Model Driven Visualization: Towards a Model Driven Engineering Approach for Information Visualization

by

Robert Ian Bull
B.Math, University of Waterloo, 2000
M.Math, University of Waterloo, 2002

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

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

Model Driven Visualization: Towards a Model Driven Engineering Approach for Information Visualization

by

Robert Ian Bull
B.Math, University of Waterloo, 2000
M.Math, University of Waterloo, 2002

Supervisory Committee

Dr. Margaret-Anne Storey, (Department of Computer Science)
Supervisor

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

Dr. Yvonne Coady, (Department of Computer Science)
Departmental Member

Dr. Nigel Livingston, (Department of Biology)
Outside Member

Supervisory Committee

Dr. Margaret-Anne Storey, (Department of Computer Science)
Supervisor

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

Dr. Yvonne Coady, (Department of Computer Science)
Departmental Member

Dr. Nigel Livingston, (Department of Biology)
Outside Member

ABSTRACT

Model Driven Engineering (MDE) is an approach to software development by which software is specified, designed, implemented and deployed through a series of models. While the capabilities of MDE have been realized in many aspects of software development, there is no MDE supported technique for generating information visualizations. Information visualization is a technique that supports human cognition through interactive graphics by enabling users to identify data patterns more easily, summarize information or abstract concepts that are not easily comprehended from the underlying data. As more systems are designed using model driven engineering approaches there is now a need to support a model driven approach for creating such visualizations. This research explores the feasibility of a model driven approach to view creation that is compatible with the goals of MDE.

We approach the problem of developing an MDE technique for view creation in two ways. First, we examine how MDE technologies are used for specifying, designing, and

maintaining software systems to uncover the aspects of software customization that are supported through MDE. Second, we analyze six existing visualization tools to determine three functional requirements and six design recommendations for visualization creation and customization tools. Combining MDE principles and information visualization requirements, we propose Model Driven Visualization (MDV), a model based approach to view creation. MDV includes platform independent models for common visualizations, as well as a technique to generate platform specific instances of these models. Finally, using MDV we show that standard visualizations can be recreated in a concise syntax, that is compatible with the goals of model driven engineering.

MDV contributes to the fields of model driven engineering, information visualization and software engineering. In particular, this research 1) provides a collection of formal view models for common information visualization techniques, 2) outlines a method for designing and customizing information visualizations using MDE, 3) presents a code generation technique for integrating MDE with the model-view-controller pattern, and 4) contributes an open-source visualization toolkit to the Eclipse project.

Table of Contents

List of Tables

List of Figures

List of Listings

List of Abbreviations

MDE:	Model Driven Engineering	MDV:	Model Driven Visualization
PDE:	Plug-in Development Environment^{tm}	GMF:	Graphical Modeling Framework^{tm}
GEF:	Graphical Editing Framework^{tm}	EMF:	Eclipse Modeling Framework^{tm}
XML:	Extensible Markup Language	XMI:	XML Metadata Interchange
OCL:	Object Constraint Language	OMG:	Object Management Group
UML:	Unified Modeling Language	MOF:	Meta-Object Facility
MDA:	Model Driven Architecture^{tm}	HCI:	Human Computer Interaction
BPEL:	Business Processes and Execution Language	COTS:	Commercial Off The Shelf Software
API:	Application Programmer Interface	JET:	Java^{tm} Emitting Template
FoF:	Friend of a Friend		

Acknowledgement

First and foremost I would like to thank my supervisor Dr. Margaret-Anne (Peggy) Storey. Peggy, your patience, encouragement and wealth of great ideas will never be forgotten. Your enthusiasm extends to the entire Chisel group, of whom I am forever indebted. I could not have completed this work without every one of you.

I would also like to thank my examining committee. Dr. Nigel Livingston, Dr. Yvonne Coady and Dr. Hausi Müller, each one of you had a profound impact on me during my time at the University of Victoria. In addition to my supervisory committee, Mr. Bran Selic deserves a very special thank-you for agreeing to be my external examiner. The advice you offered me was invaluable.

I am very grateful to Dr. Marin Litoiu, Dr. Kelly Lyons and the IBM Centers for Advanced Studies (CAS), not only for the financial and technical support, but also the emotional support offered by the very encouraging CAS community.

Several members of the Eclipse community played an important role throughout my PhD, offering guidance, wisdom and advice. Thank-you Dr. Ed Merks, Chris Aniszczyk and Wassim Melhem.

Mom, Dad, Joanna and Jacquie, thank-you for always being there for me. How I ever became a doctoral student is anybody's guess, but I know I could have never made it this far without your support.

However, none of this would have been possible without the love and support given to me by my wife and best friend, Tricia. Tricia, before I started my PhD you told me that this was something I could do. It was your belief in me that gave me the confidence to undertake this task.

Finally, thank-you Sadie for giving me smiles and hugs at the end of every day.

Part I

The Problem

CHAPTER 1

Introduction

MODELING is a cornerstone of all traditional engineering disciplines [Sel03]. From conception and design, through construction and maintenance of any engineered system, modeling plays a critical role. Over the past 40 years, software engineers have explored how lessons learned from traditional engineering disciplines can lend themselves to the design, construction, deployment and maintenance of software.

Model Driven Engineering (MDE) is an approach to software development by which software is specified, designed, implemented and deployed through a series of models. This approach helps reduce accidental complexity [FPB87] through a more rigorous engineering process. Many MDE researchers propose that every aspect of software development can be modeled and that "everything is a model" [Béz05]. MDE formalisms have been employed in several areas of software engineering, including: the design of domain specific languages (DSLs) [vDKV00], database and data structure design [KR03], user interface specification [Kov98], business process design [PE00], and reverse engineering [Fav04b].

By extending the notion of MDE into all these areas, these interrelated components have been separated into manageable units [KR03].

1.1 Motivation

While the capabilities of MDE have been realized in many domains, the approach has not been universally applied to all aspects of a software system. This became evident while studying the effectiveness of Zoomable User Interfaces [FB95] with one of our industry partners in 2003. While the results of this study were inconclusive, the experiment itself resulted in some interesting insights into the importance of having a lightweight means of customizing information visualizations. At IBM Canada in Toronto, Ontario, we, along with members of the IBM User Centred Design team, planned the experiment, designed business models and performed the study. Concurrently, a group of researchers at the University of Victoria, in Victoria, British Columbia, customized the tools for use within the business process domain [RLS+03]. As the study was performed, initial feedback suggested that additional customization was required. Due to the geographical constraints between the experimenters and the tool developers, unclear requirements and other responsibilities, each customization requests required several days to complete. In each case, these customizations were simply a transformation of the underlying model or modifications to the attributes which describe how the data was to be visualized. Since the experimenters were not familiar with the tool, they were not able to make the modifications or customizations themselves.

To support tool developers facing similar customization problems, we investigated whether modeling technologies could alleviate many of the complexities, however, we discovered that there was no MDE technique for generating and customizing information visualizations.

1.2 The Problem

While MDE has been applied to many aspects of a software system, **there is no technique for generating information visualizations within the context of Model Driven Engineering.** We define information visualization as a technique that supports human cognition through interactive graphics by enabling users to more easily identify data patterns, summarize information or abstract concepts that are not easily comprehended from the underlying data. In contrast to scientific visualization, where the visualizations tend to be based on physical objects, information visualization tends to represent nonphysical information in an abstract form [CMS99]. Examples of information visualization includes interactive charts, tables, node-link diagrams, scatter-plots, and parallel coordinates, among others.

Software engineers are starting to realize the potential of information visualizations, which are becoming an integral part of many software systems. Since an increasing number of these software systems are constructed using a series of models and model driven engineering techniques, the lack of integration between MDE and information visualization means that these visualizations are often constructed as an afterthought, commonly integrated in an *ad hoc* manner. This disconnect between information visualization and the rest of the system requires developers to manage multiple development paradigms and leads to a lack of continuity across the system. As well, many of the benefits of MDE cannot be extended to the visualizations since they have been designed without a formal model, an important component in MDE.

Effective visualizations are often designed for particular users by highly creative designers working with complex data models. At the onset of this work, it was unclear if it was feasible to model views formally and that constraining the designers by imposing a formalism may not be an effective means for creating visualizations. However, by formally capturing the design of a visualization through models, we have shown that designers can share their design, validate their models and capture the users' requirements so that future developers can make sound choices before the system is deployed.

1.3 Goals and Objectives

As more systems are designed using model driven engineering approaches and end-users demand more interactive visualizations, the need to support a model driven approach for creating such visualizations is clear. The goal of this research has been to **develop a model driven approach to view creation that is compatible with the goals of MDE**. Software modeling and models in general are designed for five reasons: *abstraction, understandability, accuracy, predictiveness* and *inexpensiveness* [Sel03]. A model driven approach to the creation of visualizations should be compatible with these five criteria:

1. **Abstraction:** A model of a system should reduce the details of the system it represents.

2. **Understandability:** A model should be easier to understand than the system it models. In essence, a model must be presented in a manner than makes sense to those that use it.

3. **Accuracy:** A model must not lie. While abstraction may summarize or hide important details, the meaning of these details should not be altered by a model.

4. **Predictiveness:** A model should be executable in order to help an engineering predict specific system behaviour. In the case of mechanical engineering, engineers design models to test characteristics such as the interplay between forces. Software models should also be executable, allowing designers to test requirements such as memory constraints and user interaction.

5. **Inexpensiveness:** A model must be inexpensive to produce. If a model is more expensive than the system it is modeling, then few designers will take the time to properly model their system. A model should allow designers to construct an accurate prototype efficiently, that is easy to understand, modify and test.

Specifically, the goals of our research has been to answer the question: **can visualizations be generated and customized using MDE techniques?** In particular, we set out to meet the following objectives:

1. Develop a technique for formally modeling visualizations;
2. Develop tool support for our technique;
3. Demonstrate how the approach can be used to create key visualizations; and
4. Evaluate our approach against the goals of MDE.

1.4 Approach

We approach the problem of developing a model driven approach to view creation in two ways. First, we examine how MDE technologies are used for specifying, designing, and maintaining software systems to uncover the aspects of software customization that are supported through MDE (Chapter 2). In particular, we explore how models, constraints and transformation languages can be leveraged for software development. Second, we analyze how information visualizations are customized (Chapter 3) by studying how six existing visualization tools support view creation (Chapter 4). From this analysis, three functional requirements and six design recommendations for the customization of visualizations are revealed. These requirements helped us formalize a series of platform independent view models (Chapter 5), as well as a platform specific toolkit for information visualization (Chapter 6). Combining these aspects from MDE and information visualization, we propose **Model Driven Visualization** (MDV), a model based approach to view creation (Chapter 7). Finally, using MDV, we show that standard visualizations can be recreated in a concise syntax that is compatible with the goals of model driven engineering (Chapter 8).

1.5 Scope

The focus of this work is to define formal models of information visualizations. By defining these models, we draw a connection between information visualization and model driven engineering. Many of the examples in this thesis focus on graph based visualizations. Graph based visualizations have been shown to be useful in a wide variety of domains including: biology, chemistry, object-oriented systems, data structures, real-time systems, knowledge-representation, project management, logic programming, and circuit schematics [HMM00]. In the fields of software maintenance, reverse engineering and reengineering, almost 70% of respondents to Koschke's survey reported using some form of graph based visualization in their tool [Kos03]. This dissertation has direct applicability to these researchers, particularly with the notion of Model Driven Reverse Engineering [Fav04b] on the horizon. While many of our examples focus on graph based visualizations, we show how other visualization techniques, such as charts and tables relate to MDV.

1.6 Evaluation

We evaluate our model driven approach to view creation with respect to our goal and objectives. First, this work is evaluated in terms of its practicality. A view is constructed using traditional view customization techniques and then reproduced using Model Driven Visualization (MDV). From here, a number of lessons learned are synthesized. Second, the work is evaluated from the perspective of model driven engineering. The five characteristics of a model are presented and we discuss how MDV meets these criteria. Finally, we discuss the limitations of this approach.

1.7 Contributions

This research directly contributes to the fields of model driven engineering and information visualization. For the model driven engineering community, this work demonstrates the feasibility of modeling visualizations by integrating these views into systems currently being built using MDE. We provide advice for how MDV can be used to design effective views. In addition, this work presents a formal specification of 12 common visualization techniques.

For the visualization research community, this work presents a notation that can be used to create, maintain and share customized visualizations. This will better enable practitioners and other researchers to reproduce previous results. The combination of platform independent view models, platform specific implementations and formal view mappings, enables designers to create a working visualization prototype from a given data-set, with little or no programming effort. Another key contribution of this work is a new information visualization toolkit for the Eclipse integrated development environment [ECL]. This toolkit has been engineered so that designers and software developers can make use of graph based visualizations without using complex graph based Application Programmer Interfaces (APIs).

1.8 Organization of Thesis

The thesis is organized into three parts, The Problem (Chapters 1 through 4), the Solution (Chapters 5 through 7) and The Evaluation (Chapters 8 and 9) (Figure 1.1).

Chapter 2 (Model Driven Engineering) serves as a background on MDE. The problem of software customization is introduced and the application of MDE is explored as a possible solution. As well, an example system is built using models to demonstrate the practicality of MDE. Finally, the chapter outlines state-of-the-art research related to model based user interface design.

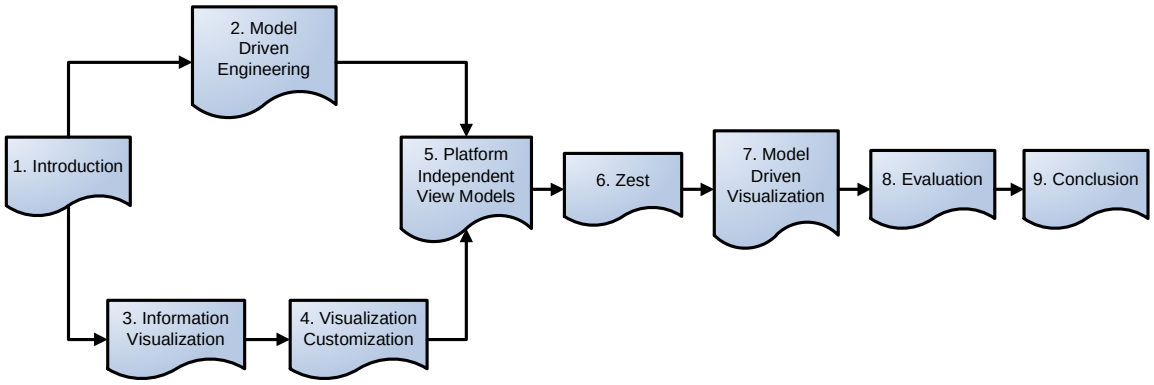

Figure 1.1: Thesis outline

Chapter 3 (Designing and Customizing Information Visualizations) examines the relevant background material regarding information visualization. The tools used to design visualizations, as well as several visualization taxonomies, are introduced.

Chapter 4 (A Study of how Visualizations are Created) examines the problem of view customization. In order to better understand view construction, five interactive visualizations are examined. In this chapter we look at the data structures the designers started with and how these structures are customized for presentation purposes. The chapter concludes by distilling a number of functional requirements and design recommendations for a visualization customization environment.

Chapter 5 (A Catalog of Platform Independent View Models) serves two purposes. For interaction design specialists, it demonstrates how visualization can be modeled and presents an extendable catalog of visualization techniques. For model driven software engineers, it presents a number of platform independent view models that can be leveraged to present complex information spaces.

Chapter 6 (Zest: A Visualization Toolkit for Eclipse) describes a platform specific graph based visualization toolkit. The toolkit is intended for Eclipse developers interested in adding node-link visualizations to their applications.

Chapter 7 (Model Driven Visualization) builds on our knowledge of model driven engineering and information visualization. This chapter focuses on how information visu-

alizations can be integrated into systems designed using MDE practices and demonstrates how visualizations can be modeled.

Chapter 8 (Evaluation) evaluates our model driven approach to view creation in the context of both MDE and information visualization. With respect to information visualization, we demonstrate the feasibility of our approach by reconstructing views that were originally designed using high-level programming languages and specialized tools. With respect to software modeling, we demonstrate how model driven visualization meets the goals of MDE.

Chapters 9 (Conclusion) summarizes this work, reiterates the contributions, outlines the limitations and presents some ideas for future research.

CHAPTER 2

Model Driven Engineering

As software systems become more complex, better abstractions and more maintainable software systems are sought. In the 1950's, software engineers turned to assembly languages to separate the program from the hardware for which it was written. In the 1950's through 1970's, high-level languages, like Fortran, Algo, Cobol, PL/I, APL, Pascal and C were introduced to further abstract the problem of writing software.

In the 1980's and 1990's, the concept of modules was introduced to improve how programs scale by offering better abstractions. One of the most successful examples of modules is the concept of the "Object" or object oriented systems. In these systems, everything is considered an "Object" and objects are first class citizens. This means that engineers can specify solutions in terms of the entities, operations on the entities and their interconnections. By raising the level of abstraction, these paradigms reduced the gap between the problem and the solution domains [GSCK04] (Page 60).

As software continues to grow in complexity, software engineers look for more abstract

representations of systems allowing them to truly specify solutions, rather than "program computers". Balzer was one of the first to describe a process of successive refinement from which an implementation could be gradually generated from a specification [Bal81]. Many other researchers built on his work proposing such systems as Computer Aided Software Engineering (CASE) [Cas85] and environments to allow developers to design software through pictures [Was90]. This early work lead many researchers and practitioners to focus on system modeling [Ken02, Sel03, Sel06].

Because of the reusable nature of models, software modeling is often used to provide customized software at a relatively low cost. This chapter introduces the notion of software customization through Model Driven Engineering (MDE) and explains how modeling can be leveraged to help support tool development. Section 2.1 describes the importance of software customization. Section 2.2 presents an introduction to MDE, and through a simple example shows how an application can be designed, customized and ultimately deployed using this approach. The core MDE concept, the model, is presented in Section 2.3. Technologies central to MDE such as constraints and model transformations are described in Sections 2.4 and 2.5. Section 2.6 explores some of the research related to model driven interface design and finally Section 2.7 presents limitations of software modeling.

2.1 Software Customization

Michaud sees the customization of software as an important part of the software lifecycle requiring the input and expertise of three main groups of stakeholders [Mic03]: 1) **developer** [1], 2) **users** and 3) **customizers**. Developers, i.e. tool-smiths, are responsible for creating the initial system and setting the stage for future customization. Users are at the other end of the spectrum as they ultimately use the application to perform tasks. The customizers [2] mediate these two groups tailoring generic software systems for particular

[1] Michaud used the term designer instead of developer, however, designer is a commonly used term especially in the context of human-computer interaction so we have chosen the term developer for clarity.

[2] Michaud also refers to a customizer as a Guru

Customization Style	Description
Options Screens	Uses check boxes, combo boxes, etc.
Wizards	Guides a user through a set of choices
Configuration Files	Settings are recorded in a separate textual format
Macros	Recorded user actions to automate common tasks
Visual Builders	Interactive diagram
Scripts	Custom or common script language
Application Generators	Interpret domain specific specifications to automate the generation of the software from pre build modules.

Table 2.1: Interaction Style Customizations

users and their needs. The three groups, developers, users and customizers, help classify stakeholders, but these groupings are an abstraction and in reality individuals often fall into more than one group.

The concept of software customization has also been used in software product lines in which a "family" of software products are all based on the same platform [Böc05]. The platform provides a number of customization points that designers can tweak to generate highly customized software.

Michaud's inspiration for software customization came from Wasserman who identified three levels of integration between two or more applications [Was90]. Michaud suggests that these three levels of integration, data, presentation, and control, are also appropriate categories to classify software customization [Mic03]. Data customization includes both the customization of the data format and data content. Customizing data is performed to make the application compatible with additional information sources and altering information for further processing. Presentation customization alters how information is organized and accessed through the display. The customization of presentation also includes the configuration of the aesthetics and "look and feel" of an application. Finally, control or behaviour customization tailors which features are available (through feature selection and feature addition), which options are enabled, how features are enhanced or constrained and finally, how multiple features are coordinated.

The customization categories, data presentation and control, also form the basis for the model-view-controller (MVC) architectural pattern. The MVC pattern was first popularized with Smalltalk-80. Smalltalk used this pattern to make a clear divide between domain objects and presentation objects. Since Smalltalk's use of MVC, many graphical toolkits include an MVC component so domain objects can be completely self contained and operate without reference to the presentation.

In addition to the three aspects of software that can be customized, two broad categories of software customization techniques have been outlined by Michaud [Mic03]: source code customization and interaction styles. Source code customization includes any activities that require the modification of the original source code. Interaction styles include customizations that do not require modification of source code such as options screens, wizards, or configuration files. Table 2.1 outlines seven interaction style customizations.

While these two broad categories seem reasonable, the distinction between them is starting to blur. End user products like the Firefox web browser, Microsoft Vista and even Facebook, provide scripting languages and sample scripts (called "add-ons" or "extensions") that users can install to create a personalized work environment. Users are not only installing existing scripts (which according to Michaud would be interaction style customizations), but they are also adapting the scripts for their own needs. We have taken inspiration from Michaud's categories, but instead of considering them as mutually exclusive groups, we have created a continuum of configuration techniques and loosely grouped these interaction techniques into three overlapping categories: 1) User Interface (UI) supported customization, 2) Domain Specific Language (DSL) supported customization and 3) customization supported through source code modification(Figure 2.1). The methods of customization in Figure 2.1 are ordered by the flexibility of the approach. Scripting languages, or end user programming, are often seen as a good compromise between ease of use and flexibility [TMWW93, TWSM94].

UI and DSL supported customizations have been used extensively in software development. As mentioned earlier, software customization through application generators was

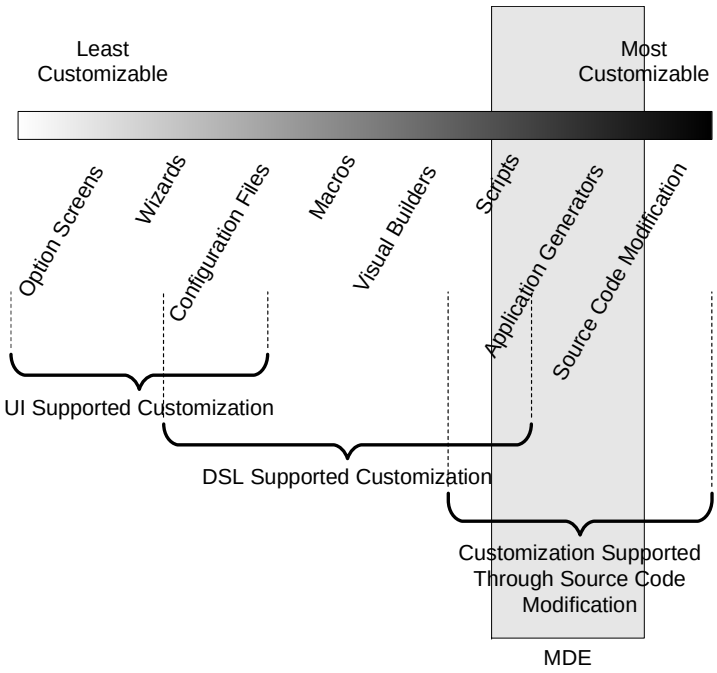

Figure 2.1: Methods of Customization

first made popular in the 1980's and 1990's with Computer Aided Software Engineering tools (CASE). While an ingenious idea at the time, many CASE tools suffered because the all-purpose graphical language used to design systems was too general and did not map well to problem domains. As well, the translation technologies available at the time produced code that was too complex, making it hard to develop, debug and maintain [Sch06]. Template driven programming, on the other hand, is just starting to emerge as a mainstream option for software development and configuration. In this case, application generators create boilerplate code from templates through the interpretation of a domain specific model, and then the boilerplate code can be customized, extended or used within other applications. The OMG's standardization of Model Driven Architecture^{tm} and tools like the Eclipse Modeling Framework and Open Architecture Ware (oAW) [BSM$^+$03, Voe] have paved the way for application generators and template driven software customization. To better understand how tools like these assist with software customization, we examine the concepts behind model driven engineering.

