixth International Conference on Quality Software (QSIC 2006)*, pages 369–376, Beijing, China, Oct. 2006.

[yWo] yWorks: The Diagramming Company: *yFiles*: http://www.yworks.com/en/products_yfiles_about.htm

[Zha03] Kang Zhang: *Software Visualization: From Theory to Practice*. Springer, 2003.

Appendix A Setup Environment for XMLBeans

To setup XMLBeans under Windows XP, XMLBEANS_HOME has to be defined as an environment variable, and the bin folder of XMLBeans distribution has to be included in the path of the user variable, as illustrated in Figures A-1 and A-2:

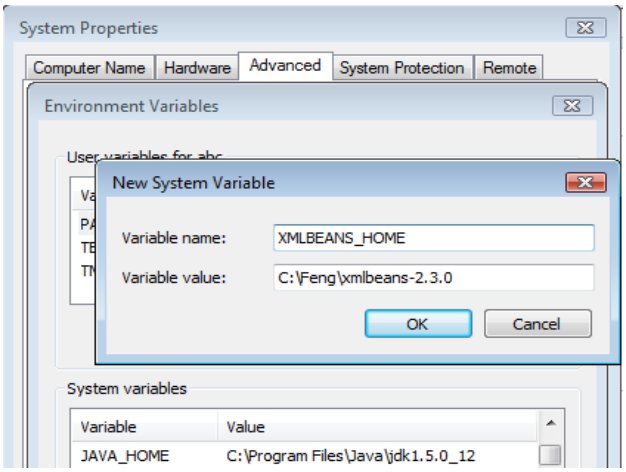

Figure A-1: Define New System Variable XMLBEANS_HOME

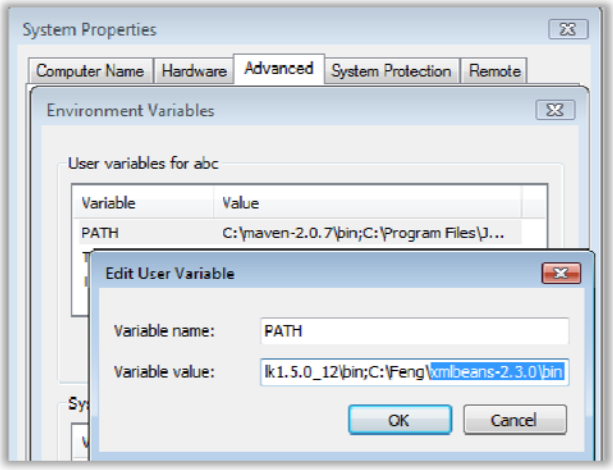

Figure A-2: Add XMLBeans `bin` Folder to `PATH` User Variable

We follow the example from XMLBeans tutorial, run the following in sample command and get the error as shown in Figure A-3.

```
scomp -out employee.jar employee.xsd
```