2.2 An Introduction to Software Modeling

Software development is not easy. Software engineering encompasses many facets such as requirements analysis, software architecture design, implementation, testing, deployment and maintenance [Som04]. Several issues can impede these activities making software development a great challenge, including [BCT05]:

1. A full understanding of the problem space is rarely achieved;
2. Many conflicting constraints must be resolved;
3. Software development is a dynamic process in which change must be addressed throughout all stages of development; and
4. An engineering process must be employed to manage risk, return on investment analysis, documentation, resource management, etc.

One of the most fundamental techniques used by software engineers to address these challenges is the use of models and modeling. "*Models are used to reason about a problem domain and design solutions in the solution domain* [BCT05]." As Wegener explains, modeling helps alleviate many of these challenges by [Weg02]:

1. reducing the cost throughout an application life-cycle;
2. reducing the development time for new applications;
3. improving application quality; and
4. increasing the return on investment.

According to Schmidt, "*Model-driven engineering technologies offer a promising approach to address the inability of third-generation languages to alleviate the complexity of platforms and express domain concepts effectively* [Sch06]". To alleviate this complexity using MDE, software engineers design Platform Independent Models (PIMs), and translate

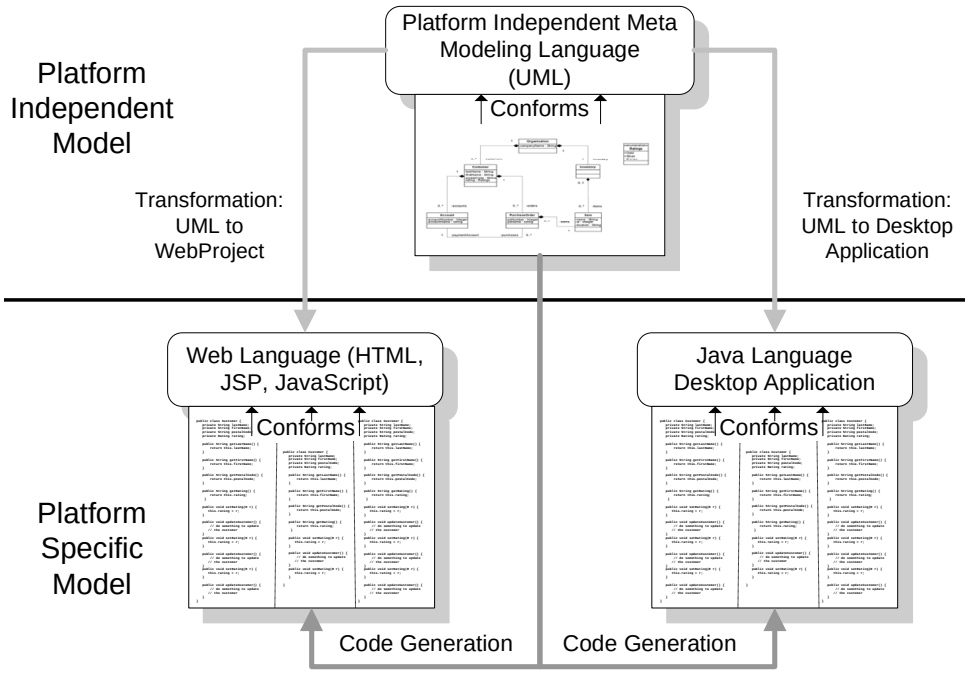

Figure 2.2: Overview of the MDE Process

these PIMs to Platform Specific Models (PSMs) [Met05, GGZ+05]. A platform independent model is a model of a system that does not contain technical details about the solution, such as, the implementation language, database technology and even delivery method (web-based or rich client).

Modeling software in a platform independent manner enables engineers to design solutions without concerning themselves with the implementation details. For example, a developer may specify a set of classes such as a customer purchase order tracking system, and using MDE, these classes can be generated for different platforms such as a C++ application targeted to run under Linux or as a Java application configured to work with a web framework. By building software in this manner, engineers can design, prototype and simulate their system before choosing their execution environment. Depending on the results of the simulation, a target environment (or multiple target environments) that meets their constraints, can be chosen.

In order for the models to be shared amongst developers and to support tool interoperability, the PIMs must conform to a standard notation [Béz05]. Several industry standards have emerged, such as the Unified Modeling Language (UML) [OMG07b], MOF [OMG06a] and XML Schema [W3C04]. Figure 2.2 illustrates how the same specification can target several different platforms. In this example, the same UML model can be used to generate a Java rich client system or a web application. To support both platforms, a transformation must exist to describe how the UML model can be translated to both Java and web based platforms. A code generator, can then be invoked to create the system [SwJBHH06]. By using existing models and transformations, engineers can abstract the concept of "programming computers" to the concept of solution modeling [SK02].

2.2.1 Example System

To illustrate how MDE is used in practice, we describe an example customer management system. The customer management system (CMS) will be used to track customers, their accounts, and their purchases. The CMS will also allow sales representatives to associate a rating with each customer such that a **gold** star customer will receive a 25% discount, a **silver** star customer will receive a 10% discount and a **bronze** star customer will receive a 5% discount.

Figure 2.3 shows a platform independent model for this system. Since the diagram is a more abstract representation than the code, it is easier to inspect and ensure it matches the designer's conceptual model.

In order to implement this system, several other decisions must be made, such as choice of platform. If the system is to be deployed as a web application the designer must further decide on which web technologies to use (such as JSP, .NET or the Google Web Toolkit (GWT)). Another decision concerns data persistence: XML or a relational database. These are all examples of platform specific issues. Software development processes, such as the waterfall or spiral model [Roy70, Boe88], require these decisions be made up-front (after considerable analysis), because development cannot continue until these platform specific

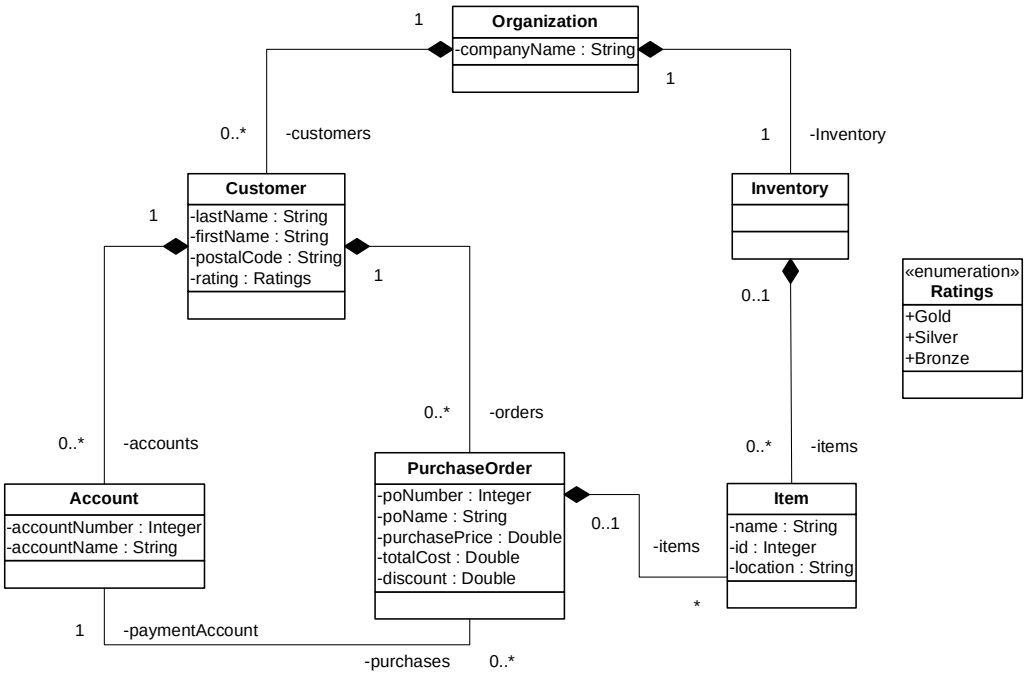

Figure 2.3: Platform Independent Model for Customer Management System

issues have been resolved. These are all decisions that engineers would rather delay until the last possible moment since changing them often incurs a great cost.

To account for changing requirements, agile development practices, such as eXtreme Programming (XP) [Bec99] encourages the constant refactoring [Fow99] of the code. However, platform decisions (such as choice of language or persistence technology) are very difficult to change once development has started. Model driven development supports the construction of the model independently of the platform, and through the use of transformations, the platform specific implementation is generated. From a single UML class diagram (such as the one in Figure 2.3), multiple implementations (such as a Java and C++ version) can be generated, and both versions can be tested for performance, scalability and other non-functional requirements.

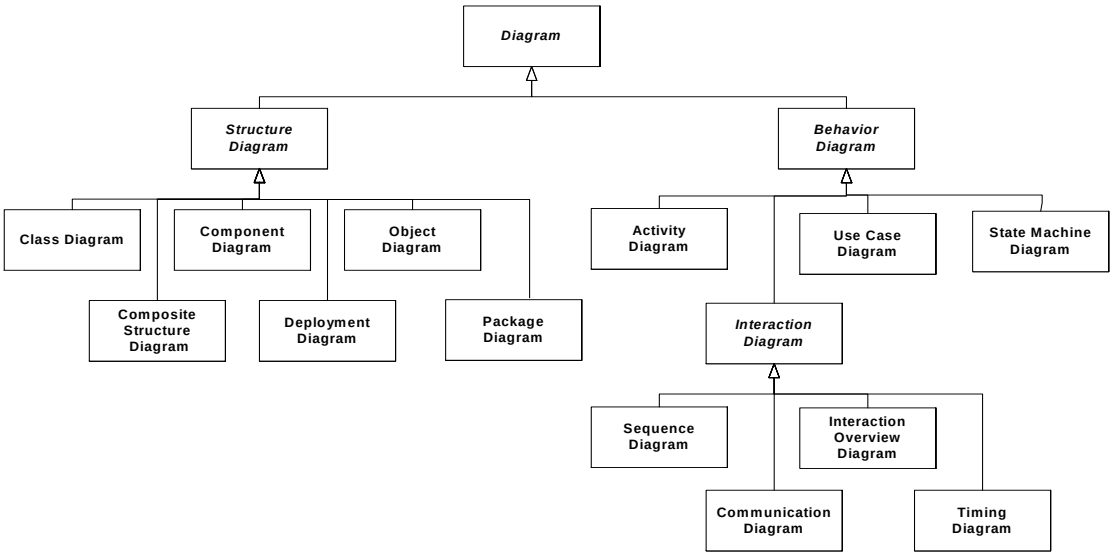

Figure 2.4: Hierarchical Classification of the 13 UML Diagrams

2.3 Models

Software models are used to help engineers design and reason about complex software systems. Software models are often informal sketches that are discarded once the code is written. While this approach may be sufficient for code-centric software development, with a model-driven approach, the models themselves become the primary software development artifacts [Sei03]. To support model-driven development, a family of standard software modeling languages has emerged. Included in this "family of models" is the Unified Modeling Language (UML), an object oriented modeling language that includes a graphical syntax. UML is used to create an abstract model of a system.

UML currently has 13 diagram types that can be grouped into three categories: 1) structure diagrams, 2) behaviour diagrams and 3) interaction diagrams. Figure 2.4 shows the hierarchical classification of the 13 UML diagram types. Arguably the most common UML diagram is the class diagram. The class diagram represents the system's classes, attributes, and relationships between classes. In particular, the class diagram can be used to model **association**, **aggregation**, **composition** and **generalization** relationships [OMG07b].

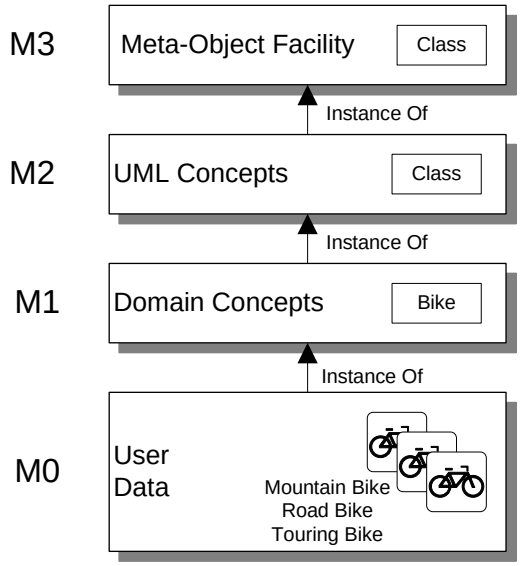

Figure 2.5: Model Management Four Layer Architecture

While UML is a platform independent modeling language, it is geared towards object oriented systems and not necessarily ideal for representing all types of models such as data warehouses, workflow, software process, etc [Béz01]. To combat the emergence of several incompatible modeling languages, a meta-modeling notation, Meta-Object Facility (MOF pronounced Mof) [OMG06a], was specified. A meta-model is a model of a modeling language. By using MOF compliant models (such as UML), designers are ensured that all their models will be compatible as models can be transformed from one language to another.

Once a model has been specified, it can be instantiated. If properly instantiated, the instance is said to "conform" to the model. A system can be designed using UML by creating a number of UML models to represent the system, and then generating (either by hand or through an automated process) a software system that conforms to the model.

To illustrate the relationship between software models, meta-models and the systems under study, a four-layer architecture and meta-modeling foundation was proposed [AK03] (Figure 2.5). In this architecture, MOF can be used to define a series of compatible modeling languages. These modeling languages (such as UML) can then be used to describe

domain specific concepts. Finally, user data can be instantiated.

Other standard formats for software have also been used. Rigi and the Portable Bookshelf (PBS) [MK88, FHK+97], two software reverse engineering tools, use the Rigi Standard Format (RSF) [Ken96] and Tuple-Attribute (TA) [Hol97] formats respectively. Both of these formats represent software artifacts as a set of triples (e.g. **call main printf** to represent that main calls printf). To combat the number of incompatible reverse engineering tools that have emerged, a general purpose graph exchange language (GXL) [GXL] has been proposed.

2.4 Model Constraints

Models themselves are not sufficient to fully represent a complex system. Models contain information about the elements, attributes and relationships in a system, however, they say very little about the constraints. Model constraints are important to enforce data integrity and to eliminate redundancies. Class diagrams, for example, can constrain the cardinality among members, however, more advanced constraints are often needed. An example of data integrity from the CMS example is that no two accounts can have the same account number. While a model can describe an account number and its properties, using a UML class diagram alone there is no way of expressing this constraint. An example of how constraints can help eliminate redundant data is to indicate that a customer's **full name** is actually a concatenation of their **first name** and **last name**.

2.4.1 Object Constraint Language

Constraints can be specified in a number of ways including natural language, predicate logic or a high level programming language such as C++ or Java. While natural languages are traditionally chosen, the ambiguities of these languages make them a poor choice. Predicate logic does not suffer from these ambiguities, but its use is often difficult to master. Standard computer programming languages, such as C++ or Java, are often seen as a good

Type	Values	Operations
Boolean	true, false	and, or, xor, implies, if-then-else
Integer	-10, 4, 8, 16, 23, 32, 42	*, +, -, /, abs()
Real	1.5, -3.14,	*, +, -, /, floor()
String	“The quick brown fox”	concat(), size(), substring()

Table 2.2: OCL predefined types

compromise, however, choosing these languages tends to tie modelers to a specific platform. The Object Constraint Language (OCL) [OMG06b] has consequently been proposed as a platform independent way to specify constraints on an object model. OCL is a standard put forth by the OMG, the same standards body working on UML and other modeling standards. OCL is suited for both specifying invariant conditions that must be true when the system is validated, as well as specifying queries over objects in the system.

An example of an OCL constraint for our customer management system is that each **Item** in our **Inventory** must have an ID greater than zero. This can be specified by adding the following constraint to the Item class: `self.id > 0`.

OCL has several predefined types available for modelers to use. The basic OCL types are shown in Table 2.2.

OCL also supports the notion of a **Collection**. A Collection is an abstract type with **Set**, **Sequence**, and **Bag** as the concrete subtypes. A Set is similar to a mathematical set in that it does not contain duplicates. A Bag is similar to Set, although it can contain duplicates and a Sequence is like a Bag in which the elements are ordered. Table 2.3 shows the operations available on the collection types.

OCL is emerging as the standard constraint language for UML based models. OCL originated in IBM’s insurance division [FC01, p. 24], making it more suited for business modelers than a highly mathematical formal languages like predicate logic.

To demonstrate how OCL can be used in practice we will add several constraints to our working example. Consider the **Customer**, **Account** and **PurchaseOrder** classes presented in Figure 2.3 from the customer management system described in Section 2.2. From

Operation	Usage
select(boolean-expression)	Selects a subset of a collection whose criteria matches the selection criteria
reject(boolean-expression)	Removes all elements who criteria matches the boolean-expression
collect(expression)	Returns a collection derived from the original collection by applying the expression to each element.
forAll(boolean-expression)	Returns true if for each element in the collection, the boolean-expression evaluates to true; return false otherwise.
exists(boolean-expression)	Returns true if the boolean-expression evaluates true for at least 1 element in the collection.
isUnique(body)	Returns true if for each element in the collection the body evaluates to a unique value
any(body)	Returns any element in the source collection for which body evaluates to true
one(body)	Returns true if there is exactly one element in the source collection for which body evaluates to true
sortedBy(body)	Returns a set in ascending order.

Table 2.3: Built-in OCL Operations

```
self.customer.rating = Ratings::Bronze implies self.discount >= 5.0
self.customer.rating = Ratings::Silver implies self.discount >= 10.0
self.customer.rating = Ratings::Gold implies self.discount >= 25.0
```

List of Listings 2.1: Customer Ratings Constraints

these three classes it would be acceptable to create a data-set where one customer's account was paying for another customer's purchase. While some customers may consider this a nice gesture, this behavior is undesirable. In addition to this constraint, several other constraints are needed for our simple example. In particular, it is important to ensure that purchase order numbers, account numbers, and stock numbers are unique. Table 2.4 shows how these constraints can be specified in OCL.

Constraints can also be specified to ensure that a customer's discount is computed correctly. Listing 2.1 shows how the discount amount can be verified using OCL.

Constraint	Class	OCL Specification
Unique ID	Inventory	self.items→ isUnique(id)
Unique customer account numbers	Organization	self.customers.accounts→ isUnique(accountNumber)
Unique purchase order numbers	Organization	self.customers.orders→ isUnique(poNumber)
Customers only pay for their own purchases	Account	self.purchases→ forAll(po|po.customer=self.customer)

Table 2.4: Basic Customer Management System Constraints

2.5 Model Transformations

Model transformations are essential for realizing the power of MDE [SK02]. Model transformations transform data instances conforming to one model to data instances conforming to another. There are several varieties of model transformations including text to model transformations, model to text transformations, and model to model transformations. Text to text transformations (like those supported by TXL [CDMS01]) are beyond the scope of this work, since these transformation are not necessarily related to the modeling of visualizations.

2.5.1 Text to Model Transformations

Text to model transformations are concerned with the transformation of strings (or text files) to models [Voe, JBK06]. These transformations are designed using an annotated grammar which specifies how strings are parsed and which model elements relate to which part of the language. Text to model transformations are a key component for creating domain specific languages [vDKV00].

```
<<FOREACH features AS f>>
        private <<f.type.name>> <<f.name>>;
        public void <<f.setter()>>(<<f.type.name>> <<f.name>>) {
                this.<<f.name>> = <<f.name>>;
        }
        public <<f.type.name>> <<f.getter()>>() {
                return <<f.name>>;
        }
<<ENDFOREACH>>
```

List of Listings 2.2: Model to Text Generator for an Accessor / Mutator Pair

2.5.2 Model to Text Transformations

Model to text transformations describe how a model should be written as a string or text file. In contrast to text to model transformations, in which the transformation describes how plain text is transformed to a model, these transformations specify how model elements should be written in plain text. Listing 2.2 presents an examples of how an accessor / mutator pair could be generated for an object-oriented system.

2.5.3 Model to Model Transformations

In order for software systems to truly realize the potential of platform independence, models must exist for several target platforms and model to model transformations [BH02] must be designed to convert object models from one canonical form to any platform specific instance. MDE proponents expect that writing model to model transformations (hereinafter known simply as model transformations) will become a common task in software development and these transformations will be shared among engineers [JK06].

Figure 2.6 presents an overview of how model transformations are applied to transform a source model to a target mode [CH06]. Both the source data (Da) and the target (Db) conform to their respected models (Ma and Mb). The transformation is driven by a transformation engine. The transformations are written in terms of the source and target models, which means that a particular transformation can be executed on any system conforming to Ma and produce a system conforming to Mb.

There are a number of strategies for model transformations. Since software models

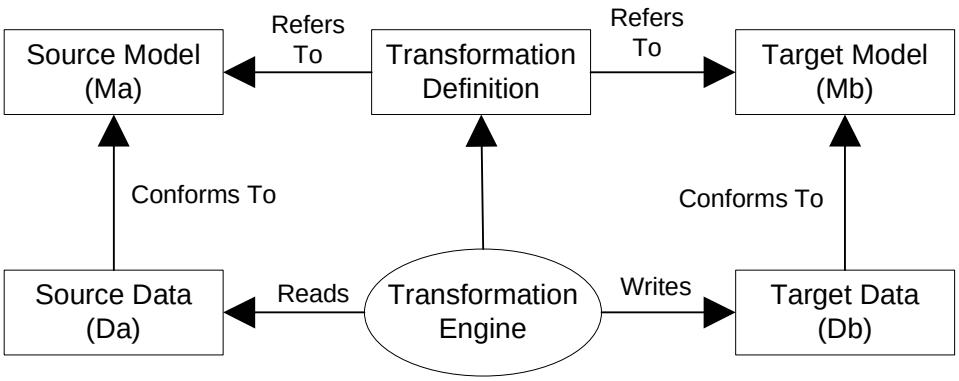

Figure 2.6: Model to Model Transformation

usually have a graph based representation, traditional graph transformation theories can be applied. Many of the approaches stem from triple graph grammars [Kön05, GGL05] specified by Schürr in 1994 [Sch]. Triple graph grammars use three graph grammars [Pra71] to specify transformations. The first graph grammar defines the input graph. The second graph grammar defines the output graph, and the third graph grammar defines the correspondence graph, which tracks the correspondence between the input and output graph.

Czarnecki classified a number of approaches to model transformations [CH03, CH06]. Using feature diagrams, Czarnecki created a classification that represents the areas of variation within a domain of model to model transformations. Some of the areas include: types of transformation rules, rule scoping, rule scheduling, and tracing. Regarding transformation rules, Czarnecki indicates that rules can either be declarative, imperative, or a combination of the two. Declarative languages, like XSLT [W3C99] are more common and often express the transformation rules in a concise syntax. Declarative languages simply "declare" how elements in the source are mapped to elements in the target. Imperative languages, on the other hand, are usually more powerful but achieve this at the cost of complexity. Imperative languages are Turing complete, meaning they can compute every Turing-computable function and are as powerful as traditional high level languages. Most transformation languages support both an imperative and declarative syntax.

Metzger presents a classification of the different types of model mappings [Met05].

Metzger indicates that model mappings fall into one of three categories: *non-ambiguous mapping*, *selection* and *approximation*. Non-ambiguous mapping means that elements of the source can be directly mapped to elements in the target. A selection mapping means that a subset of elements in the source are mapped to the target. Finally, approximation indicates that no semantically equivalent elements exist in both the source and target models. The last two cases require additional processing such as, filtering or the specification of how the approximated target elements are derived from the source.

To support transformations of models such as UML, the OMG has created a standard for transformation languages known as Query View Transform (QVT) [OMG07a]. Unlike triple graph grammars, the source and targets in a model transformation language (like QVT) are specified in UML or some other MOF compliant model, and the transformation pre- and post-conditions are specified in OCL. Triple graph grammars and model transformation languages present two different approaches for transforming graph based structures, and it has been shown that both approaches have equivalent power [GK07].

While QVT is the ideal solution for our work, at this time there is no reference implementation available. Since there is no QVT implementation available, we have chosen the Atlas Transformation Language (ATL) [JK06] in order to document our transformations. ATL is considered very close to the QVT specification. While we have chosen ATL, it should be noted that any graph transformation language could be used and we have also replicated many of these transformations using other languages such as Open Architecture Ware's (oAW) XPand [VK06].

Listing 2.3 shows an example declarative transformation rule that maps a Class with attributes to an HTML table. When this transformation is applied to our customer model, the HTML page in Figure 2.7 is produced. By using this transformation, software engineers can easily keep on-line documentation synchronized with their models.

```
rule ecore2html {
        from s : ECore!EClass
        to t: EHTML!List (
                title <- s.name,
                items <- s.eAttributes,
                items <- s.eReferences
        )
}

rule reference2listelement {
        from s : ECore!EReference
        to t : EHTML!ListItem (
                value <- 'get' + s.name.firstToUpper() +'()'
        )
}

rule attribute2listelement {
        from s : ECore!EAttribute
        to t : EHTML!ListItem (
                value <- s.name
        )
}
```

List of Listings 2.3: Simple transformation from ECore to HTML

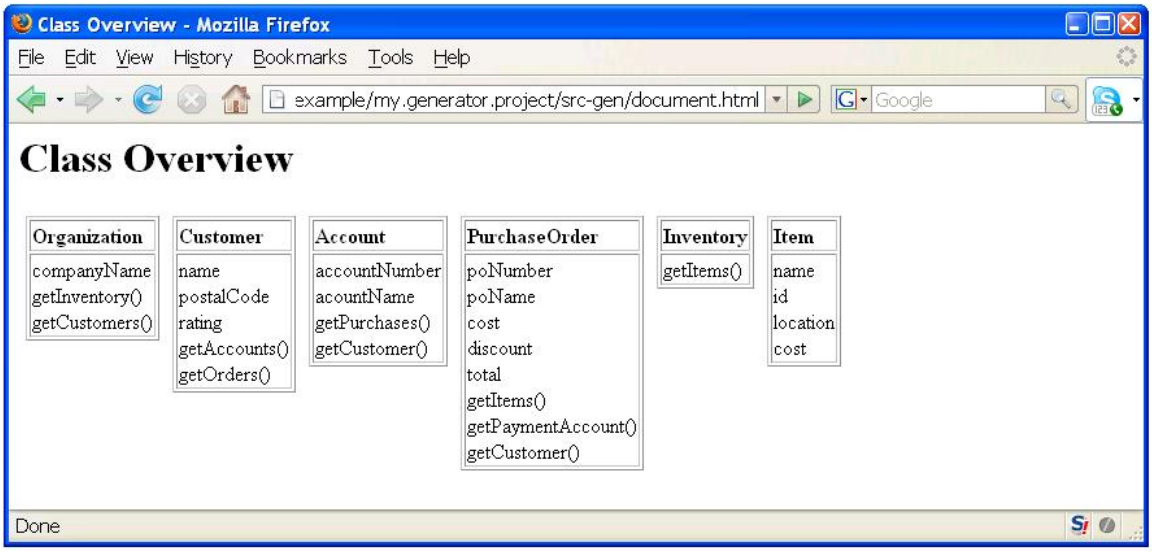

Figure 2.7: HTML Showing Class Overview

2.6 Model Driven User Interfaces

Several researchers and practitioners have extended the notion of model driven engineering to include the design and customization of user interfaces. Kovacevic studied how UML models can be extended to incorporate user actions and interface controls [Kov98]. Kovacevic argues that software models often focus on the internal data structures used by the system, and rarely assist with the design and development of user interfaces. However, studies have shown that in some systems, nearly 50% code is dedicated to the UI [Mye89, Kov98]. Kovecevic proposed additional UML modeling constructs to enable UI design support [Kov98].

Bodart *et al.* presented an approach to UI design based on task analysis [BHLV94]. His work, Tools foR an Interactive Development EnvironmeNT (TRIDENT), preceded MDE but the contribution is still significant. The proposed series of models and tools to support a task analysis process for UI design.

Xulin Zhao *et al.* studied how business process models can be leveraged to generate E-Commerce user interfaces [Xul07]. While not specific to MDE, this work demonstrates the importance of integrating processes and tools within existing platforms using languages familiar to the designer.

Finally, the Graphical Modeling Framework (GMF) project [GMF] enables software modelers to design and generate a graph based user interface from a model. While similar to graph based information visualizations, GMF focuses on data editors. The greatest challenge in GMF is how to keep data and edit model synchronized. In the case of information visualization, views are only used for displaying information and the challenge here lies in how abstractions and summary information can be modeled.

2.7 Drawbacks of MDE

Model Driven Engineering is suitable for the design, generation and customization of software systems which embody common patterns and recurring solutions. Using MDE for the development of a CMS system is ideal, as customer management systems are used in a variety of circumstances, and the models, constraints and transformations, can be reused effectively. However, some argue that MDE is less suited for one-off solutions which often do not require much, if any, customization [Gre06].

As in any modeling activity, it is vital that all parties involved interpret the specifications the same way. However, the existing specifications, such as UML, have been criticized for their complexity, a situation which has led to many ambiguities [HS05] and the term *metamuddle* [FGDTS06].

Another criticism of MDE is that the "*specification of a development problem is not discovered, it is created as the primary artifact*" [Weg02]. This requires considerable up-front cost, a cost that may never be realized especially in problem domains where reusable patterns and common practices have yet to emerge [Gre06]. Some critics have claimed that developers working with modeling tools are not any better able to understand the problems then they were 20 years ago and that "*system requirements can never be stated fully in advance, not even in principle, because the user does not know them in advance - not even in principle*" [Sne07, MJ82].

Since formally describing the entire behaviour of a system up front requires considerable cost, modeling has mostly focused on static structures. The approach MDE has provided for defining and integrating behaviour, an important component in any software system, is seldom utilized effectively [HS05]. There are a number of reasons cited for this, such as complex notations, lack of round-tripping tools and poor integration with the software lifecycle [HS05].

Finally, MDE requires full buy-in from all the stakeholders. Many tools assume models are being used for all aspects of the system, and mixing modeling and non-modeling tech-

nologies is a challenge. This is because "round-tripping", going from a model to code and back to the model, still contains a number of open problems. Many of these problems are currently being addressed in tools such as: *From UML to Java And Back Again* [BGH+05], but will likely not impact mainstream software development for a number of years.

Even with improved round-tripping support, mixing development paradigms is undesirable. While undesirable, mixing paradigms is unavoidable for aspects in a software system, such as information visualization, that are seldom designed in a way compatible with model driven engineering.

2.8 Summary

Model driven engineering is a software engineering process in which a system is designed, built and deployed through a series of models. In this chapter, we introduced the concept of software customization and discussed how software modeling, and MDE in particular, can play a role in this process. Several MDE concepts were introduced including the model, model constraints, model transformation languages and model driven user interface design. Finally, the drawbacks of MDE were presented. In particular, the integration problem between modeled and non-modeled aspects of a system was discussed. One area in particular that is affected by this integration problem, is information visualization. In the next chapter we will investigate how information visualizations are created and customized to better understand the ways in which view models can designed.

CHAPTER 3

Designing and Customizing Information Visualizations

ADVANCED visualization techniques are thought to be useful in the fields of program comprehension [SWFM97, PBS93], knowledge engineering [Ala03, ERG02, SMS+01, MG03], financial analysis [Wat99], business process design [RLS+03] and many other disciplines [CMS99, HMM00], assisting both with information processing and information communication. Effective graphic communication assists with transcribing relevant data to answer pertinent questions and help communicate what has been discovered [Ber81].

An effective visualization makes use of our cognitive ability to process visual information. A visualization supports thinking through visual queries on information and by extending one's memory [War04]. Visual queries work by providing visual patterns, that if detected, help solve part of a problem [War04]. Extending one's memory, on the other hand, is achieved by the way an image (or pattern) can rapidly trigger long-term memory to retrieve nonvisual information. In recent years, computer interface technology has improved greatly, and the notion of computer aided visualization has evolved. With higher

1. **Data Visualization:** The use of computer-supported, interactive visual representations of data to amplify cognition.
2. **Scientific Visualization:** The use of interactive visual representations of *scientific data*, typically physically based, to amplify cognition.
3. **Information Visualization:** The use of computer-supported, interactive, visual representations of abstract data to amplify cognition.

Table 3.1: Visualization Definitions

resolution displays and faster response times, computers have enabled designers to rapidly create highly interactive visualizations.

In this chapter we examine the construction and customization of information visualizations. The chapter begins by defining information visualization (Section 3.1) and the technology which enables it (Section 3.2). The chapter then explores how visualizations are currently classified (Section 3.3) and discusses the limitations of these classification systems. The chapter concludes with a look at how visualizations are currently designed and customized (Section 3.4).

3.1 Information Visualization

Data are presented visually for two purposes: 1) for communicating an idea, and 2) for discovering the idea itself. While graphics have been utilized for both thinking and communication for thousands of years, the evolution of the computer has dramatically changed the way these graphics are 1) designed, 2) rendered, and 3) interacted with [CMS99].

With the advent of computer graphics, the notion of *information visualization* has emerged. Information visualization has been defined as **the use of computer-supported, interactive, visual representations of data to amplify cognition** [CMS99]. Data are often considered "plain facts", while information is the result of processing, organizing or structuring the facts to make them meaningful. In contrast to data visualization (defined in

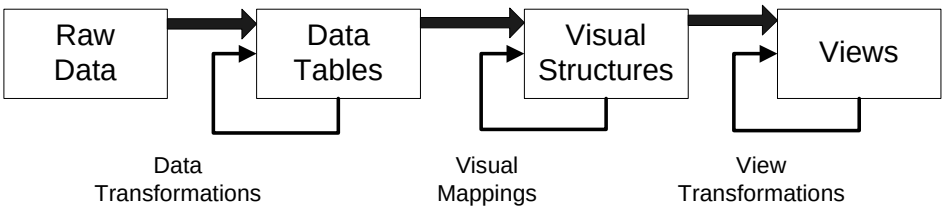

Figure 3.1: Reference Model for Visualization

Table 3.1 [CMS99]), information visualization focuses on the representation of **abstract data** or processed data. As well, information visualizations focus on abstract concepts as opposed to scientific visualizations that display physically based entities such as the earth or the human body.

Since abstract or processed data are vital to information visualization, the concept of data transformations, which typically involves the loss or gain of information, is a central component in the visualization reference model (Figure 3.1). In this model, raw data are mapped to data tables where it is continually refined until the desired, more abstract representation, is achieved. Once the desired level of abstraction has been reached, visual attributes can be associated with data elements and views created. These views can then be rendered and presented to the user.

3.2 Enabling Visualizations Through Technology

Information visualizations make use of both computing hardware and software to render static and interactive digital images. Building effective visualizations that can help users with their tasks is an ambitious undertaking. Not only do visualizations need to be designed and prototyped, they need to be implemented using robust tools that scale to large data-sets without compromising performance. The visualizations must also be tested to ensure they actually assist with the tasks users perform. There are several toolkits available to help software engineers construct such visualizations.

Widget toolkits such as SWT [NW04], AWT [AWT], and GTK+ [GTK] provide com-

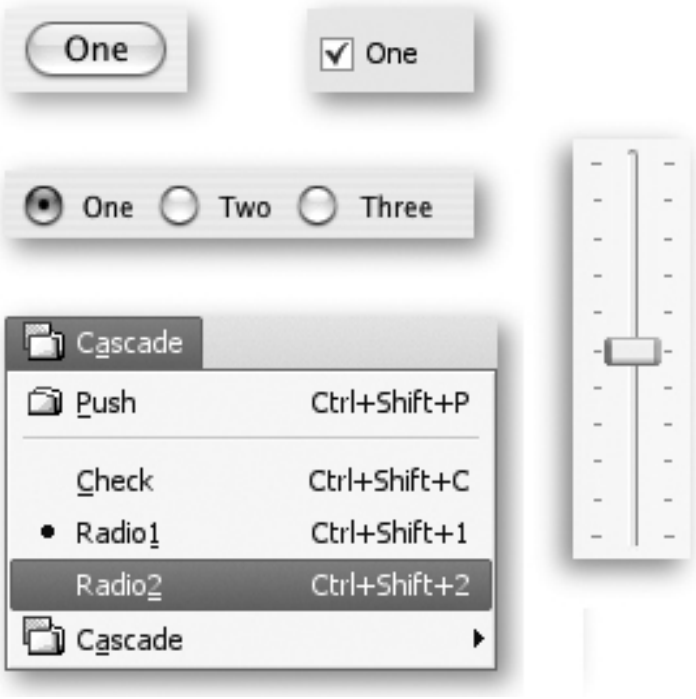

Figure 3.2: Sample Widgets

ponents, also known as widgets, for building User Interfaces (UIs). Examples of widgets include buttons, checkboxes, sliders, radio buttons, and menus (Figure 3.2). Widget libraries are extremely versatile, however, they are difficult to use effectively. Since each widget represents a basic building block like a button, label, or icon, interfaces must be carefully designed by composing and placing these elements on a blank canvas to represent information and facilitate interaction. The graphical widget toolkits ensure that the individual controls are consistent and conform to user interface standards [DFAB97, p.414], however, the toolkit cannot enforce the same constraints among groups of controls.

Graphical frameworks, such as JFreeChart [JFr], JGraph [JGr], Piccolo [BMG00], and GEF [GEF], build upon basic interface components and present the developers with a more abstract set of tools for building interfaces. For example, instead of drawing squares and rectangles to create a chart, a designer, with JFreeChart, can create a bar chart by specifying X and Y-axis, bar series, colours, and titles. These toolkits are less versatile than the

underlying widget toolkits but are easier to use as they are targeted towards particular interface styles. For example, JGraph and GEF are used for displaying and editing data using a directed graph based approach, JFreeChart is designed for displaying and interacting with statistical data and Piccolo is intended for zoomable user interfaces [FB95].

Treemaps [JS91], hi-graphs [Har88], space trees [PGB02], cone trees [RMC91] and the table lens [RC94] have also been designed and built using a combination of basic widgets and higher level toolkits. We consider these to be specialized visualizations. By combining these specialized visualizations, visual attributes and advanced navigation techniques, researchers have been able to create representations of data of a million items or more [FP02]. Despite their capability, these advanced visualizations are not frequently seen in Commercial Off-The-Shelf (COTS) software.

A number of applications exist to help users visualize information, such as SHriMP [BSM02], Many Eyes [VWvH$^+$07] and the Business Intelligence and Reporting Tool (BIRT) [BIR]. Creating and customizing visualizations using these tools requires less programming than the frameworks and toolkits mentioned above, however, they do so at the cost of customizability. While the tools often support "end user programming" through scripts, preference dialogs and wizards, they cannot easily be customized for particular user tasks. They also suffer from "feature bloat", especially if users only need a small subset of their available features.

Figure 3.3 shows the spectrum from widgets to specialized visualizations with respect to versatility. While basic UI widgets are the most versatile, they are the most difficult to use for creating advanced visualizations. If the data to be visualized maps directly to a specialized visualization tool, such as a cone tree or treemap, than these tools can be quite easy to use. If on the other hand, the data does not fit the models, or the information rendered by these visualizations does not facilitate the user's needs, then these tools can become quite cumbersome and ineffective.

There are a number of criteria to consider when using tools to design and customize visualizations. To understand and categorize the types of visualizations and the constraints

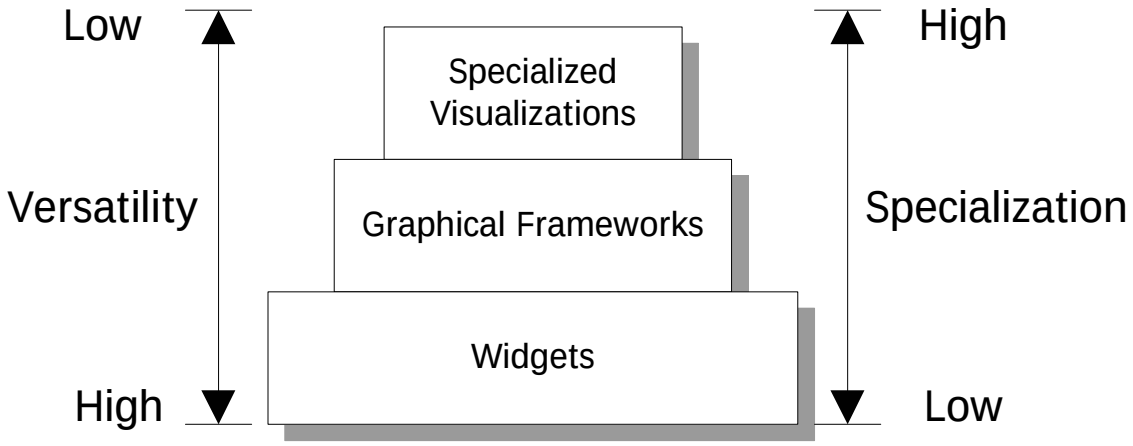

Figure 3.3: Relationship Between Versatility and Specialization

they impose, we examine the existing classification systems for information visualization.

3.3 Information Visualization Taxonomies

Interaction designers who design interactive visualizations of information can be broadly grouped into two categories: 1) those who establish techniques for representing information (creators), and 2) those who customize the techniques to create specific views (customizers). While this creator / customizer relationship is not unique to interaction design, the skill level differs across these two groups. Creators of new visualization techniques, such as the Table Lens, Treemap, Cone Tree, Space Tree, or graph layout algorithms, are often computer scientists specializing in the area of information visualization. On-the-other-hand, those who customize these techniques for their own data and use cases, such as financial analysts, medical researchers, and computer programmers, are often skilled in their domain but do not necessary have a deep knowledge of information visualization.

A visualization taxonomy gives researchers a common framework, as well as a catalog of results for others to reference. Bertin categorizes visualizations by the type of data and provided advice to customizers based on these data types [Ber81]. For example, Bertin suggests that colour value and size should be used for ordered attributes, but only size

should be used for quantitative data. Shneiderman [Shn96] outlined how visualizations could be categorized by their dimensionality (one-, two-, three-dimensional data, temporal and multi-dimensional data, and tree and graph data) and tasks (overview, zoom, filter, details-on-demand, relate, history and extracts). Tweedie [Twe97] classified a number of visualization techniques by the data they represent (data values or data structure, constructed data or converted data), the interactivity they support as well as their input/output relations across time. Card and Mackinlay [CM97] take a similar approach to Tweedie and classified the variables used in a visualization technique in terms of their data type (ordinal, quantitative, spatial, etc.), recording function, mark type, retinal properties (colour, size), position in space / time, among others. Pfitzner's considered five design aspects in his information visualization taxonomy [PHP03]: data, task, interactivity, skill level and context. While this is one of the most comprehensive taxonomies we have seen, Pfitzner did not categorize any visualization techniques using his taxonomy. Finally Tory and Möller [TM04] extended the work done by Shneiderman by splitting visualization techniques into two categories, visualizations that are suited for continuous vs. discrete data, an important distinction for our work.

Others, such as Price [PBS93], considered how visualization can be categorized within a specific domain. Price, for example, looked at software visualization. Software visualization focuses on how the design artifacts related to software and its development process can be presented graphically [Die07, p. 3].

Chi presented the date-state reference [Chi00], a catalog of 36 visualization techniques. For each visualization technique, Chi outlined the abstractions and transformations that could be applied. By presenting the visualizations in this manner, a customizer can quickly see the possible operations for a given technique. Of all the taxonomies, Chi's is the most closely related to our work as Chi begins to explore the way visualizations can be customized.

Many of the proposed taxonomies are aimed at designers who create new views by providing them with a framework and vocabulary that organizes the design space. With larger

displays and powerful computers available to many end-users, more software engineers are simply *using* information visualization as a tool and few frameworks provide a catalog (with accompanied tool support) to assist engineers when choosing between alternatives and exploring the customizable features of each.

3.4 Customizations of Visualizations

As discussed in Section 2.1, to study how visualizations are customized we take inspiration from Michaud and ultimately Wasserman [Mic03, Was90]. Michaud identified three ways in which software systems can be customized: 1) data, 2) presentation and 3) behaviour. We have refined these customization terms slightly from how Michaud used them, however, much of the context remains.

3.4.1 Data Customization

Choosing which data to present is central to any effective visualization [CMS99]. The customization of data within the context of information visualization refers to which data are presented and how the data is structured. The customization of data are commonly known as data preparation. Three data preparation operations can be applied to a data-set:

1. **Filtering of information**: Filtering removes information so it is not displayed in the view. Information can be removed because it is not relevant to the task at hand, because it is omnipresent, that is, it is everywhere and showing it is not needed, or because new information has been derived and the original data are no longer needed.

2. **Moving and renaming existing information**: Moving and renaming information is the process of changing the structure of the data without adding or removing anything.

3. **Deriving new information**: Deriving new information is the most complex task, but when done properly, often leads to the most interesting results. Grouping or combin-

ing elements, calculating new attributes or inferring relationships are all examples of how new information can be computed.

While these three data processing operations are commonly performed, it is the combination of these operations that are required to create truly effective visualizations.

3.4.2 Presentation Customization

In any visualization, the image is what immediately captures the attention of the user and it is the image that is intended to help the user with their tasks. Presentation customization refers to how artifacts are rendered within a view and is known as visual mapping [CMS99]. This includes choice of colour schemes, icons and visual cues, decorators, two or three dimensional rendering, and so forth. The choice of how visual attributes are customized is only as limiting as the designer's imagination. Enumerating all the possibilities is not feasible. View transformations, where particular elements are magnified or brought into focus, are also a form of presentation customization. The result of presentation customization is a customized visual rendering of the processed data.

Presentation customization can also be used to configure animations, another powerful mechanism enabled by computer supported information visualizations. Animations can be used to highlight elements of the view, bring details into focus, or help a user maintain context while switching perspectives.

Although enumerating or modeling all the customizable attributes of a visualization technique is not feasible, choosing a subset and standardizing these visual attributes is an important usability concern. By standardizing the ways in which the visual attributes can be customized, users can begin to interpret the interface through recognition rather than recall [Nor98].

An effective framework allows customizers and designers to extend a system's presentation in any manner, often through changes to the source code, but it also supports the formalization and sharing of common customizable components.

3.4.3 Control / Behaviour Customization

The biggest distinction between a diagram (one presented on paper for example) and an information visualization, is the interactions enabled by the machine. Like presentation, the number and types of actions that can be applied within an interactive visualization are infinite and are dependent on the type of visualization chosen. Node link diagrams, for example, often support layout actions, and tables may support the sorting and reordering of rows and columns. The control / behaviour customization refers to the customizer's ability to specify the actions supported by the view and configure the appropriate attributes. This includes how elements respond to selection, which elements should receive focus, and when and how multiple interactive views can be combined.

3.4.4 Current Approaches to Visualization Customization

Separating data from how it is presented is considered good engineering practice. A number of researchers have approached this problem through different model view controller implementations. While this architecture is effective, there is no standard approach to customizing visualizations. Graph query languages, shell scripts and visual queries have all been used.

Graph query languages such as GraphLog [CMR92], Grok [Hol99], and GUPRO [LSW01] can be used to manipulate graph structures. While a seemingly reasonable approach for customizing relational data, these languages are not designed for view customization as they have their roots in predicate logic or graph algebra. In all these cases, tool support and seamless integration between the language and view is fairly rudimentary. This means that view customization and data rendering are decoupled and processed in batch mode. Because of the lack of tool support and limited integration, these languages can be difficult to use and do not map easily to the task of customizing views.