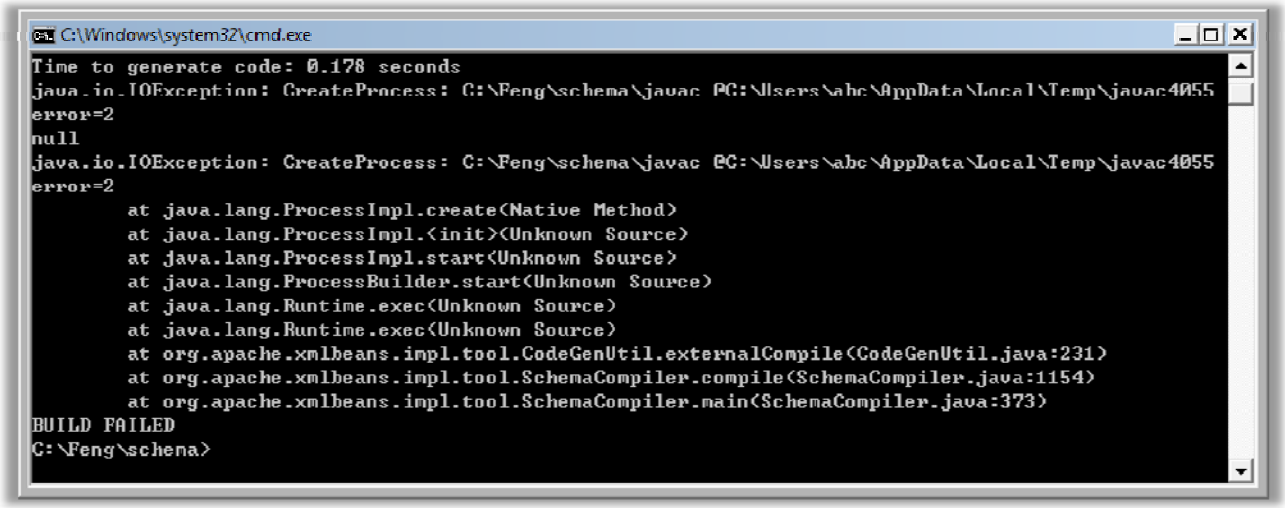

Figure A-3: IOException Occurs during XMLBeans JAR Compilation

When looking at scomp.cmd file in the XMLBeans bin folder, we recognize that the following main class is used for generating the JAR file: `org.apache.xmlbeans.impl.tool.SchemaCompiler.java`.

After getting the XMLBeans source files, setting up an XMLBeans project and debug breakpoints at the suspicious location (`CodeGenUtil.java` line 231), as depicted in Figures A-4 and A-5, we successfully re-produce the exception as illustrated in Figure A-6:

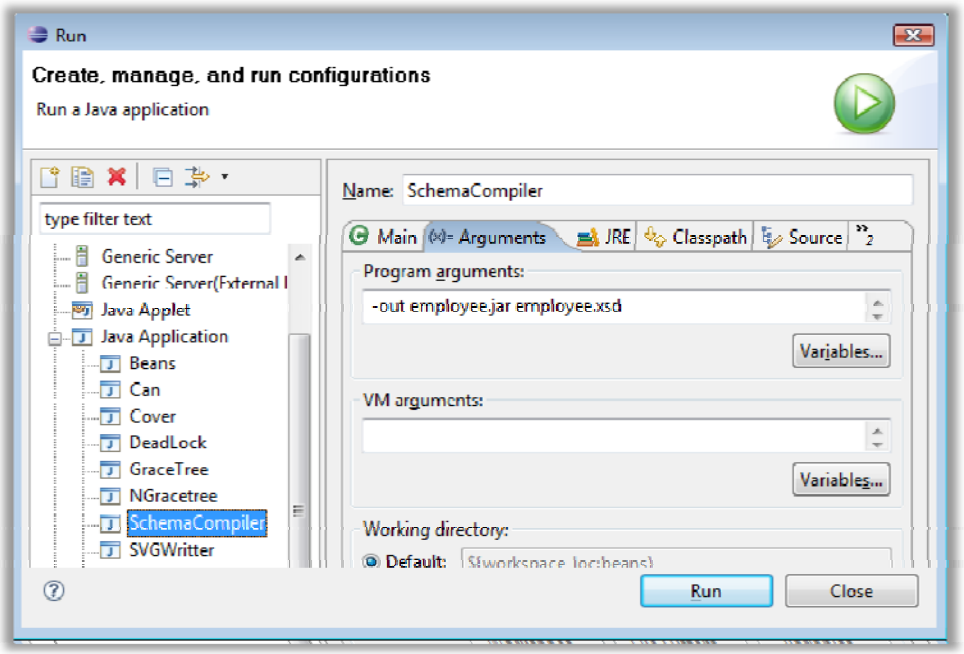

Figure A-4: Setup Arguments for SchemaCompiler Class in Eclipse

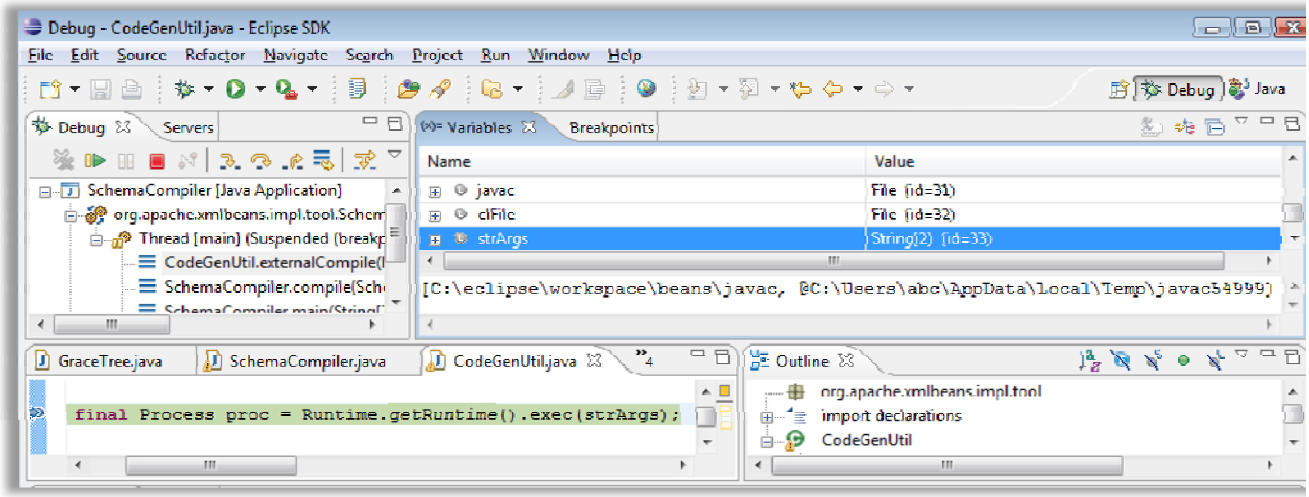

Figure A-5: Setup Debug Breakpoint in Eclipse

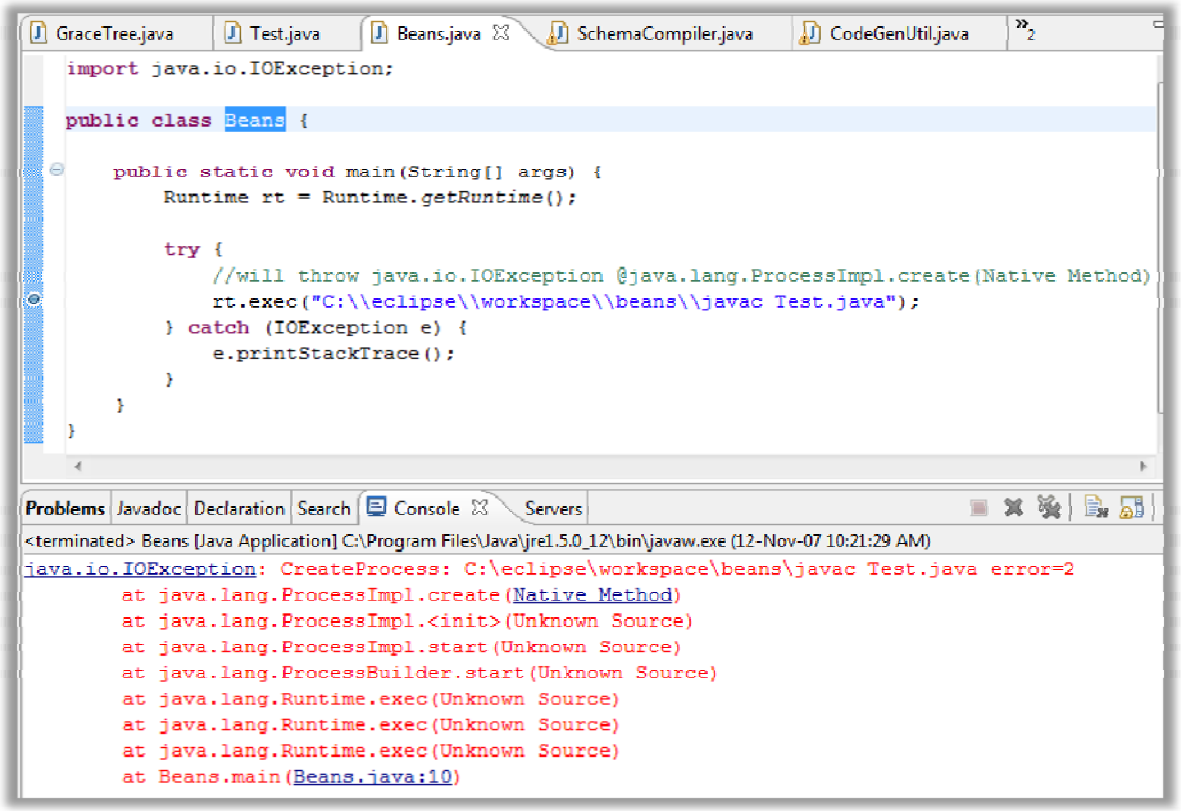

Figure A-6: Reproduce IOException in a Java Test Class

The next step is to find out how to solve the problem. After several tries, the following statement is run successfully:

```
rt.exec (“javac Test.java”);
```