Tools such as Snap Together Visualization [NS00, NCS02], SHriMP [BSM02] and Rigi [TWSM94] provide scripting languages to customize the views. These languages can

be used to filter the types of data displayed and modify the display properties. Coupling the scripting language with the visualization tool helps the interaction designer, but like the languages mentioned above, they can be difficult to learn and do not necessarily improve the efficiency of view creation. Mackinlay [Mac86] used sentences of graphical languages to support automated view creation of databases. For example, designers could use the phrase *"Present the Price and Mileage relations. The details about the set of Cars can be omitted"* to represent information about price and mileage in a database of cars for-sale. This approach, like many query languages, assumes that the individual posing the queries has a strong understanding of how the data are structured.

G^{See} [Fav01], Bloom [Rei01], Chive [CE04] and the work by Consens *et al.* [CMR92] present software visualization tools that have domain specific languages designed for describing views. Unlike the languages described above (Grok, GUPRO, etc.), these languages integrate the customization language within the visualization toolkit so users can easily experiment with ideas and immediately see the result. GUESS [Ada06], a language and interface for graph exploration, is the most recent tool in this space. GUESS combines an interpreted language, called Gython, with a graphical front end combining analysis and visualization into one package. The GUESS authors refer to this as Exploratory Data Analysis (EDA) [Tuk77] for graphs.

Exploratory data analysis through Visual Query Languages (VQL) and Visual Query Interfaces (VQI) provide a more intuitive interface for data analysts to pose questions [MK93]. Favre suggested using the Hy+ [MS95] query language to compose graphical queries with the G^{See} system, while Reiss proposed the MURAL [Rei02] VQL. Using visual query languages to create views for software systems is still in its infancy and the existing tools provide little support to help an interaction designer create the most effective view for their structured data-set.

3.4.5 Discussion

While Michaud has shown that software customization can be achieved and properly described by separating the customization of data, presentation and control / behaviour, very few visualizations are designed is such a formal way. In fact, many interactive visualization are designed in an *ad hoc* manner, as one-off solutions to a specific problem without documenting the design decisions that influenced the view. This practice makes it very difficult for future researchers and tool designers to recreate the success that the original view designer achieved. This also means that common patterns uncovered by view designers are not adequately made explicit for replication purposes. To provide insight into how tools are leveraged to create and customize information visualizations, we next examine several views that have been designed over the past decade.

CHAPTER 4

A Study of how Visualizations are Created

VISUALIZATIONS, such as those created for software systems or other large bodies of knowledge, are most effective when they have been customized for individual users and their tasks [MMC02, Sto98]. Information visualizations can be designed and customized by people ranging from end-users with very little technical expertise, to advanced computer users.

Effective visualizations are ideally created by domain experts, a group of people skilled in a particular domain but whose technical expertise can vary greatly from one to another. While some domain experts may be experienced computer users who are comfortable programming a custom visualization from scratch, others are much less technical. To support the design and customization of visualizations by domain experts with varying degrees of technical skills, a variety of different approaches to view creation have been proposed.

Visualizations can be created using end-user applications. These applications ease the process of designing visualizations through wizards and dialog boxes. Often limited in

scope, these applications are easy to use and integrate well with existing data-sets such as spreadsheets and common databases. While these tools can be easy to use, they are often bound to one specific application.

For more tailored applications, such as a software development environment, visualizations are much easier to use if they are integrated with the rest of the system. Domain specific visualizations can provide the infrastructure to achieve this level of integration, but a designer with some technical expertise is often required. The designer in this case is responsible for customizing a visualization toolkit and integrating it with existing technologies. Once the integration is completed, the views can be used and shared among end-users.

To facilitate task-specific visualizations, a custom view may have to be built from scratch. Views built from scratch are designed by software engineers working closely with domain experts. Unlike a visualization that has been customized for an entire domain, such as a generic visualization for computer networks, a custom coded visualization provides a more focused view. This is due to each artifact, its visual attributes and associated behaviour being carefully designed to assist users with a particular task.

In all cases of 1) end-user applications, 2) domain specific visualizations and 3) task specific visualizations, three aspects must be considered: 1) data customization, 2) presentation customization and 3) control / behaviour customization. In this chapter we study how these three aspects to view creation are actually configured for a series of six visualization tools and applications. This exercise culminates in a set of requirements that can be used for designing a model based environment for designing and customizing visualizations.

4.1 Case Study Overview

To understand the process of designing and customizing views better, we study six different approaches to view creation. Two general purpose visualization applications, three domain specific visualizations and a custom task specific view are analyzed. The two gen-

Methods of Customization					
Customizable Aspects			UI Supported Customization	DSL Supported Customization	Source Code Modification
	Data Customization	**Information Filtering**	Jambalaya / Creole / M.E.	PBS	Creole/ PDE Viz
		Renaming / Moving Information	BIRT	BIRT	Creole
		Deducing New Information	Creole	PBS / BIRT	Creole PDE Viz
	Presentation Customization		Jambalaya/ Creole / BIRT	PBS	PDE Viz
	Behaviour / Control Customization		Jambalaya	BIRT	PDE Viz

Table 4.1: Overview of visualization tools and their customization techniques

eral purpose applications studied are Many Eyes [VWvH+07], a public website used to create and share views, and the Business Intelligent and Reporting Tool (BIRT) [BIR], a report designer for relational data. The three domain specific visualizations examined are Jambalaya [SMS+01], Creole [LMSW03], and the Portable Bookshelf (PBS) [FHK+97]. Jambalaya is a domain specific visualization toolkit for knowledge management integrated with Protégé [NFM], while Creole and PBS are visualization toolkits for software visualization. Creole is integrated with Eclipse [ECL] while PBS is a suite of tools geared towards software design reconstruction and integrated with several command line Unix utilities. Finally, the custom view studied is a visualization tailored to help software engineers understand the dependencies between Eclipse Plug-ins (PDE Viz). This view was designed from the ground up to meet the needs of sophisticated users and integrates seamlessly with the Eclipse Integrated Development Environment (IDE).

These six technologies were chosen because together they enumerate the possible combinations of the customizable aspects of software, and methods of customization (Table 4.1). Studying how visualizations are designed and customized is a non-trivial task as most designers neither adequately document how their views are created nor describe the design decisions they considered.

In recent years, researchers have focused on the importance of evaluating views, but without proper documentation of how the views are created, it makes it difficult (and sometimes impossible) for others to recreate the view. Tool designers looking within the research community for ideas on how to best visually represent their data, often have to try to understand how a view was constructed from screen-shots and the prose describing it. Even a simple statement like, *the nodes represent classes, the edges represent relationships between them and a spring layout algorithm is applied* is ambiguous. Which classes are represented, which relationships (inheritance, composition, calls, depends, uses, etc.) are included and which parameters are passed to the spring layout? These are important decisions, and if the view received a favorable evaluation, tool designers will need to know this information if they need to recreate the view.

To understand the process of designing and customizing visualizations better, and to derive the requirements for a visualization design environment, we follow this process:

1. **Analysis of the underlying data-model:** When available, the underlying data-model for each tool is captured and documented using UML.
2. **Review of the visualizations:** For each tool, the available visualization techniques and a description of the uses of each visualization technique are documented.
3. **Examination of the view creation:** A description of how views are customized from the underlying data-model is uncovered. This includes which visualization techniques are chosen and which data processing operations are performed.
4. **Examination of the presentation customization:** The techniques used to customize the visual aspects of the view are captured. This includes how colours, shapes and other visual attributes can be customized.
5. **Examination of the control / behaviour:** The actions chosen and the ways they are configured is documented. For example, in the case of graph visualizations, this may include the ability to select elements or re-execute layout algorithms.
6. **Review of the lessons learned:** A set of lessons learned is distilled for each tool.

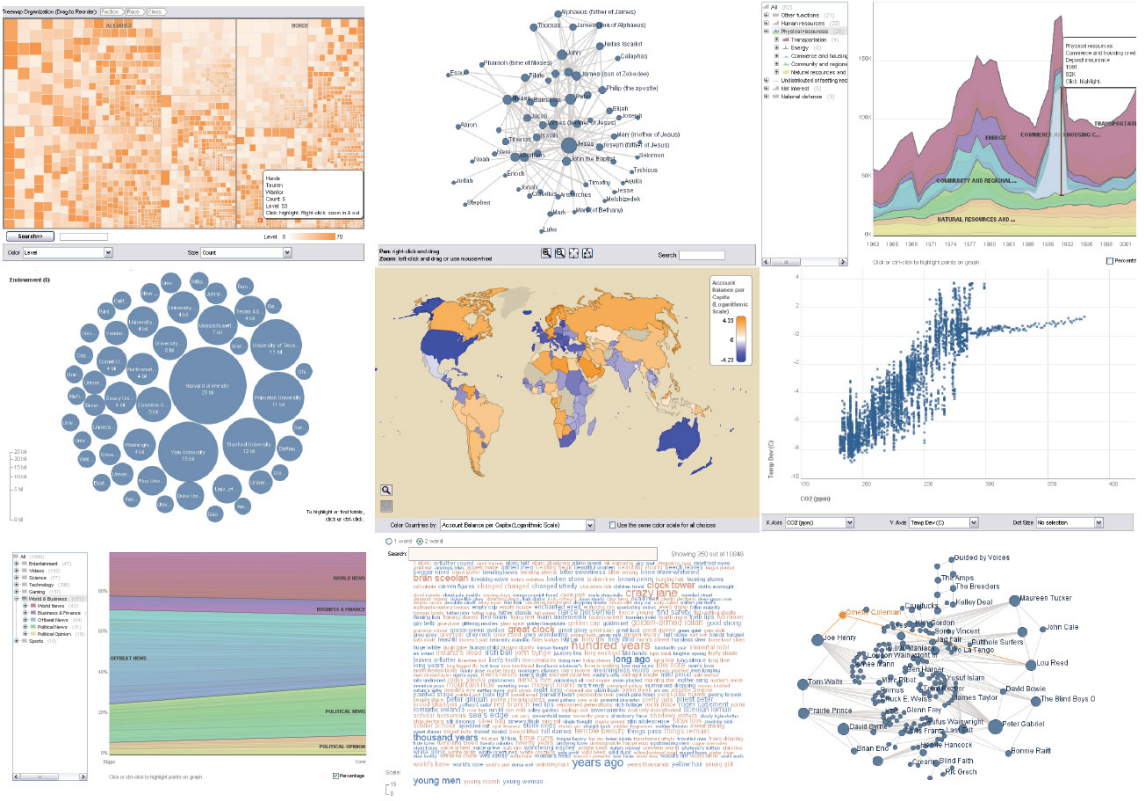

Figure 4.1: Nine Views from Many Eyes

4.2 Generic Visualization Tools

The first category of technologies studied are generic visualization applications. Many visualization applications are stand-alone software systems intended for end-users that support the creation and customization of information visualizations. Visualization applications are often configured to work with a number of existing data-sources, and through wizards, preference panels and domain specific languages, end-users can create custom views. While the applications may support a number of standard data formats, they rarely provide a means to incorporate the views into other systems. Two applications that meet this profile, Many Eyes and BIRT, are presented below.

4.2.1 Many Eyes

Many Eyes is a public web site that enables anyone to upload data, create interactive visualizations, and share their results [VWvH+07]. As the designers of Many Eyes describe: *"the goal of [Many Eyes] is to support collaboration around visualizations at a large scale by fostering a social style of data analysis in which visualization's not only serve as a discovery tool for individuals but also as a medium to spur discussions among users."* [VWvH+07] The site supports a number of visualization techniques including: charts (bar, line, stack and bubble), maps, tag clouds, treemaps and graphs (Figure 4.1).

4.2.1.1 Many Eyes Model

To make use of the visualizations provided by Many Eyes, comma delimited text file must first be uploaded to the site and a visualization technique must be selected. Different visualization techniques have different constraints. For example, to create a node-link view, the data must contain at least two columns with intersecting sets of data. Once uploaded, the user can select a column for the source and a column for the destination nodes. Many Eyes will then generate a view, apply a layout, and render it using a Java applet that supports pan and zoom operations.

4.2.1.2 Many Eyes Views

Many eyes contains 16 visualization types including maps (both world and individual countries), charts, node-link diagrams, treemaps and tag clouds. The authors of Many Eyes have not published formal models for any of the views, however, by reading the documentation and experimenting with the system, approximate models can be derived. Finally, the views cannot be linked in any way, so designing views with overview+detail, or details-on-demand is not possible.

In terms of the visualization reference model described in Section 3.1 [CMS99], Many Eyes provides no support for the data preparation stage.

4.2.1.3 Many Eyes View Creation

Views are created in Many Eyes by mapping elements to visual attributes. Once a data-set is uploaded, a view type (bar chart, line chart, map, graph, etc.) can be selected and then elements from the data-set can be mapped to visual artifacts. This simplified interface means that complex data customization operations cannot be performed. To perform data abstractions, or other data customization operations, the data must be processed prior to being uploaded.

4.2.1.4 Many Eyes Presentation Customization

While data can be mapped to visual elements, Many Eyes provides little support for configuring visual properties such as colour, icon, or label. By not supporting these customization options, Many Eyes has a relatively simple user interface.

4.2.1.5 Many Eyes Control / Behaviour Customization

Each view has a number of associated actions, but like the customization of the visual attributes, Many Eyes provides no facilities to customize them nor facilities for choosing the actions available for each view. For example, in the layout action on the node-link diagram, the user has no control over which layout is performed nor for choosing the control parameters.

4.2.1.6 Many Eyes: Lessons Learned

Using Many Eyes to create visualizations of data is remarkably easy, but ease of customization is achieved by limiting the available options. Many Eyes provides very little support for data exploration since all queries and data processing must be completed using a separate program. This separation removes any record of the data processing step. Starting from a single data-set, and performing a number of abstractions and calculations, a user can quickly end up with several data files without any history of how each was created. Also,

once a data-set is uploaded, it cannot be modified to evolve the view.

Many Eyes facilitates the construction and sharing of visualizations for the end-user through an easy to use interface and web based system. From a single data-set, an end-user can create both a bubble, bar and line chart, share them with others, and solicit feedback on the findings.

4.2.2 Business Intelligent and Reporting Tool

The second general purpose visualization application studied is the Business Intelligent and Reporting Tool (BIRT) [BIR]. BIRT is a reporting system for both desktop and web applications. Working from several data-sources, a variety of standard charts and table based reports can be designed and exported to a number of formats or published on the web. BIRT also has a "report designer" a tool that enables designers to construct new reports by dragging and dropping report elements from a pallet. Those users designing BIRT reports are usually advanced business users with a moderate degree of technical skills. Once the charts have been designed and exported, they are presented to an end-user for analysis. The end-user in this case is not required to have any technical expertise.

4.2.2.1 BIRT Model

BIRT supports a number of data-sources including flat files, structured XML, web service data and relational databases. By interfacing with these data-sources, designers can create reports without first exporting their data or maintaining multiple data files. Also, since the visualizations are designed directly on the underlying data, the views automatically update as the data evolves.

4.2.2.2 BIRT Views

BIRT reports have an assortment of standard statistical charts, such as, line, bar, pie, etc. While several charts can be placed in one report, the charts cannot be linked, however,

they do provide more interactivity than other business charting applications such as Excel. For example, elements in a BIRT chart can be associated with an action and configured through JavaScript, such that when an end-user selects something, a dialog or pop-up can be displayed.

4.2.2.3 BIRT View Creation

Views are created using a sophisticated report design tool intended for designers or end-users with a moderate degree of technical expertise. Once the charts have been selected, the data axes can be configured. BIRT also supports many advanced features allowing the designer to create summary reports directly from the data-source using a variety of abstractions and data filtering operations. The report design tool uses a combination of dialog boxes and SQL queries for customization purposes. SQL queries are used to derive information and compute new data columns, and the dialog boxes are used to select which columns to display. This approach enables a single data-source to be used in the construction of several views, and as the data-source is updated, the views dynamically evolve.

One notable shortcoming to BIRT's approach to view creation is that no preview of the view is available at design time. Consequently, once a chart is configured, the designer has to go through an additional step of exporting the view before they can see the results. This is likely due to the fact that heavy computing resources are needed to extract the data from the data-source and render the view.

4.2.2.4 BIRT Presentation Customization

Visual attributes can be selected and customized using a series of dialog boxes. Effects such as a 3-dimensional appearance or the choice of colours and fonts can all be customized. In some views, a designer can also apply effects based on a criteria such as "exploding" a slice in a pie chart if a certain condition is met.

4.2.2.5 BIRT Control / Behaviour Customization

Choosing actions and customizing the control is supported in BIRT through JavaScript. By selecting one of the preset configurable actions, which includes Mouse Click, Keyboard Click, and Mouse Over, a custom script can be executed. BIRT provides a number of standard scripts such as "navigate to URL" and "highlight element." The combination of a scriptable interface with predefined scripts, provides the designers with the tools to easily make use of common controls while still supporting the flexibility of advanced operations.

4.2.2.6 BIRT: Lessons Learned

BIRT provides a rich environment for designing and customizing chart based information visualizations. The drag and drop interface used to create the views is simple to use, while still supporting powerful data deriving operations through SQL queries.

Unlike Many Eyes, BIRT supports data processing so that many different views can be created from a single source. Moreover, when a data-source is updated, the views can be instantly regenerated.

BIRT's approach to customizing presentation and control is highly versatile. By combining a UI driven approach with script writing, highly customized reports can be created. BIRT also provides a number of predefined scripts so that both novice and experienced designers can make use of the most common operations without the need to write code.

4.3 Domain Specific Visualizations

In contrast to the generic visualization applications discussed above, which run as stand-alone tools for the purpose of visualizing any type of information, a domain specific visualization is a suite of visualization tools intended for a particular domain. Because domain specific visualizations target a specific set of end-users, these views can often be integrated with existing systems that are familiar to the user.

We study three domain specific visualization toolkits: 1) Jambalaya, 2) the Portable Bookshelf (PBS) and, 3) Creole. In each case, we uncovered how advanced users created and customized a specific view for end-users to consume.

4.3.1 Jambalaya

Jambalaya was created to assist with the comprehension of ontologies. Jambalaya is integrated directly within Protégé [NFM], a popular ontology editor, and can be further customized to produce a number of task-specific views.

Allen [All03], a researcher at the University of Victoria, further customized Jambalaya to create a tool called SHriMPBib. SHriMPBib was designed to help researchers track, rate, search and archive publications. The highly customized instance of Jambalaya provided researchers with a rich interface from which to enter the data about publications, browse existing entries, and visualize the information.

When SHriMPBib was put in production, 15 users registered and almost 600 documents were entered. The documents could be rated and marked with a research area. While the size of the system is small, the complexity of managing these documents was still a barrier for the users. Allen created several visualizations to help users manage their documents.

4.3.1.1 Jambalaya Model

Jambalaya uses an ontology as its core data model. An ontology is an explicit specification of a conceptualization [Gru95]. An ontology describes the elements that exist in a domain and the relationships between them. In the case of SHriMPBib, Allen defined an ontology for academic publications consisting of four main class hierarchies (Figure 4.2): 1) DocumentArea, 2) User, 3) Document, and 4) Rating. Each of these hierarchies has a number of classes and each class has a number of slots. For example, the document hierarchy has six concrete subclasses, (Book, ConferenceProceeding, Generic, JournalArticle, Article and Webpage), and these classes have slots such as Pages, Keywords, and Author. Slots can

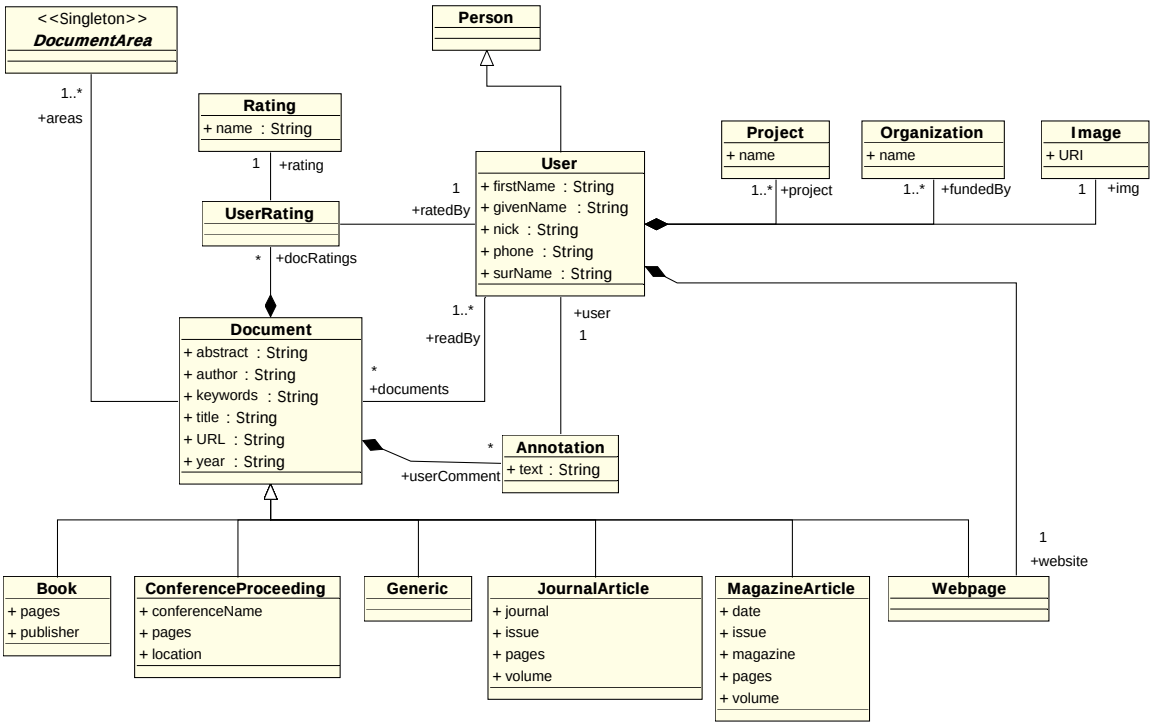

Figure 4.2: SHriMPBib Model

also connect classes, such as the connection between a user and their read documents. The DocumentArea class actually represents a taxonomy of document areas (such as software engineering, information visualization, and algorithms), but was omitted from the diagram for readability.

4.3.1.2 Jambalaya Views

Jambalaya's main visualization is a nested node-link view. By default, Jambalaya shows all nodes and edges that exist in a system. For SHriMPBib, Allen created several custom views to help users manage a collection of documents, three of which are discussed below. The first view is the "Member Document View". In this view, a node link diagram is displayed and each user is represented as a node "connected" to the documents they rated. By applying a spring layout to this graph, a visualization, such as that shown in Figure 4.3, is rendered. It is easy to see that for this view that three users (User 1, User 2 and User 3),

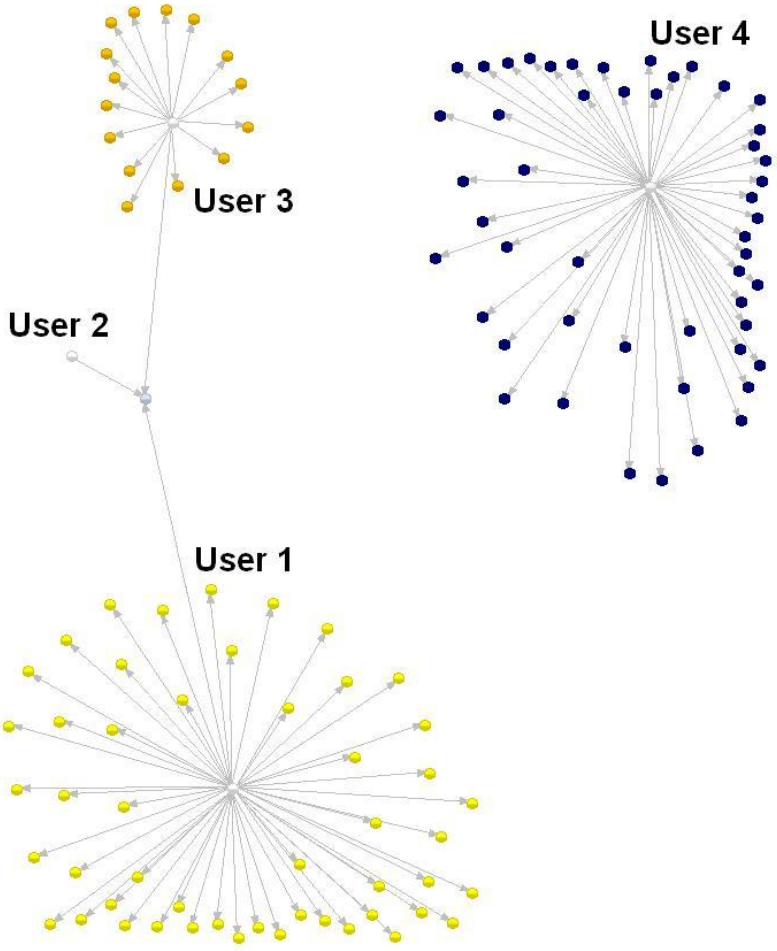

Figure 4.3: Member Document View

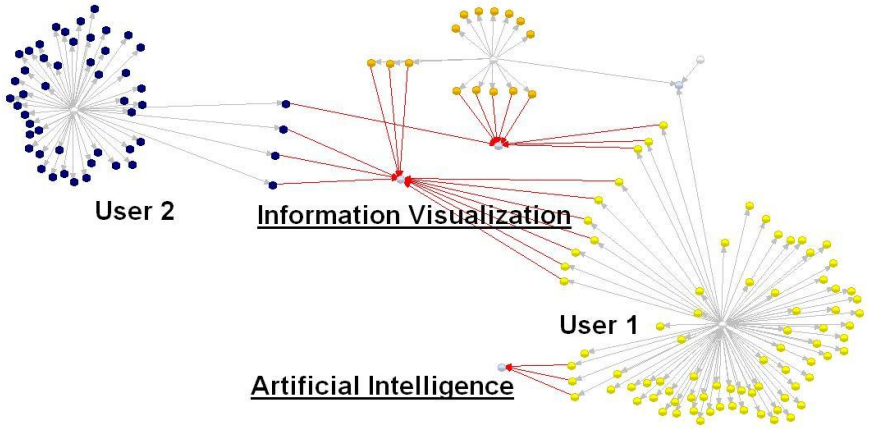

Figure 4.4: Members, Documents and Area of Interest

all reviewed the same paper.

The second view (Figure 4.4) extends the first view by adding the DocumentArea as a node to the graph, such that each document is connected to a "document area". The document is also connected to all the users who rated it. Using this view, it can be seen that User 1 and User 2 both read papers on Information Visualization, while Artificial Intelligence articles were only read by User 1.

The final view uses a Treemap [JS91] to show all the research areas (Figure 4.5). Each rectangle in the treemap contains a list of papers in that area, colour coded by the individual who added them. The boxes are sized according to the number of documents contained within it. This view highlights the areas of research that are dominant, and by looking at the map, the experts in each area can easily be identified.

4.3.1.3 Jambalaya View Creation

Creating views in Jambalaya, like the ones Allen created, requires a moderate degree of technical skill. Because Jambalaya is already configured to work with Protégé, no custom code is required to create the views. Instead, the designer customizes the views by enabling filters, deriving new relationships and applying specific layout algorithms through a set of preference panels. The first view is designed by opting to show only elements of type

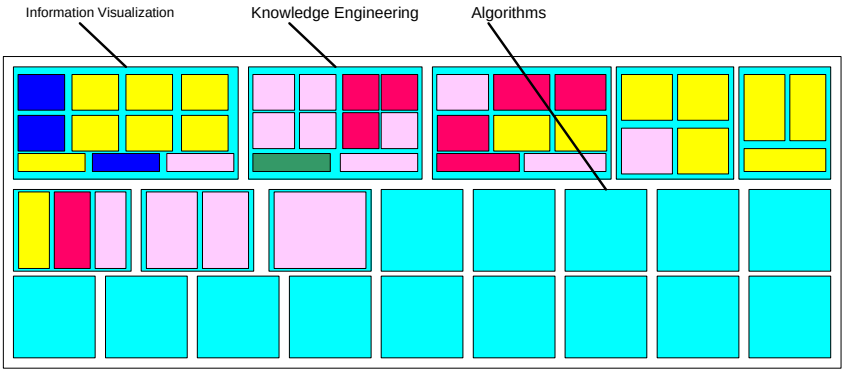

Figure 4.5: Treemap Showing Documents and Areas of Research Interest

"User" and "Document". For the document, the node is coloured based on the "Added By" field. Finally, all edges that connect users to their documents are shown. To create the second view, the first view is extended and all the elements of type "Area" are enabled. An edge is added from each document to all of its associated areas. Finally, the edges are coloured red and decorated with an arrow head to make them stand out.

A different approach is used to create the third view. In this case, three of the elements (Area, Document and User) are grouped hierarchically. A treemap [JS91] layout is then applied such that the size of the rectangle is dependent on the number of descendants. All other elements are filtered from the view.

In all three views, the choice of elements to show was performed using a selection dialog. The selection dialog provides an option to show and hide elements by type. While this was satisfactory in this case, the UI does not support complex filters.

4.3.1.4 Jambalaya Presentation Customization

Jambalaya supports the customization of visual attributes through wizards and UI dialog boxes. For SHriMPBib, Allen was able to choose a colour, border style, label position and optional image. For edges, the possible customizable visual attributes include colour and edge style such as solid, dotted or curved.

As well as configuration based on type, nodes can be coloured based on their attributes.

For example, a colour value can be chosen based on number of descendants or the value of a given attribute. In the case of SHriMPBib, the documents are coloured to help users see who added them. If two documents are added by the same user, they are coloured the same.

4.3.1.5 Jambalaya Control / Behaviour Customization

Jambalaya has a number of default actions such as the execution of a layout algorithm, selection triggers, zooming and panning. To change the default behaviour, or to disable these actions, the designer is required to write custom code. The only exception to this are the "Quick-Views", a tool that allows designers to create a custom view for end-users. Using the quick-view editor, the designer can decide which nodes and edges to show, and which options the initial layout should include.

4.3.1.6 Jambalaya: Lessons Learned

Creating a visual representation of an instance of an ontology with Jambalaya, like those Allen created, requires a number of subtle features. Allen's views made use of: 1) filtering (nodes and edges), 2) configuration of visual attributes (namely node and edge colour), 3) re-organization of the hierarchy, and 4) the configuration of layout algorithms.

Jambalaya's UI approach to view customization made it very easy for Allen to customize the view for the needs of her users, however, there are a few limitations to this technique. First, not all data deriving operations are possible. For example, if a node is hidden, all the descendants of that node are also hidden (i.e. the nodes cannot be re-parented). Second, views can only be specified at a high level. For example, filters cannot be chosen based on the value of an attribute such as hiding all documents whose name starts with *Artificial*. Finally, since all the customizations were designed through a UI, the actual design decisions were never captured making them impossible to fully reproduce. In the case of SHriMPBib, the configuration of the spring layout algorithm was never documented, although it would appear this configuration was an important aspect of the tool's success.

4.3.2 Portable Bookshelf (PBS)

The second domain specific visualization studied is the Portable Bookshelf (PBS). PBS is a tool to help software engineers understand and re-document the design of a software system. PBS has three components: 1) The C Fact Extractor (CFX), 2) a domain specific language for manipulating graphs (Grok [Hol]), and 3) a graph rendering tool (LSEdit [Pen93]).

Many software systems do not have a documented system architecture [GS93] and to address this issue, Bowman *et al.* made use of the PBS toolkit to design an architectural reconstruction and re-documentation process and applied this technique to the Linux kernel [BHB99]. The architectural reconstruction and re-documentation process follows the general reverse engineering process outlined by Chikofsky and Cross [CI90]. In particular, low-level artifacts such as files, functions, variables and the relationships between them are extracted using the CFX fact extractor [FHK+97, Hol97] to re-construct the "As-Built" [MNS01] system architecture. These artifacts are then abstracted to produce high-level information, such as, the dependencies between modules and subsystems using Grok [Hol]. Grok is a domain specific language for manipulating graphs based on Tarski algebra. Finally, the results are visualized using a rudimentary tool called LSEdit [Pen93].

4.3.2.1 PBS Model

PBS uses a relational model based on triples. The triples can be of two forms: 1) source, destination and relation name, or 2) element, attribute name and attribute value. The first type of triple defines edges in a graph while the second type defines attributes. In addition to these two types of triples, PBS supports named entities. Since the entities, relationships and attributes are named, a model (or schema) can be created.

The artifacts extracted by PBS's fact extractor, CFX, are "low-level", similar to those extracted by Rigi [MK88], Dali [KC98], CPPX [MDH01] and Ciao [CFKW95]. Low-level artifacts are code artifacts close to the statement level, such as, function `main` *calls*

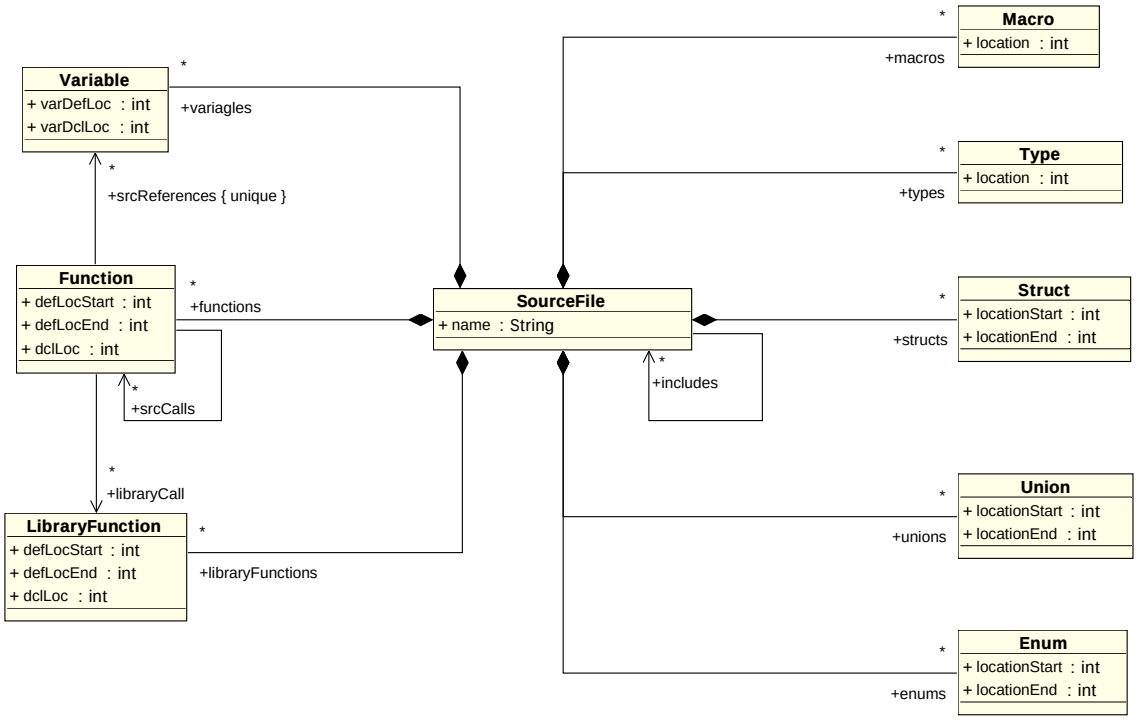

Figure 4.6: Model to Which Artifacts Extracted from CFX Conform

function `print` and function `print` *uses* variable `device`. In the case of the Linux Architecture example, we could not find an explicit schema to which the facts extracted by CFX conform, however, Bowman *et al.* present the relevant concepts in their paper *"Connecting Architecture Reconstruction Frameworks"* [BGH00]. In this paper, Bowman *et al.* presents nine elements: Variable, Function, Library Function, Macro, Type, Struct, Union, Enum and Source File. There are several relationships in this model including Variable Definition, Variable Declaration, Function Definition, Function Declaration, Reference, and Call. Figure 4.6 presents the model as a k class diagram.

4.3.2.2 PBS Views

PBS supports a nested graph view similar to the one used in Jambalaya. In the case of the Linux kernel visualization, Bowman used the view to show a high-level description of the major subsystems and their dependencies (Figure 4.7 (A)). From this view, it can be

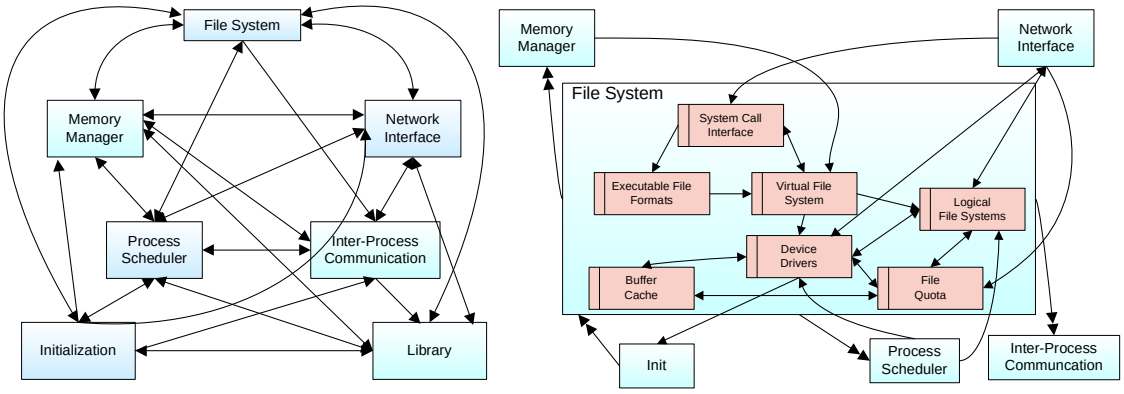

Figure 4.7: Architecture of the Linux Kernel

seen that the Linux kernel has seven subsystems and 37 inter-subsystem dependencies. The nested view also supports a number of navigation actions, including "dive-into" in order to display elements within a subsystem (see Figure 4.7 (B)).

4.3.2.3 PBS View Creation

Creating views to help re-document software design from low level artifacts is a multi-step process. In the case of the architectural reconstruction of the Linux kernel, Bowman first mapped each file (or module) to a subsystem. The subsystems were then grouped hierarchically. Bowman performed this mapping manually, creating data files that contained the desired mappings. Once completed, Bowman merged his mapping files with the extracted source artifacts.

Since subsystems have no direct relationship between one another, composite relationships, or lifted edges, must be computed [FKO98]. In this case, a relationship between two subsystems (A and B) is added if an element in the set of descendants of subsystem A has a relationship (*calls* or *references*) to an element in the set of descendants of subsystem B. Bowman used Grok [Hol], a DSL intended for manipulating directed graphs, to compute these abstractions.

4.3.2.4 PBS Presentation Customization

PBS supports two methods for customizing visual attributes. Initially, each element can be annotated with an attribute describing its shape, label and colour. For the Linux architecture visualization, the subsystems are represented using blue rectangles and modules are rendered as red rectangles with double borders on the left. The initial visual attributes are configured using a Grok script. In addition to Grok, PBS supports dialogs to further configure the visual attributes. By combining both approaches, an initial assignment of colours and shapes can be performed by an experienced designer, while still allowing the end-user to change these attributes as their tasks evolve.

4.3.2.5 PBS Control / Behaviour Customization

PBS neither supports the customization nor the configuration of control / behaviour. There are several actions available, such as navigation, panning, zooming and a history panel, but designers are not able to modify the behaviour nor can they disable features.

4.3.2.6 PBS: Lessons Learned

Recovering software architecture diagrams or other design artifacts from source code and presenting the results visually is common practice to help software engineers better understand the structure of a large software system [FHK+97, MK88, HH02, TH96, KC98]. Construction of these views requires a number of visualization customization methods, including: 1) the union of several data-sources, 2) configuration of visual elements (edge colour and node shapes), 3) edge lifting or abstraction and 4) node and edge filtering.

By separating the configuration of the view from a description of the view itself, and specifying the design through a DSL, a detailed description of how the view is designed is available. This approach makes it much easier for future researchers and tool-smiths to recreate the visualization for a different, yet similarly structured data-set (such as the architecture of the Mozilla web browser [GL00]).

To further customize a visualization, the designer must first modify the DSL and then use the modified program to re-generate the views. The main drawback with this approach is the disconnect between the view and tools used to customize the view. This lack of integration may limit the designers efficiency. Finally, it is unclear if Grok, a DSL based on Tarski algebra, is the most intuitive notation for designing visualizations of software systems.

4.3.3 Creole: Software Visualization

The final domain specific visualization studied is Creole, a plug-in for the Eclipse integrated development environment that supports a number of different software visualization options.

While several tools exist to help developers understand the static structure of their software, Wu [Wu03] noted that version control information is hard to understand because of ineffective visual representations. To improve the understanding of historical software artifacts, Wu developed Xia, a customized version of Creole for version control information.

When Wu began her work, Creole did not support version control data, so the design and customization of Xia's views was done through a mixture of programming and user interface preference panels. Wu wrote custom code to extract CVS information, filter uninteresting elements and construct a more suitable data representation. The preference panels were used to customize the visual attributes and fine tune the data filtering parameters.

4.3.3.1 Creole Model

Creole uses a typed graph model with edges labeled as "hierarchical" to support nesting. As well, Creole can interpret a number of standard graph exchange formats including TA [Hol97], GXL [GXL, HSW00] and RSF [MK88]. By default, Creole extracts a number of source code artifacts from a Java program.

To visualize information about software history, Wu used the Eclipse CVS API to pop-

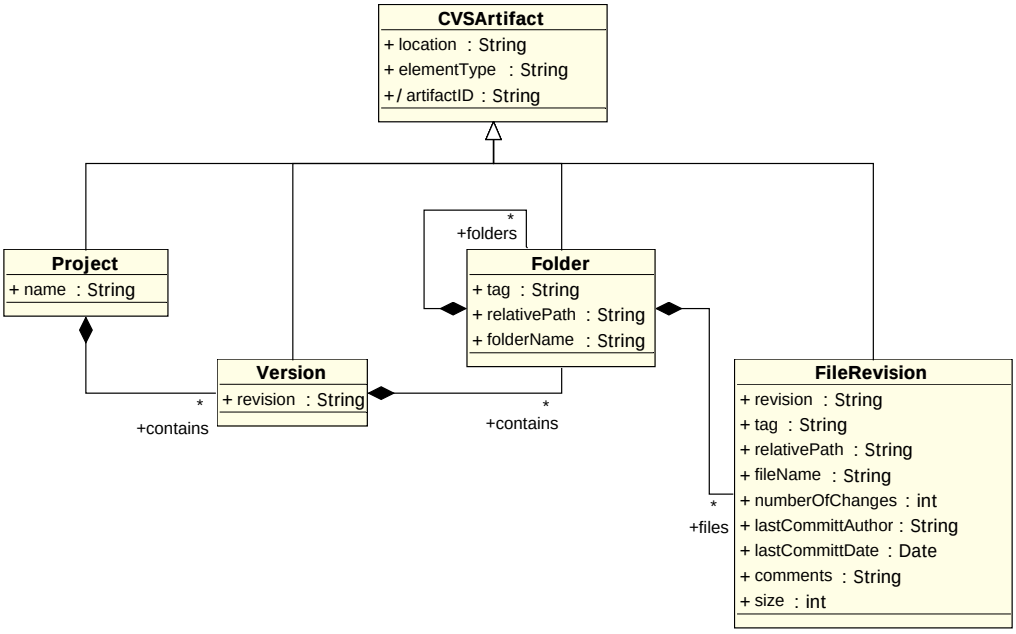

Figure 4.8: Xia Data Model

ulate a custom data-model. Wu's model has four types of elements: **Project**, **Version**, **Folder** and **FileRevision**. Each type has an artifactID to uniquely identify the element, a networkLocation, and a field representing the elementType. The **FileRevision** type also contains a number of attributes including name, size, numberOfChanges, commitAuthor, commitDate, tag and comments. The edges are limited to containment relationships, such that file revisions are "contained" within their parent folder, folders are contained within versions (or other folders) and versions are contained within projects. Figure 4.8 shows a UML class diagram for this model.

4.3.3.2 Creole Views

Like many other toolkits, Creole uses a nested graph view. Both the nodes and the edges can be configured and a number of graph layout algorithms can be applied. In the case of Xia, only nested treemap views are used. Projects, versions, folders and files are nested within one another. The project is shown at the root, with versions, folders and files below. The hierarchy is *expandable*, meaning that a user can just view the project, or expand all

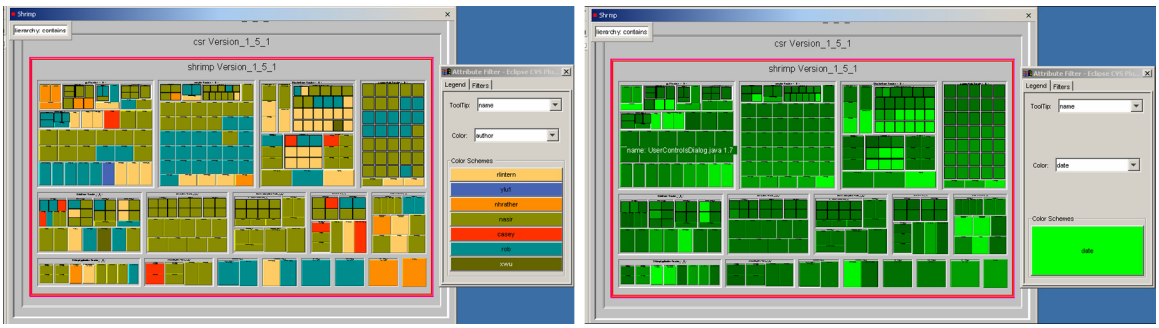

Figure 4.9: Xia CVS Visualization

the descendants to see the files that make up the different versions.

The treemap algorithm is configured in a number of different ways depending on what is intended from the view. For example, within a folder, all the subfolders and files can be positioned based on their size (lines of code) or number of revisions.

Colour is also used frequently within Xia to distinguish both nominal and ordinal data. Different colours are assigned to different nominal attribute values, and colour intensity is used to distinguish ordinal data.

Figure 4.9-left shows how Xia displays an overview of all files in the system colour coded by the developer who last worked on them. Figure 4.9-right presents the same system, but in this case colour intensity was used to represent the last modified date.

4.3.3.3 Creole View Creation

Creole supports many software visualizations including call traces and system decomposition diagrams. Using a series of dialog boxes, a designer can tailor these views or create an entirely new one. Since Creole did not support CVS information when Wu began her work, she first developed a custom data extraction tool to retrieve the necessary data from CVS, and then used a series of dialog boxes to finalize the presentation.

4.3.3.4 Creole Presentation Customization

Creole supports the configuration of visual attributes through a series of UI dialog boxes. The node properties that can be configured are: size, shape, colour, border and label. Also, a number of visual attributes can be configured for the edges, such as colour, line style, width and label.

In the first Xia view, colour is used to represent the developer who committed the most recent change. In the second view, colour intensity is used to indicate the date of the most recent change. Labels are placed above the nodes, and the default rectangle shape was chosen.

4.3.3.5 Creole Control / Behaviour Customization

Creole does not support an extensive set of customizable actions. Current versions of Creole allow designers to specify a set of "Quick View" actions, like the ones specified in Jambalaya, but these were not available when Wu designed Xia.

4.3.3.6 Creole: Lessons Learned

Creole can be customized by mapping Java artifacts to visual elements. In the case of Xia, Wu extended the model by including a custom data extractor to integrate file version history into the application.

Once the version history was extracted, the visual artifacts were customized based on attribute values such as date and author. For nominal data (author) a simple colour mapping was performed. For ordinal data (date committed) a colour intensity value was assigned.

Because Creole does not support an adaptable data model, an intermediate data representation was designed to bridge between Eclipse and Creole. This extra step meant that the designer, Wu in this case, had to understand the internal details of both Creole and Eclipse.