Thus, it is the value of the first element in the `strArgs` variable as shown in Figure A-7 which causes this problem. As illustrated in Figure 4-8, this variable must be set to the value of the absolute path of a file instead of the relative value.

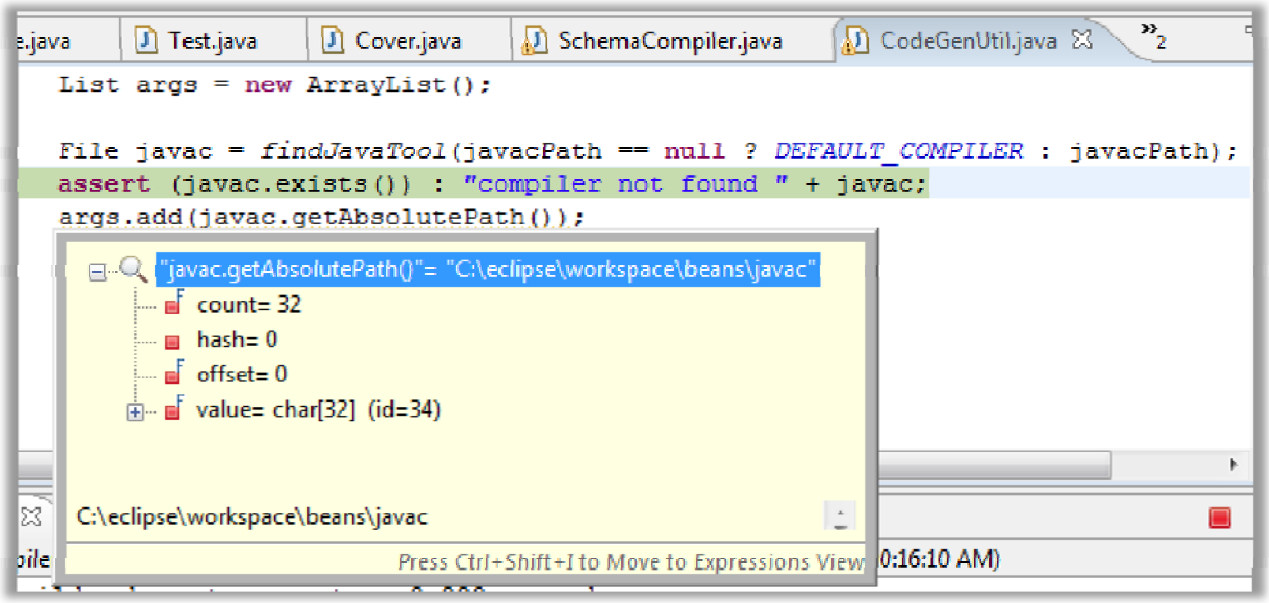

Figure A-7: Root Cause of the Problem

After fixing this bug using the relative value of the <code>*File javac*</code>, XMLBeans generates the expected Java library `employee.jar` as depicted in Figure A-8:

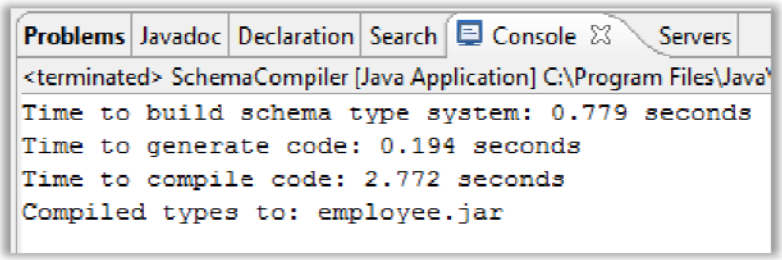

Figure A-8: Create the JAR File Based on the Sample XSD File

The next step is to replace the defect file `CodeGenUtil.class` in `xbean.jar` with a file in the `bin` folder of XMLBeans binary distribution. First, we fix

the bug in `CodeGenUtil.java` file. After installing Apache Ant, we run the build command to execute the `build.xml` included in XMLBeans source distribution as shown in Figure A-9.

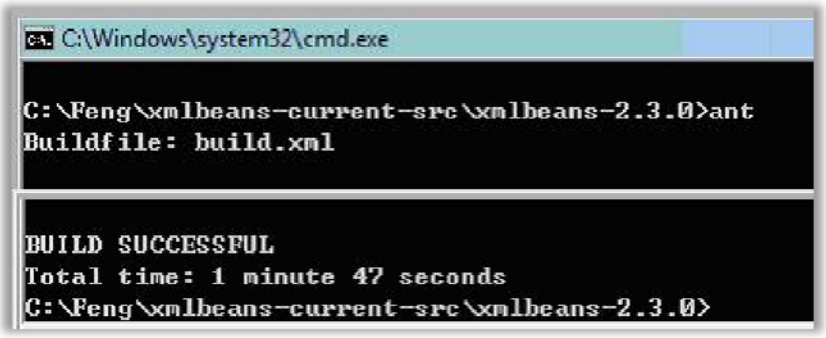

Figure A-9: Run Ant Command to Build Updated XMLBeans JAR Files

After replacing the `xbean.jar` file in the `lib` folder of XMLBeans home, we XMLBeans is functioning as shown in Figure A-10.

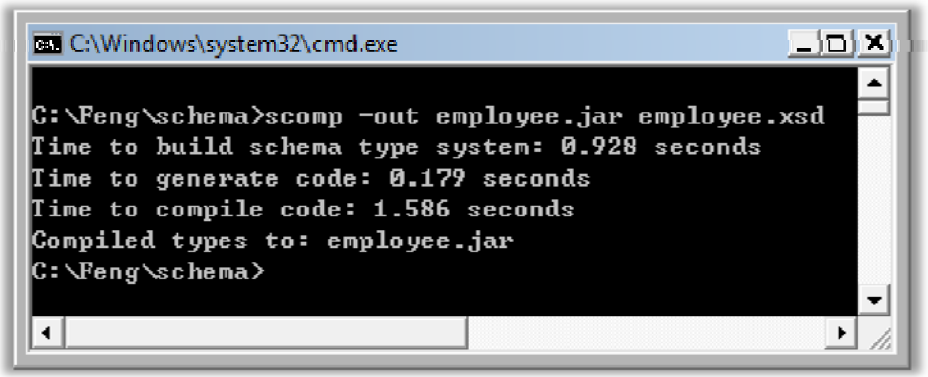

Figure A-10: XMLBeans is Functioning

Appendix B An XML Schema for Organization Charts

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
          elementFormDefault="qualified"
          targetNamespace="http://rigiuvic/organization"
                 xmlns="http://rigiuvic/organization">
   <xs:annotation>
      <xs:documentation xml:lang="en">
         XML schema for organization chart
         Copyright 2007 Rigi UVic. All right reserved.
      </xs:documentation>
   </xs:annotation>
   <xs:element name="organization">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="person"
                              type="personType"
                               maxOccurs="unbounded" />
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:complexType name="personType">
      <xs:sequence>
         <xs:element name="name" type="xs:string"
                           minOccurs="1" maxOccurs="1" />
         <xs:element name="id" type="xs:ID"
                           minOccurs="1" maxOccurs="1" />
         <xs:element name="title" type="xs:string"
                           minOccurs="1" maxOccurs="1" />
         <xs:element name="supervisor" type="xs:IDREF"
                           minOccurs="0" />
      </xs:sequence>
   </xs:complexType>
</xs:schema>
```