Like Jambalaya, the UI based customization used in Creole provides an easy way to

modify the view. Since the view was customized through both source code modifications and option dialogs, the details of the customization were not isolated, but rather distributed across a number of files in multiple formats.

4.4 Task Specific Visualizations

Many visualizations are designed and implemented to assist end-users with a specific task. These views are highly tailored and are developed as custom solutions to specific problems. Since it is difficult to understand the subtleties of how a specialized tool is created and customized without taking part in the tool's design, we studied a visualization that we engineered from a set of requirements gathered from experts in the field of Eclipse plug-in development.

The view we constructed is a plug-in dependency view for Eclipse [Bul]. The view is intended to assist software engineers who develop plug-ins for Eclipse, as they often have trouble understanding the dependency stack for a given plug-in. As described by a Eclipse committer on a feature request:

> *"When validating the set of runtime plug-ins, it's almost impossible to understand why a given plug-in is disabled. Doing this requires hunting around in the tree of disabled plug-ins*. [Hud]"

The plug-in dependency view (shown in Figure 4.10) was implemented over the course of a summer and the requirements taken from a number of experts. The view was well received and initial feedback was quite positive. To study the customization of this view, we looked at what was required to configure data, presentation and control / behaviour. The view was designed using Eclipse and a high-level programming language (Java).

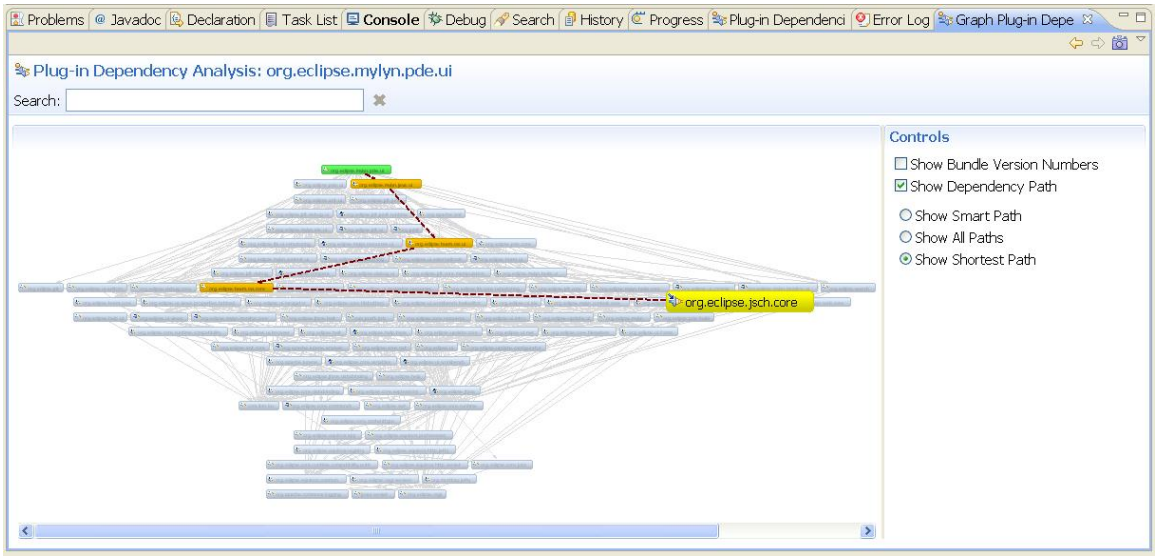

Figure 4.10: Plug-in Dependency View

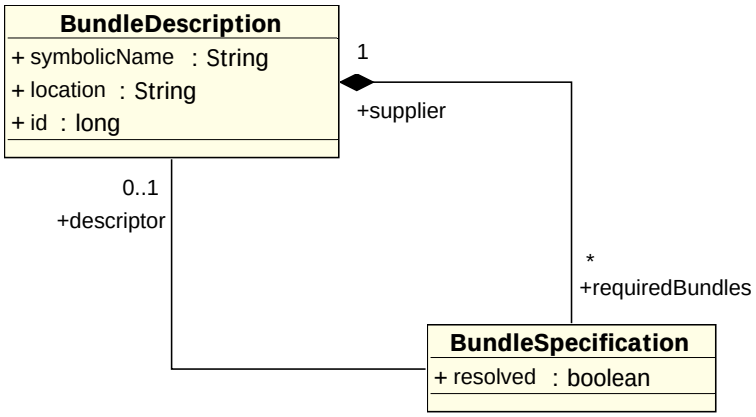

Figure 4.11: Plug-in Dependency Model

4.4.1 The Plug-in Model

The model used by the Plug-in Development Environment (PDE) to represent dependencies is straight-forward. Each plug-in is represented by a **BundleDescription** and all the dependencies are represented by a **BundleSpecification**[1]. Each specification has 0 or 1 descriptors (depending whether or not the specification can be resolved). The model is shown in Figure 4.11. In Eclipse, the model is represented as an object model described by a series of Java classes.

4.4.2 The Plug-in View

The PDE visualization is a single view used to display the dependencies between plug-ins. A number of navigation actions are available, as well as a number of actions that highlight plug-ins for different analysis tasks. The default view (Figure 4.10) shows the selected plug-in and all dependent plug-ins. Directed edges are drawn to show the relationships and a hierarchial layout algorithm is applied.

The view is integrated within Eclipse using the Eclipse plug-in mechanism. By integrating the view in this manner, end-users can view their dependencies without the burden of launching a separate tool.

4.4.3 Customizing the View

The creation of the plug-in dependency view is done by tightly integrating a graph rendering engine [BBS04] with the Eclipse plug-in development environment. This approach provides up-to-date information about the plug-ins and their relationships. To create the view, the plug-in registry is first traversed, starting with the selected element. For each dependency, the specification was resolved and a list of indirectly dependent plug-ins is computed. This idea is applied recursively until there are no more plug-ins left to be resolved.

[1] The terms bundle and plug-in are used interchangeably, although bundles are the terms used in the OSGi specification [All08], the term plug-in is still accepted for historic reasons and the term more widely used within Eclipse.

```
public Color getBackgroundColour(Object entity) {
  if ( entity == this.rootNode )
    return LIGHT_GREEN;
  else if ( entity == this.selected )
    return LIGHT_BLUE;
  else if ( interestingDependencies.contains(entity) )
    return ORANGE;
  else
    return LIGHT_GREY;
}
```

List of Listings 4.1: Colour adapter for graph node

In the view, a node is used to represent each plug-in. The name of the plug-in is used as the label, and a decorated icon is added to the model to indicate a number of properties about the plug-in. A directed edge was drawn between the source plug-in and all the plug-ins directly referenced.

The graph based rendering engine is an MVC framework that makes use of model adapters, a design pattern [GHJV95] that separates the model from its visual representation and places all the code used to calculate the view model in one place.

4.4.4 Configuring the Presentation

The customization of the visual attributes, such as node and edge colours, labels, font, and icons is done programmatically and supported through an adapter pattern. This pattern supports the addition of decorators (modified icons), labels, colours and so on, in a model independent manner. For example, setting the node colours to blue requires no knowledge of the graphing framework or rendering engine, instead, a single interface with all the configuration options is implemented. Listing 4.1 shows the method used to choose the colour based on selection.

Customizing the visual attributes in this manner places all the UI configuration options in one place and provides fine grained control over the customizable options, however, a complete rebuild of the application is required for the changes to take effect.

4.4.5 Configuring the Control / Behaviour

The plug-in dependency viewer, of all the tools studied, had the most configurable actions. Each of the actions were configured using a high-level programming language.

The plug-in dependency view supports a number of actions that are triggered based on selection, such as, highlighting the shortest path between two elements and filtering nodes through a quick filter option. It was hypothesized that by providing actions such as these, users could more easily understand the relationships between the plug-ins. The shortest path action was designed by incorporating Dijkstra's shortest path algorithm [Dij59] into the `onSelection()` method.

4.4.6 Custom Visualization: Lessons Learned

The primary difference between the creation of a custom view and the creation of the other views is the predominant use of a programming language to configure the visualization. A high level programming language provides more flexibility than any of the previous approaches, but more tightly couples the implementation of the view to its specification.

The use of an MVC framework and model adapters helps isolate the view customization from the underlying rendering engine. Model adapters are used both for the customization of the data and the presentation. Concerning the data, an abstraction is performed to bypass the **BundleSpecification** and directly link all **BundleDescriptions** that are dependent on one another. Regarding the presentation, a graph node is used for each **BundleDescription** and graph connection is used for each derived relationship. The icon, colours and labels are then configured using the label providers. All these adapters made it very easy to modify which elements are shown and how they are displayed. For example, after a consultation with a graphics designer, the colours were modified to meet existing colour.

A high-level programming language supports the design, customization and implementation of an interactive visualization. However, to design effective views in this manner, a programmer, as well as someone with a deep understanding of the domain, is required.

	Visualization Tools	Concrete Visualizations	UI Supported Customization	DSL Supported Customization	Customized Through Programming	Developed by Us	Data Customization	Presentation Customization	Control / Behaviour Customization
Many Eyes	•		•				•		
BIRT	•		•	•			•	•	•
Jambalaya		•	•				•	•	•
PBS		•		•			•	•	
Creole		•	•		•		•	•	
PDE Viz		•			•	•	•	•	•

Table 4.2: Visualization Tools Comparison Chart

4.5 Discussion

To better understand the process of creating and customizing visualizations, two applications, three frameworks and a custom view were examined. Table 4.2 summarizes these six tools according to the customization options they support.

Creating and customizing visualizations for a particular domain is not an easy task. As the previous examples have shown, and the reference model for visualization reaffirms (Section 3.1) [CMS99], the data, visual attributes and control/behaviour must be designed appropriately for the end-users. Since it is difficult for anyone to fully evaluate a solution without a working prototype, the design and customization of views such as these is a highly iterative process, one in which developers provide users with working prototypes to evaluate. The results of these evaluations drive further enhancements, but these too must be evaluated. To support this cycle, developers, customizers and users need a mechanism to quickly incorporate changes back into the system and remove unnecessary complexity. Many engineers are integrating new tools within existing, familiar environments to help solicit feedback without requiring the end-users to switch tools or completely change their

work habits.

Integrated development environments, such as Eclipse, Visual studio, IBM Montana [SKBS97] and even Emacs, and office productivity tools, such as MS Office and Open Office, are effective because they integrate relevant tools into a single environment. In the case of the office productivity tools, spell checkers, spreadsheets, image editors, etc, are all available without requiring the user to switch applications or to learn new tools. In addition to the assortment of tools provided, these environments can be extended to add additional functionality through scripting, extensions points or an open API [KLM04].

Just-in-time comprehension [SLVA97] outlines the importance of providing comprehension tools to users *when they need it*. Since visualizations can often be used to help users understand information, visualizations should also be presented to the user in a just-in-time manner. This means that users should not be faced with the task of switching applications or exporting information in order to assist them with their current task.

While an open access layer is important to support integration, more than just embedding an application is often needed. A tool should conform to the same "look and feel" and provide the same interaction techniques as the system in which it is integrated. By tightly integrating visualization tools within existing applications, not only is "just-in-time visualization" supported, but it also helps lower a substantial barrier to adoption [FES03]. This is particularly important in software engineering tools where many developers believe they "don't need visualizations" [Die07] even though there is evidence that the cognitive support offered by visualizations improves productivity for some users [Die07, SDBP98, Zha03].

To support the creation and customization of visualizations, and the integration of these views into existing systems, we need to determine a general set of requirements for environments that support view creation. Distilling requirements and providing advice from the study of existing tools as been successfully done in other domains. In particular, Kienle studied relevant tool building experiences and case-studies to distill practical advice for how software components could be used to build reverse engineering tools [Kie06]. Wong studied the successes and failures of several tool integration and reverse engineering tech-

nologies to provide feedback into software understanding tool requirements [Won00].

Regarding visualization customizations, both functional requirements and design recommendations are distilled. The functional requirements are based on the aspects of software that can be customized, as outlined in Section 3.4: 1) data, 2) presentation 3) control/behaviour. The design recommendations are a direct result of our experience evaluating these six visualizations.

4.5.1 Functional Requirements

The functional requirements for a customizable visualization environment are based on the general requirements for any customizable software component and related to the customization of data, presentation and control / behaviour.

Data integration is an important aspect of any visualization. As described in Section 3.4.1, a visualization must be configured to show the relevant information and hide the rest. This is accomplished through a combination of operations that filter data, derive new data and move / rename data. The visualizations we evaluated reiterated the importance of these operations. The means by which the data were customized between the tools varied, but the importance of being able to easily specify simple things (such as filters) and still support complex data processing (such as data deriving queries) is clear. The tools that do not support complex queries often require custom code. This further separates configuration options and makes it harder to reproduce the visualizations.

As Section 3.4.2 outlines, the image in a visualization is what immediately grabs the users' attention. In the case of the six visualizations we studied, the majority of configurable visual elements revolved around choice of icons, labels, colours, and shapes. All of these attributes are commonly exposed through an API, prompting a recommendation that *a visualization customization environment should allow the customization of all exposed visual attributes.*

Control and behaviour are important aspects of any visualization as explained in Section 3.4.3. Many widget toolkits and frameworks outline which types of actions should be

customized and provide hooks for those developers who wish to override or add entirely new controls. Both the configuration of common behaviour (such as selection and mouse interaction) and the addition of new behaviour (such as the introduction of new layout algorithms) should be supported by a visualization customization environment.

4.5.2 Design Recommendations

In addition to the functional requirements, we propose several design recommendations. These recommendations were derived by looking at the facilities that make each visualization toolkit appropriate for the design and configuration of its views.

From studying these tools, the UI based approach provides the most efficient customization methods. While the UI based approach was the most efficient, since configurations could all be completed through option panels and dialog boxes, this approach was not nearly as flexible as the visualizations coded by hand. The plug-in dependency view was highly customized for a particular use.

In addition to the tradeoff between efficiency and flexibility, the importance of an integrated environment for designing visualizations was also realized. In the case of Jambalaya and Creole, the customizer could immediately see the results of their customization, providing verification that their modifications worked as intended, whereas in the case of the Portable Bookshelf, each modification required an entire batch process to be executed, and a new instance of the visualization to be launched. This constant switching between data processing scripts and viewing the results leads to a less streamlined feedback loop.

As well as integrating the view customization techniques within an integrated environment, any complex queries or transformation rules that exist should be written in a language familiar to the customizer. Developing a new domain specific language and adding it to the sea of technologies that already exist, means a steeper learning curve and higher barrier to adoption.

The learning curve is not only a problem for end-users but also for developers faced with integrating several toolkits and technologies. In many graphical toolkits, developers

are required to tie themselves to a particular framework, resulting in the maintenance of the original data model plus the view model. By providing model adapters (for both the data model and the presentation options) and designing the tool as an MVC framework, the developer can concern themselves only with their original data model. This design principle is generally known as "separation of concerns" [Som04, p.425].

Any existing viewers or widgets should be leveraged especially if a standard has previously been established. Users become familiar with the actions, look and feel, and interaction techniques provided by a system, such as single click vs. double click for action invocation. An unfamiliar widget toolkit often requires users to retrain themselves even for basic tasks [DFAB97]. If an existing widget is not available and a new one is designed, care should be taken when choosing default behaviours, colours and feedback mechanisms.

Finally, the importance of isolating the view from its specification was realized, not only to help separate the concerns, but to better document how the view was created. Very few tool designers, in research or elsewhere, adequately document how they construct their visualizations, making studies like this very difficult. If the views have been properly decoupled from the data, the customization code could be isolated.

4.5.3 Summary

In summary, an environment that supports the configuration and customization of visualizations should meet both the functional requirements outlined by Michaud and follow the design recommendations distilled through the analysis of these tools. Table 4.3 summarizes these requirements and recommendations.

From our study of these six visualization tools and the analysis of both the functional requirements and design recommendations, we now turn our focus to how visualizations can be created and customized using model driven engineering. Chapter 5 describes how PIMs can be used to capture the customizable properties of a visualization technique. Chapter 6 focus on the PSM of one such technique; and finally, Chapter 7 combines the two models to form **Model Driven Visualization**.

Functional Requirements:

F-1 **Data Customization:** A visualization must be configured to show the relevant information and hide the rest. This is generally accomplished through a combination of *data filtering*, *data deduction* and *moving / renaming data*.

F-2 **Presentation Customization:** A visualization customization environment should allow the selection and configuration of all exposed visual attributes.

F-3 **Control/Behaviour Customization:** An environment must support both an efficient method for action selection and Turing complete language for the design and creation of new controls and behaviours.

Design Recommendations:

DR-1 **Efficient view specification:** The manner in which view are customized (data, presentation and behaviour) should be done in a efficient manner when possible, while still supporting the case for complex operations.

DR-2 **Integrated tool support:** View customization should be performed in a tool that instantly allows the customizer to see the results of the efforts. As well, for a selected view, the customization options should be obvious.

DR-3 **Familiar language or notation:** While concise, efficient view specification is always sought, sometimes complex queries are required. This case, a familiar language or notation is suggested. This is especially important if the customizers are already using an existing language for other system customization specifications.

DR-4 **Provide view model adapters:** By providing model adapters (for both the data model and the presentation options) the toolkit manages the synchronization and allows the developer to concern themselves only with their original data.

DR-5 **Use existing viewers when available:** Since users familiarize themselves with colours, look and feel and the behaviour of many standard widgets, using a native toolkit allows a user to leverage this knowledge.

DR-6 **Explicit view specification:** By explicitly describing how the view is designed, modifications and enhancements can more easily be applied. As well, the design choices can be formally analyzed and future researchers can recreate the views from the improved documentation.

Table 4.3: Summary of requirements and recommendations for a visualization customization environment

Part II

The Solution

CHAPTER 5

A Catalog of Platform Independent View Models

THE grace and beauty of an experienced craftsman's work is astounding. The design always seems right and even the blemishes seem planned. An experienced interaction designer is no different. Visualizations seem to "make sense" and the intended effects appear to leap off the page. What makes these experts so proficient is experience.

Johann Gutenberg's printing press (1439) was one of the first tools to support the creation and sharing of visualizations. While book printing was considered an art form, and the skills required were passed down from master to apprentice, the computer, and other recent technical advancements, have enabled virtually anyone to create visual representations of information. In a field such as computer supported information visualization - where many people design and customize visualizations, but few are experts - capturing, publishing, and sharing proven designs is paramount. To effectively capture and communicate design rationale, an accepted notation must be used.

Various systems exist for classifying visualizations based on different facets, such as

dimensionality, interactivity and data-type. Often designed to guide researchers to new opportunities [Shn96], these systems are not necessarily intended for visualization designers, as they rarely discuss the customizable properties available for each visualization.

Enumerating all visualization techniques and customizable properties that each technique supports is neither feasible nor necessary as experienced interaction designers do not solve every problem from first principles [GHJV95, page 1]. Instead, they draw on previous experience and reuse solutions that have worked for them in the past. The purpose of this chapter is to document these design decisions and the customizable attributes of information visualization techniques as a set of patterns.

Patterns have been successfully documented and are used in several fields, including software design [GHJV95] and urban architecture [AIwMJ+77]. A *visualization pattern* is an expression of how an information visualization technique can be customized for a particular use. Each pattern includes a section to illustrate the problem it intends to solve and presents the solution through a formal model. The solution describes the variables, or configurable attributes, which can be used to customize the solution to fit specific needs.

The central concept in each of our visualization patterns is the formal model. The model describes the visualization in terms of its customizable attributes, in particular the data, presentation, and actions it supports. Each model is described using a MOF compliant class diagram and has also been modeled using EMF [BSM+03].

This chapter begins by reviewing the different types of data (Section 5.1) around which our visualization models are designed. The chapter continues by describing how platform independent view models can be described as a catalog of patterns. This includes a description of the pattern language (Section 5.2). Section 5.3 presents four such patterns for the bar chart, node-link, treemap and nested node-link visualizations. We then present some ideas for how the concept of visualization patterns can be extended using alternative modeling techniques (Section 5.4). Finally, Section 5.5 summarizes the chapter.

5.1 Types of Data

Data, in the most basic form, can be distinguished according to four basic types: **nominal**, **ordinal**, **interval**, and **ratio** [Ste46]. Nominal data are data that belong to distinct discrete categories such as eye colour (blue, green, brown, black). There is no rank or measurable difference between these types. The only operation nominal data supports is equality. Ordinal data are measurements that can be ranked. For example, students at a university can be categorized and ordered by year of study. Ordinal data supports both equality and ranking. Interval data are characterized by their scale, such that the differences between data points can be meaningfully measured. An example of the interval scale is the height of the students in a class. All interval data can be reduced to ordinal data, that is, all interval data has an implicit rank, however, the opposite is not true. Interval data supports equality, rank-ordering and equality of intervals. Finally, ratio data exists when the four operations: equality, rank-ordering, equality of intervals and equality of ratios, apply.

Within our models, we have introduced a few custom data types. These data types are related directly to one of the data types mentioned above, but provide more context. Included in these custom data types are binary (a nominal data type), longitude and latitude coordinates (both interval data), ImageURI (a nominal data type used to represent an image), and Label (a nominal data type used for headers and titles). As well, rectangle, point and dimension are defined (these are all tuples of interval data).

When new visualization patterns are presented, it is suggested that new custom data types be introduced if they provide more context and no suitable type exists. For example, a time-line may introduce the interval type **date**.

5.2 Pattern Language

The visualization patterns are both informally and formally described. Informally, each pattern describes when the visualization is appropriate and gives an example. Formally,

each view is presented with a platform independent model described as a MOF compliant model with constraints described in OCL. The model outlines how the data are structured. This should help an engineer decide if their data-set is properly suited for visualization using a particular pattern. Each visualization pattern has the following components:

1. The **model name** is used to identify the technique.
2. The **intent** informally describes what will be achieved by applying this technique to a particular data model.
3. The **motivation** outlines when this view technique should be applied and informally describes how a viewer for this data would be structured.
4. The **formal model** presents the platform independent models of the view technique. The models describes the *types* of elements rendered by this technique, as well as the configurable attributes and model constraints. Because visualization techniques can often be used for many structurally different data-sets, multiple models can be described for a single visualization technique.
5. The **configurable attributes** outline which visual attributes can be configured.
6. The **configurable user interactions** detail the interaction techniques that can be applied to this view to support both understanding and navigation.
7. The **known uses** gives a concrete example of how this technique has been used. Software toolkits that support this technique are also listed.

5.3 Catalog of Patterns

As in other fields where patterns have been utilized, the patterns described here are not new. Many of these patterns have existed for decades, but until now, they have not been formally captured. Visualization patterns help customizers choose between alternatives and make it easier to reuse successful designs. The patterns presented here are not meant to be

complete; however, they give an overview of the problem, solution and alternatives, for how visualizations are designed.

Four key visualization patterns have been documented here. These four patterns were the most common visualization techniques encountered while studying how visualizations are designed and customized (Chapter 4). An additional eight patterns have been included in Appendix A. This catalog can be expected to grow as other researchers contribute to it.

Bar Chart

Intent

A common visualization, especially in business reporting, is the bar chart. Bar charts are used in a variety of situations where non-continuous items (or groups of items) each have an ordinal or interval value. These charts can either be horizontal or vertical and the height (or length) of the bar represents the value.

Motivation

When faced with at least nominal items (or categories of nominal items) each with an associated interval value, it is often desirable to compare these items. Looking at sales records over time, or sales records by division over time, can indicate market trends, pin-point top performers and highlight problem areas. While commonly used to compare values, bar charts seldom scale as all bars should be visible to produce an effective visualization.

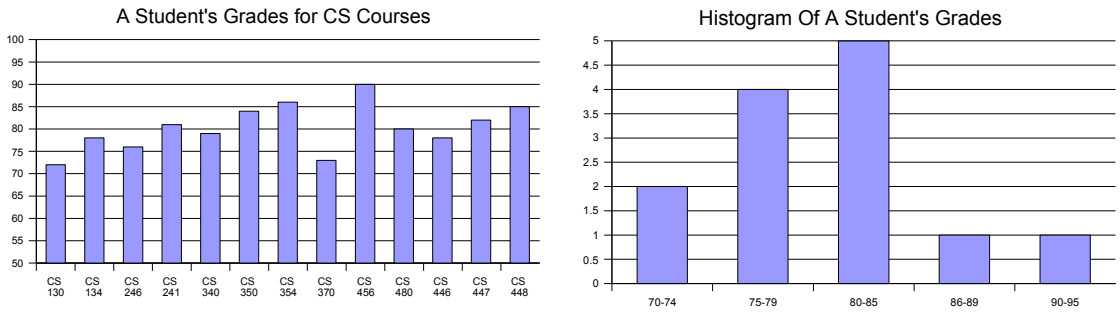

Figure 5.1: Example of a Bar Chart and Histogram

Another option, when faced with too many items, is to group the number of elements into “buckets”, and present the number of elements in each bucket (as often done in a histogram). Figure 5.1 shows the same data represented as a bar chart showing the ordinal

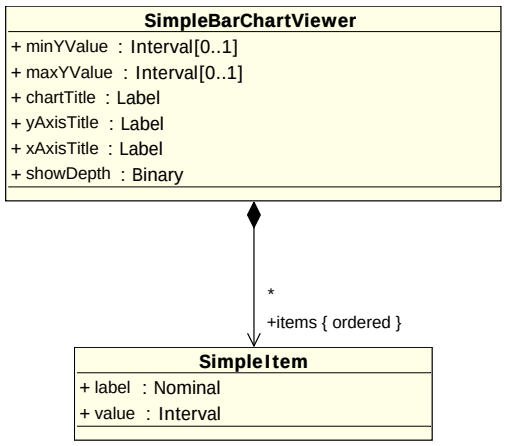

Figure 5.2: Bar Chart View – Formal Model A

data and a histogram that has grouped the items into buckets in five number increments.

Formal Model

Since bar charts can be used to represent a set of items, or a set of series, we have coded two different formal models. The first model (Figure 5.2) shows the basic chart. In this model, the **SimpleBarChart** has an ordered collection of items. Each item has a label and a value. In addition to the simple bar chart, we have also modeled a **CategorizedBarChart** in which each bar in the chart can belong to a category, or series (Figure 5.3). In both models, the viewer can be configured with a title, X and Y axis labels and minimum and maximum Y scale values.

There are a number of constraints specified for this model. The constraints are used to formally state that each bar in the chart must have a category and each category must have a value.

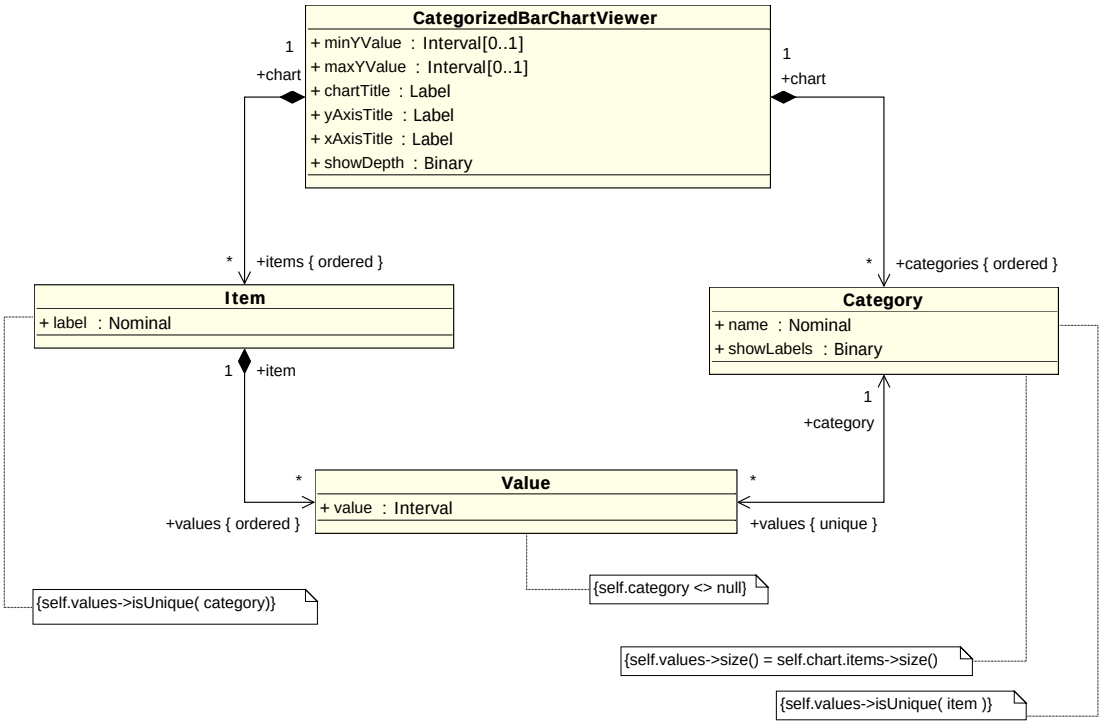

Figure 5.3: Bar Chart View – Formal Model B

Configurable Attributes

Colour, label and order are common configurable attributes for a bar chart. Colour can be used to distinguish between items and it can also be used to highlight items and bring them to the user's attention. The X-Axis, or item ordering, is also configurable as we are more inclined to correlate items or recognize chronological ordering if they are in close proximity.

Bar charts can also be rendered either vertically or horizontally. Often labels are easier to read when shown vertically.

Configurable User Interactions

Many bar charts support configurable user interactions such as a customized event upon selection. Tools such as BIRT support configurable user interactions through scripts which can be executed whenever the interaction occurs.

Known Uses

Since the introduction of charting into spreadsheet software such as Excel, bar charts have been used extensively especially in business reports such as the reports created by BIRT (see Chapter 4).

Typed Node Link (Graph)

Intent

Typed node-link diagrams, or graphs, use both nodes and connections to represent relational information spaces.

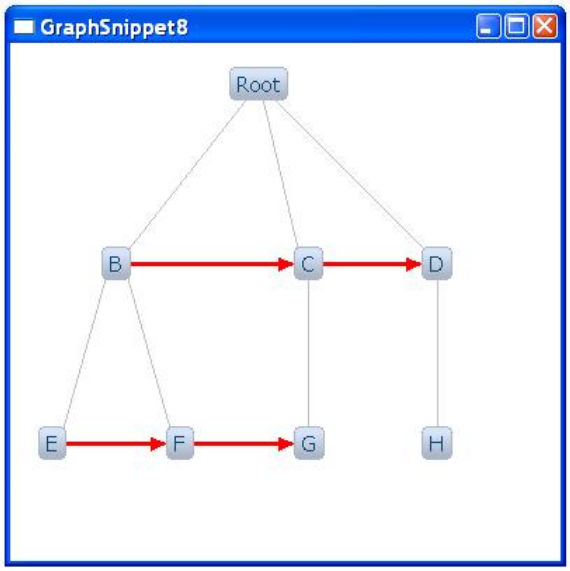

Figure 5.4: Graph Example

Motivation

When faced with a collection of elements, related to one another in some manner, understanding the relationships between these elements can often be a challenge. Examples of such a structure include the relationships between classes in a software system or the relationships between individuals in a Friend of a Friend (FoF) network.

In many cases, the elements (vertices) and connections (edges) have a type or colour. For example, in a software class diagram, a user may wish to distinguish between inher-

itance edges and composition edges. In a FoF network it may be desirable to separate friends from work acquaintances.

A problem with using a graph based approach is choosing, implementing and executing an appropriate layout algorithm to initially position all the vertices. In some cases, such as a class hierarchy diagram, a tree layout algorithm often produces aesthetically pleasing and easily understandable diagrams. However, in highly connected information spaces, such as a FoF network, a tree layout is not very useful.

The number of nodes, edges, available screen real estate and the structure of the graph must be considered when choosing a layout. Some layouts [BETT94], such as the spring layout algorithm where the nodes are positions based on "forces" between them, run in quadratic ($O(n^2)$) time, which for large graphs can be considerably slow. The performance in edge routing algorithms is often dominated by the number of connections instead of the number of nodes.

Formal Model

The formal model for the typed node link diagram contains five classes. The **GraphViewer** is the root of the model, and contains both the **GraphNodes** and **GraphEdges**. Each edge must contain a source and destination node, and each node can have 0 or more incoming and outgoing edges. Both the nodes and the edges have a type.

Configurable Attributes

A graph viewer often supports a number of configurable visual attributes. These attributes include size and position of the nodes, as well as colour, label, and icons on both the nodes and the connections. Connections can also support properties such as line thickness, the inclusion of arrow-heads or line style properties. Finally, the node-link diagram usually supports an initial layout algorithm.

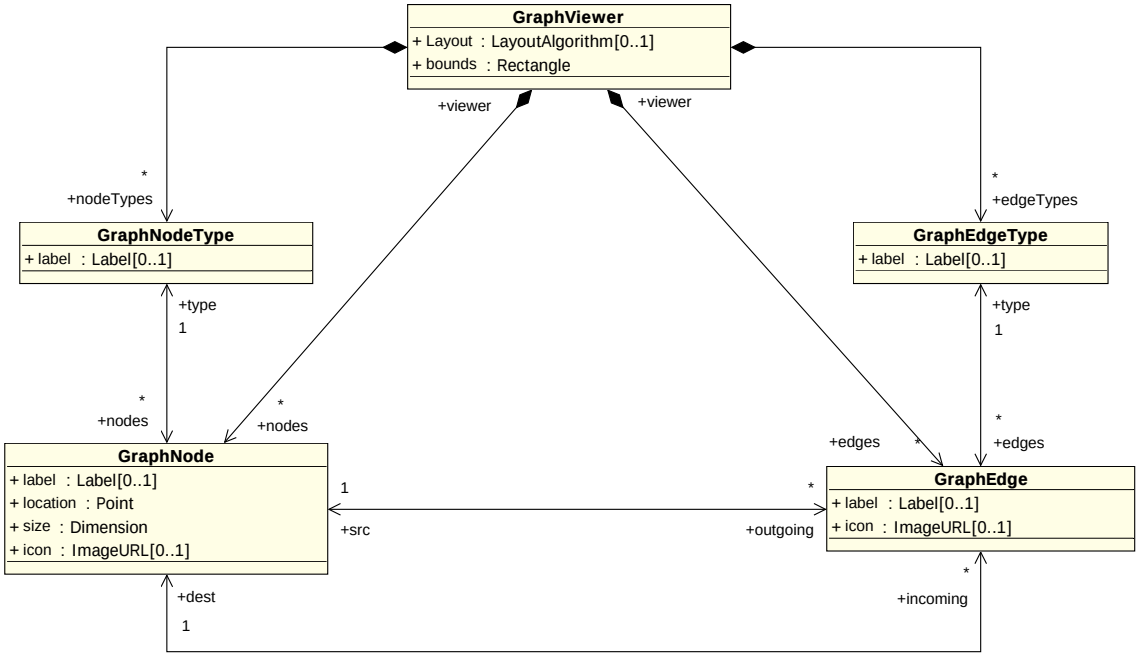

Figure 5.5: Graph Viewer – Formal Model

Configurable User Interactions

User interaction is very important, especially on large, highly connected graphs. User interactions such as highlighting neighbors, moving and resizing nodes, filtering, selecting, and layout have been used to help users explore complex, highly connected data-sets.

Other strategies such as fisheye views, zooming and panning, can also be used to provide user assistance. Storey described a technique through which nodes can be distorted, showing more information about the node while still providing context of the surrounding elements [SM96].

Known Uses

Node link diagrams have been used in a variety of domains, by a number of researchers to provide a visualization of everything from software systems to physical networks.

TreeMap

Intent

Treemaps are a hierarchical visualization technique which has successfully been used to represent over a million nodes [FP02] in space constrained views. Treemaps efficiently show hierarchically structured information using nested boxes of varying size and colour.

Motivation

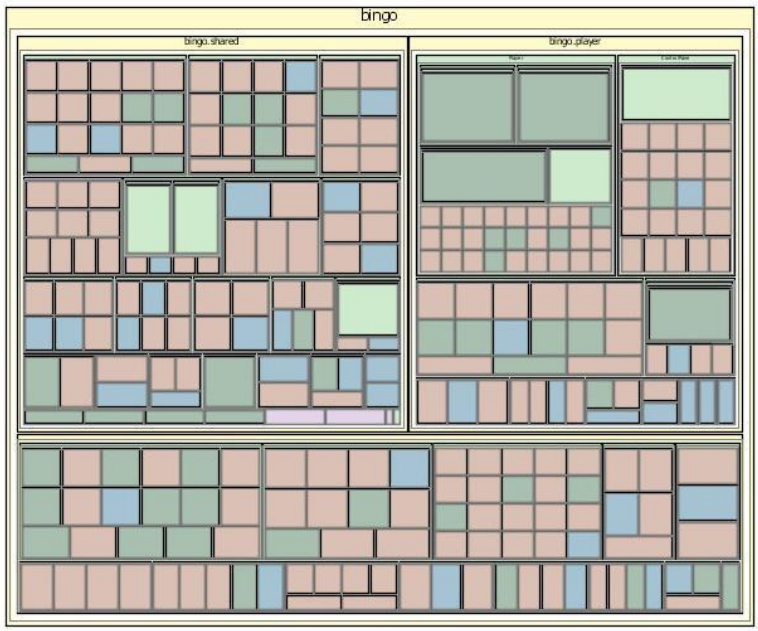

Figure 5.6: TreeMap

Treemaps were first designed by Shneiderman in the early 1990's to help explore disk space issues for the hard drive in his research laboratory [JS91]. The classical tree viewer (Appendix A Page 206) and node link diagrams (Page 89) grew too large to be useful, so Shneiderman explored ways of presenting trees in a space-constrained layout.

Treemaps work by turning a hierarchical data structure into a space-filling map. In many cases, the size of the leaf nodes are computed based on some attribute and sized relative to one another. The internal tree nodes are then sized based on the sum of the sizes of their descendants. In the example involving the hard drive, each file was represented as a box and sized based on the file size. The directories that hold the files were then subsequently sized based on the sum of the sizes of their descendants. This algorithm was recursively applied until only the root directories remained. This technique allowed the lab administrator to see which top level directories contained the largest files, and then allowed the administrator to "zoom-in" to investigate which files in these directories are consuming the most space. This approach helped identify both large directories that contain lots of small files (such as music files) or directories that contain a small number of large files (such as a core dumps).

Several algorithms to control the aspect ratio of the rectangles that make up the treemap have been proposed [BHvW00, Ben01, BSW02]. These algorithms include the ordered, squarified and quantum treemap algorithms.

Formal Model

The formal model for the treemap (Figure 5.7) contains three classes, the **TreeMapViewer**, **TreeMapContainer** and **TreeMapItem**. The TreeMapContainer is a subclass of the TreeMapItem. Each container, contains 0 or more items, and the items have a size, location, value and label. The viewer itself can be configured to work as a linear, ordered, or quantum treemap. This configuration will automatically compute the size and location of each item.

Configurable Attributes

The visual attributes of a treemap that can be configured are the colour, order and label for each item. The default algorithm, linear, ordered or quantum, can also be specified.

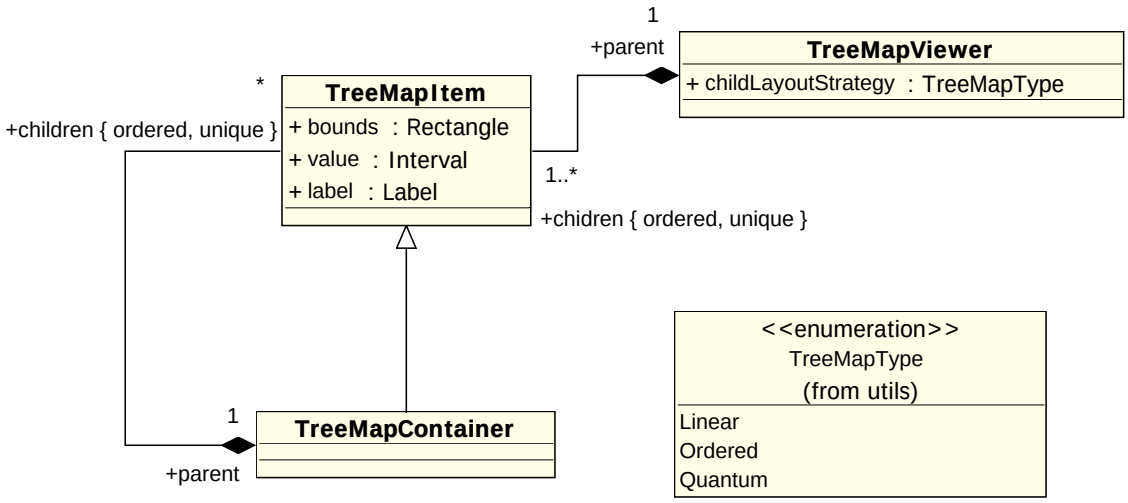

Figure 5.7: TreeMap Viewer – Formal Model

Configurable User Interactions

Selection and mouse-over hover operations are commonly supported by treemaps. Since treemaps often support several levels of nesting, the mouse-overs can provide information about node. Many treemaps also support zooming, so users can drill into a particular section of the tree.

Additional controls such as fisheye distortion and filtering can also be effectively used on a treemap viewer.

Known Uses

A treemap view is available as part of the Konqueror file explorer [Kon]. The viewer is useful for identifying directories and files which occupy a large portion of the system. The treemap viewer is also used in a popular online stock market tracking tool [Wat98, Wat99]. By organizing the stock market into industry verticals such as Health Care, Technology, Financial, Energy, etc. it can be easily seen which vertical is contributing the most to the market and in turn, which corporations are driving those verticals. Finally, the treemap has been used in reverse engineering tools [WMSL04].

Nested Node Link (Nested Graph)

Intent

The nested node link diagram combines power of a node link viewer with the structure of a hierarchical system. This view is ideal for information structured as a graph in which a set of edges span the graph as a tree. In this case, the "tree" edges can be used for nesting and then the containers and leaf nodes can be attached to one another using connections.

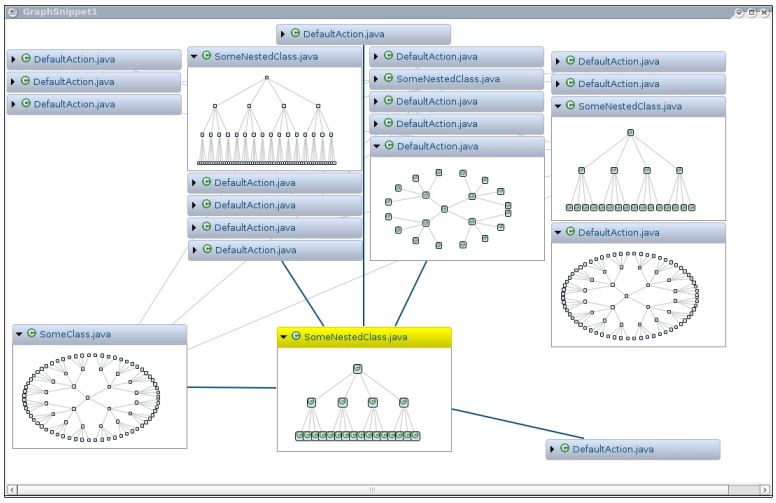

Figure 5.8: Nested Viewer

Motivation

When graphs contain an edge type which is inherently hierarchical, or spans the entire graph in a tree (or forest) like manner, it may help to nest the nodes instead of displaying these hierarchical edges. By nesting the nodes within one another, the same navigation facilities used by trees can be leveraged within graphs.

Unlike trees (Appendix A Page 206) or treemaps (Page 92), the nested node link viewer also supports edges connecting the elements. Nodes can be connected if they share the same

parent, or if they have different parents. While the latter can produce graphs where the edges seem to "fly-off" the screen, this can be addressed through a number of interaction techniques such as zooming and fisheye displays. Like node link diagrams, the nested variety also supports typed nodes and connections.

Formal Model

The formal model for this viewer (Figure 5.9) is similar to the node link viewer (Page 89) with the addition of the container node. The **NestedGraphContainer** is a subclass of NestedGraphNode, meaning it can be used in place of a graph node. Each node (or container) has a label and an icon. The edges connect nodes (or containers) to one another, and the nodes, containers and edges can each have a type.

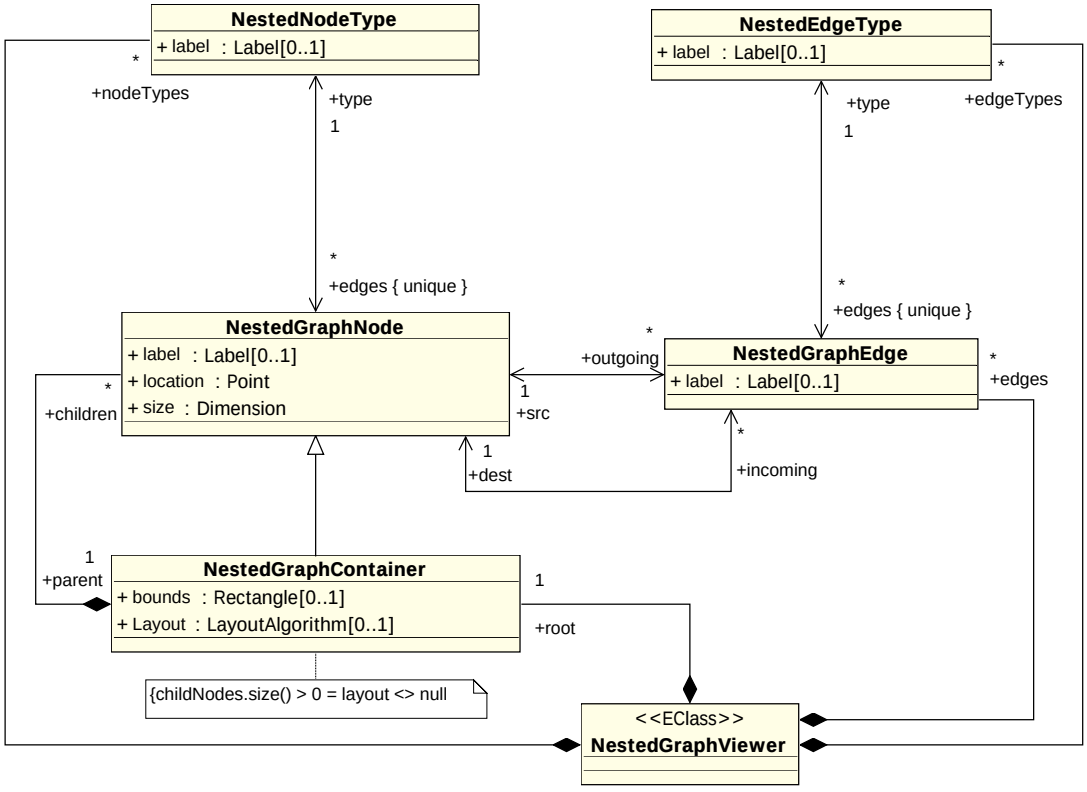

Figure 5.9: Nested Graph Viewer – Formal Model

The only constraint specified on this view is that a layout algorithm must be defined on any container which has one or more children. Other constraints can also be defined. For example, a connection may be limited so it only connects nodes with the same parent.

Configurable Attributes

Nested node link diagrams support the same configurable attributes as a standard node link diagram. This includes label, icons and colours for the nodes, containers and edges. In addition, the edges can support a line style. Finally, in a nested node-link view, an edge type can be specified as "hierarchical", meaning edges of that type will be used for nesting.

Unlike the graph, the nested node link diagram can utilize a different layout for each level of the hierarchy. For example, in Figure 5.8 some containers were drawn using a tree layout while others were laid out radially.

Configurable User Interactions

Nested node link viewers often support zooming and panning, allowing users to "dive-into" a container, as well as operations to navigate across an edge. Additionally, nested diagrams can support a fisheye view which enlarges a single container while still maintaining its context [SFM99].

Finally, nested graphs commonly support filtering and ability to expand / collapse containers to hide extraneous information.

Known Uses

Both SHriMP [BSM02] and LSEdit [Pen93] make use of the nested node link viewer to render graph based data. LSEdit is primarily used for program comprehension while SHriMP was originally used for this purpose but has been extended to a number of other domains such as medical ontologies.

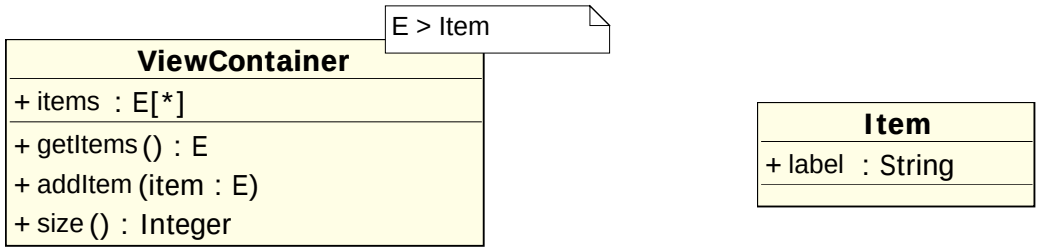

Figure 5.10: Template Model of an Abstract View Container

5.4 Future Work

The visualization patterns described here (and in Appendix A) outline both the structure, and configurable aspects, of the visualization techniques. While the models have been instantiated and used in a number of systems (Chapter 7 and Chapter 8), other design alternatives are available. Two avenues for future work include, better element reuse between models and the design of rendering models.

In general, the models have been designed in isolation so each model could be created and evolved independently, however, there are compelling reasons for building a single environment for all the models. For example, lists, maps and charts all exhibit a similar structure, as they each have a viewer that contains a number of items. Templates could be used to encourage reuse between these models. Figure 5.10 shows an example of how an abstract view container could be modeled, and then a concrete view model (such as a tree or a map), could bind items to this container (Figure 5.11).

As well as reusing elements between models, the way in which the models are described could be altered. Currently, each model describes the structure and configurable aspects of the visualization technique. On the other hand, the visualization could be described in terms of how it will be rendered. For example, instead of modeling the nodes, containers and edges, in a nested node-link diagram, the model could have been designed in terms of rectangles, lines and pixels. A description of how these view models are transformed to rendering models would completely remove the need for platform specific viewers, as any

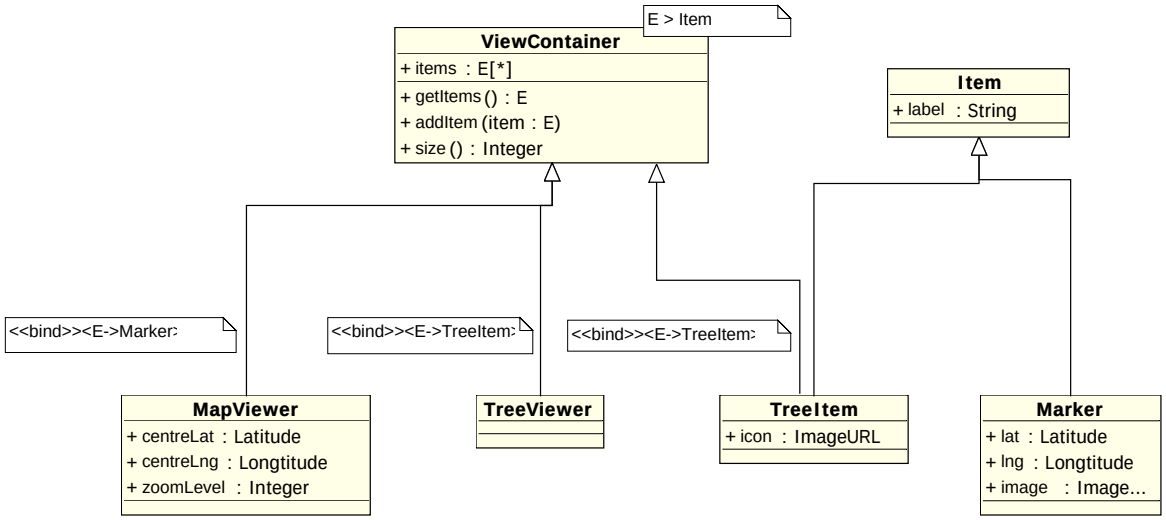

Figure 5.11: Concrete View Models Based on the View Container

compatible rendering engine (such as OpenGL) could be used to execute the view. In our current implementation, platform specific viewers are used since many of these tools have years of development and are well polished.

5.5 Summary and Limitations

A visualization pattern provides a mechanism to document the configurable features and formal model for a visualization technique. In this chapter we have presented four such patterns for key visualizations. In addition to the four patterns presented here, Appendix A presents eight more. Each of these patterns present a platform independent view model that can be used by tool designers when creating platform specific visualization rendering technologies.

The patterns provide a means for interaction designers to better document the knowledge they have about information visualizations, in a way that can be easily consumed by others. This documentation will help future engineers design custom views since the data, attributes and customizable controls, are all explicitly stated.

The platform independent patterns presented here describe a single model and the common customizable attributes for each visualization technique. For each pattern, there are several different ways the model could be structured and a number of additional attributes that could be considered. Using facilities such as a "meta-model zoo" [Fav04a], we hopefully can encourage other modelers to extend these patterns by adding alternative models, constraints, and additional configurable view elements.

Another limitation of these models is the lack of formal validation. Designing models, much like software, is a highly iterative process influenced by a large number of requirements. While the models have been used extensively throughout this work, and test cases have been designed to ensure quality, no formal approach has been used to guarantee validity. While this may be seen as a problem, we believe that by placing the models in the public domain, others can help refine and perfect the models.

Finally, while a visualization pattern is a means of representing a view technique's data model and configurable elements, an instance, or platform specific implementation must be developed to enable the visualization to be incorporated within a software system. In the next Chapter we describe an implementation strategy that can be used for several of the graph based visualization patterns including: 1) node-link (Page 89), 2) treemap (Page 92), and 3) nested node-link (Page 95).

CHAPTER 6

Zest: A Visualization Toolkit for Eclipse

FORMAL view models are central to the realization and application of Model Driven Visualization. A Platform Independent Model (PIM), as presented in Chapter 5, describes possible data, customizable attributes and associated actions for each visualization pattern. While these models formally describe visualization techniques, a Platform Specific Model (PSM) is needed if a visualization is to be used in a software system. PSMs can either make use of a generic rendering engine or a **viewer** to display the information. A viewer is a user interface component for managing and displaying information and conforms to a PIM. We have chosen to use viewers in this work as many highly optimized ones exist. Examples of a viewer include: bar charts, graphs, maps, tables and trees. When a viewer is instantiated, such as a map viewer that shows a map of Canada, it is known as a view. Figure 6.1 shows the relationship between a view, viewer and associated PIM.

To include an information visualization in an application, a software engineer must first select and instantiate a viewer, and then populate the instantiated viewer with the ap-

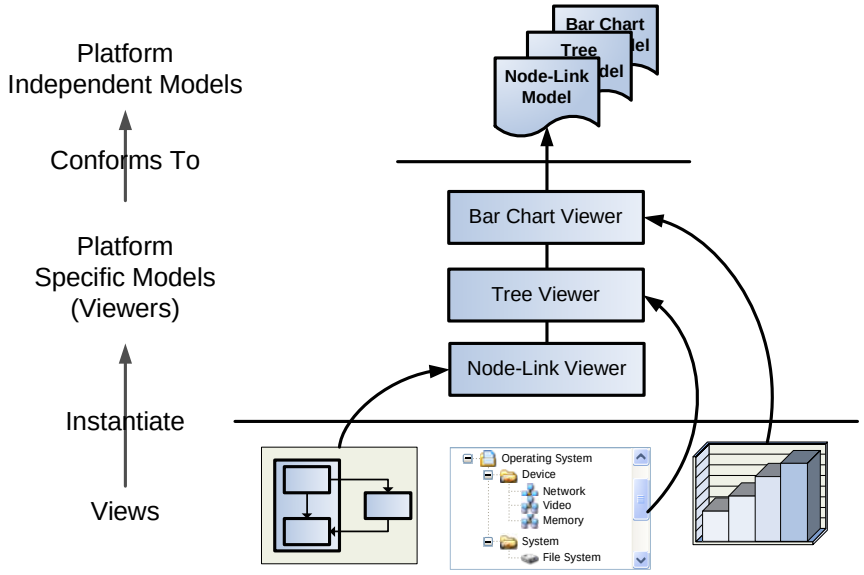

Figure 6.1: Relationship Between a View, Viewer and Model

propriate data. To populate a viewer with data, a software engineer must either align the data-model with the view-model, or design an adapter to bridge the two models. While both approaches are commonly used, they each have a disadvantage. Aligning data and view models results in a highly coupled system, while a standard model adapter places the burden of synchronizing models in the hands of the software engineer.

In this chapter we describe Zest, a graph visualization toolkit that provides a mechanism to separate the data-model from the view-model using an MVC framework and a concept known as a content provider [SHNM05]. The MVC framework supports the configuration and customization of a visualization without requiring the developer to manage the model synchronization, or construct data models conforming to the viewer's representation.

When using an MVC framework and a set of content-providers, the synchronization of the data and view-model is managed by the viewer, instead of the developer. Zest is a visualization toolkit that enables a developer to make use of a content provider for node-link and nested node-link diagrams. Zest is intended for software developers who wish to include graph based views within their application. While Zest focuses on node-link diagrams, the concept has also been applied to sequence charts (Section 6.5).

6.1 Modularizing Graph Based Visualizations

When designing user interfaces, most software engineers use an existing widget toolkit. Widget toolkits provide controls such as buttons, sliders, menus, trees, tables and lists. These interfaces are usually constructed by laying out controls in a window, essentially using the controls as basic building blocks. However, when developers require richer interfaces, custom solutions are designed or a mixture of unrelated technologies are integrated. Two outcomes are a result of this: 1) very few applications actually make use of the graph based views, and 2) when these techniques are leveraged, there is no standardized interaction mechanism.

To improve the adoption and standardize the interaction styles for graph based views, a node-link widget is needed. Zest has been designed to address both of these issues by providing a standard nested graph widget for SWT [NW04]. As an extension to SWT, Zest's API is similar to that of other widgets, meaning that developers who are comfortable with SWT and the design patterns it employs, should have little difficulty using Zest. There are four main elements in a Zest view: 1) nodes, 2) connections, 3) containers, and 4) the graph itself (see Nested Node-Link pattern on Page 89).

Each of the classes in Zest have a number of attributes that help control the appearance of the graph. The nodes can have a colour, size, label, and icon. Connections attributes are line style, label, width, and colour. Containers attributes are colour, size, label, and icon, and finally graph attributes include background colour, bounds and layout algorithm.

6.2 Why Design this Framework?

Several graphing toolkits and applications, such as SHriMP [BSM02], Jambalaya [SMS+01], Creole [LMSW03], JGraph [JGr], GEF [GEF], YFiles [ytdc], Jung [JUN], and Prefuse [HCL05] exist, questioning the need for yet another toolkit. Many of these toolkits have years of development, solid documentation and an active user community. While many of these

frameworks are very powerful, none of them met all of our needs for a model based visualization toolkit.

Applications such as SHriMP and YFiles are very stable and easy to use, however, both of these applications are geared towards end-users and are not intended to be integrated with existing tools. Consequently if a user wants to visualize information in one application, they first have to export their data, launch a second program, and use this second program to generate the visualization. The cognitive burden of constantly switching tools is not conducive to tool adoption.

Toolkits, such as SHriMP, integrate well with other tools like Protégé and Eclipse, however, since the entire application is integrated with these tools, the separation of data, presentation and control is difficult. To achieve this level of customization, a finer grained integration strategy is needed.

Jung and JGraph offer an integration mechanism with a finer grain of control than SHriMP. These tools could have been leveraged for our work, however to make use of these toolkits within a software system, developers are expected to fit their data-model with the toolkit's view-model. This is often achieved in one of the following ways. The developer can include information about the visualization directly within the data-model, tightly coupling the data with the view, or the two models can be developed separately and an adapter created to keep the models synchronized. Another common approach for separating the view and data-model is through the use of an MVC framework.

6.3 MVC Framework

An MVC framework for visualization tools is enabled through the use of content providers. A content provider is a single interface used to reconcile a view and data-model. The content provider interface represents all the elements in a view-model, and it is the responsibility of the engineer using the toolkit to implement this interface for their specific data-model. Listing 6.1 shows a content provider for the node-link diagram. Content providers have

```
1  public interface GraphContentProvider extends IContentProvider {
2
3      public Object[] getNodes (Object inputElement);
4      public Object[] getEdges (Object inputElement);
5      public Object[] getNodeTypes (Object inputElement);
6      public Object[] getEdgeTypes (Object inputElement);
7
8      public Object getTypeOfGraphEdge (Object graphEdge);
9      public Object getTypeOfGraphNode (Object graphNode);
10
11     public Object getDestination (Object graphEdge);
12     public Object getSource(Object graphEdge);
13
14     /** Attributes **/
15     public Object getLayout (Object inputElement);
16     public Rectangle getBounds (Object inputElement);
17     public String getLabelForEdge (Object graphEdge);
18     public IconURL getIconForEdge (Object graphEdge);
19     public String getLabelForNode (Object graphNode);
20     public IconURL getIconForNode (Object graphNode);
21 }
```

List of Listings 6.1: Content Provider for Node-Link Viewer

been successfully applied to tables, trees and lists in the JFace toolkit [GB03, SHNM05], but we have not seen it used in more complex visualizations.

By implementing this interface, a developer indicates which artifacts in the data-model are represented by which elements in the view. For example, the developer using this toolkit indicates which data elements represent nodes, edges, node types, edge types, and how these elements are related.

Listing 6.2 shows how a developer could implement this interface to create a graph representing the rules in a Paper-Rock-Scissor's game (Figure 6.2).

Once the content provider interface is implemented (Listing 6.2), the MVC framework uses it to construct the internal representation. The construction algorithm works as follows. The toolkit first queries the content provider for all elements in a depth-first manner (Figure 6.3-A). A mapping between the objects in the data-model and view-model is created. The mapping is traversed again to resolve all reference and attributes (Figure 6.3-B). This instantiated view model can now be rendered using a toolkit such as Zest.

In addition to creating a visualization, the content provider approach supports dynamic updates. Updates can be applied at the granularity of subtrees (Figure 6.3-C). If a subtree is

```
1  class GraphContentProvider extends GraphContentProvider {
2
3    public Object getDestination (Object graphEdge) {
4      if ( graphEdge.equals("breaks")) return "scissors";
5      if ( graphEdge.equals("cuts")) return "paper";
6      if ( graphEdge.equals("covers")) return "rock";
7      return "";
8    }
9
10   public Object[] getEdges(Object inputElement) {
11     return new String[]{"breaks", "cuts", "covers"};
12   }
13
14   public Object[] getNodes(Object inputElement) {
15     return new String[]{"rock", "paper", "scissors"};
16   }
17
18   public Object getSource(Object graphEdge) {
19     if ( graphEdge.equals("breaks")) return "rock";
20     if ( graphEdge.equals("cuts")) return "scissors";
21     if ( graphEdge.equals("covers")) return "paper";
22     return "";
23   }
24 }
```

List of Listings 6.2: Code Snippet for Zest Example

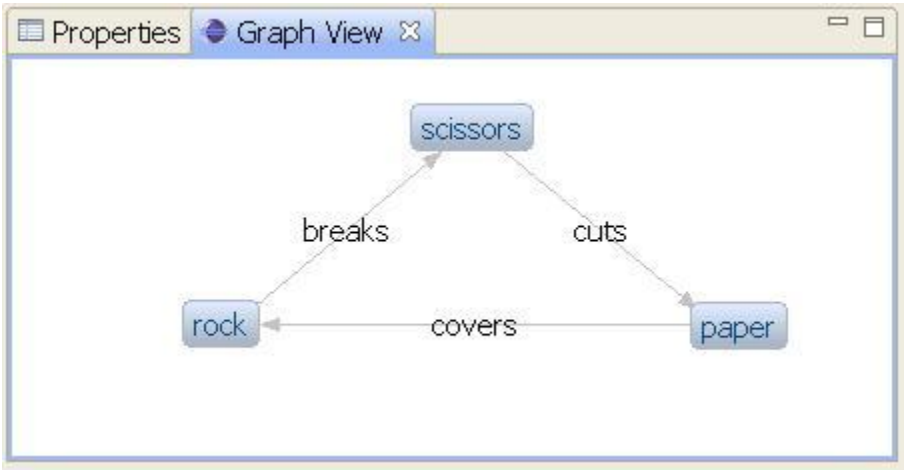

Figure 6.2: Zest Screenshot

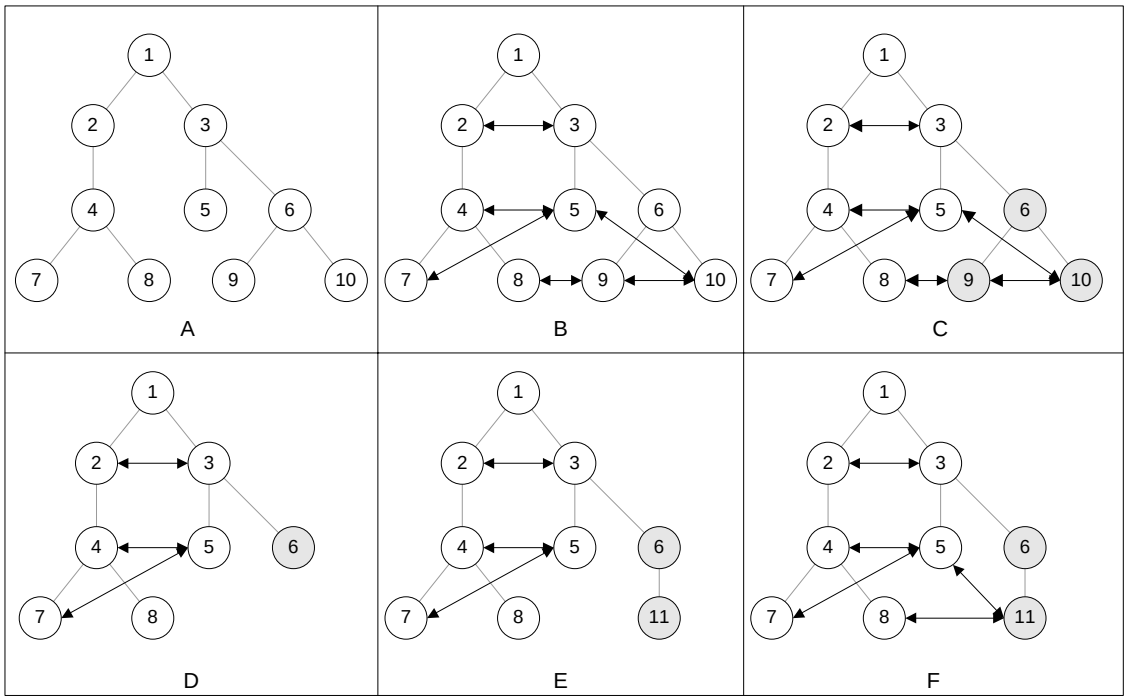

Figure 6.3: Overview of Adapter Algorithm

invalidated, it is removed from the view. The removal includes all elements, relationships and attributes (Figure 6.3-D). Starting from the root of the subtree, the content provider is used again to re-query the user model and recreate the subtree again in a depth-first manner (Figure 6.3-E). Finally, this new subtree is queried again for any attributes and references (Figure 6.3-F).

Zest provides an MVC framework and content provider approach to view creation for Eclipse developers wishing to design visualizations. The toolkit has been implemented in Java using SWT [NW04] and GEF [GEF] and the code is now part of the official Eclipse release, available for download from: *http://eclipse.org/gef/zest*. To avoid redundancy and to leverage the work that has already been performed in the area of graph visualizations, Zest builds upon SHriMP [BSM02], Jung [JUN] and GEF [GEF]. Zest uses the Draw2D rendering library from GEF and the layouts from Jung and SHriMP.

While the MVC framework enables the use of existing data-models, a custom content

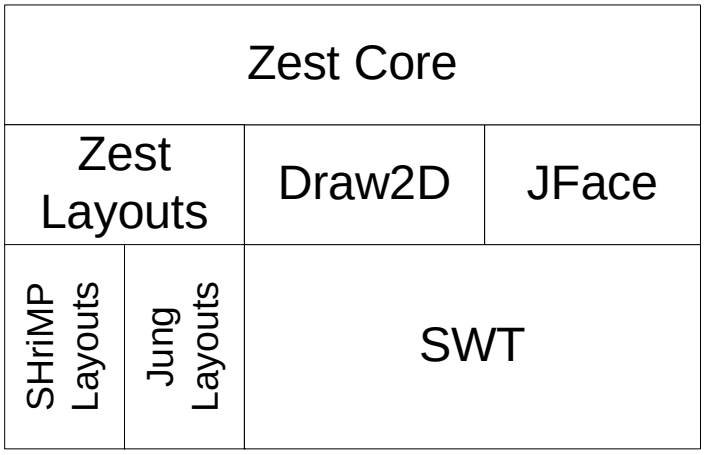

Figure 6.4: Zest Architecture

provider and mapping algorithm must be implemented for each viewer. Also, whenever the model changes, the content provider and the custom mapping algorithm must be updated.

6.4 The Architecture of Zest

Zest has two main components, *Zest core* and *Zest layouts*. The layouts package is a device independent library of algorithms. As a device independent library, the layout package can be used with a variety of rendering engines. Each algorithm assigns size and location information to nodes and edges in a graph. Some examples of layout algorithms that we currently support are: 1) tree, 2) vertical tree, 3) radial, and 4) spring [BETT94].

Zest core is an SWT based rendering engine for nodes, edges and containers. Built on top of Draw2D, Zest provides a number of optimizations for large graphs and integrates smoothly with other SWT / Draw2D interfaces. Figure 6.4 shows an overview of the Zest architecture. This layered architecture implies that components are dependent on those components below them. Zest requires the layouts, Draw2D and JFace, while JFace and Draw2D require SWT.

6.5 Additional Zest Views

In addition to node link diagrams, the concept of a content provider based visualization tools has also been realized by other researchers. Using patterns inspired by Zest, a general purpose sequence chart was designed by Del Myers as described by Bennett *et al.* [BMS+08].

The sequence chart viewer was designed by synthesizing the available features provided by existing sequence chart tools. Many of the existing tools worked as stand-alone applications, and Bennett *et al.* realized this limitation and designed a reusable sequence chart. By first reviewing the existing tools, Bennett *et al.* uncovered a set of requirements and subsequently designed a visualization model for sequence charts with a series of customizable elements. The sequence chart viewer has been deployed and used in a number of projects including a tool to visualize runtime interactions of a Java program.

6.6 Discussion

By creating visualization toolkits as a set of reusable components, and designing the viewers as an MVC framework with associated content providers, software engineers can experiment with different visualization techniques and more easily integrate visualizations into other systems.

From our experience with Zest, we found it was important to state exactly which features Zest supports through the use of a formal model. By designing a formal model, we could decide which elements and attributes to expose. Since we have the models (Chapter 5), it was very easy to perform this analysis. Only the elements, attributes and references outlined in our model were exposed. When new functionality was added, we first explored how the model could be extended and how these extensions would impact the design. Once this extension was completed, these features were then exposed in Zest.

Exposing new configurable elements means the content providers must be modified.

This is accomplished by examining the formal view model and adding the appropriate methods manually. In the next chapter we discuss how this process can be automated.

CHAPTER 7

Model Driven Visualization

THE need to customize visualizations for particular users and their tasks is a complex process, one supported through a variety of technologies. While toolkits that support customization through the user interface are easy to use, they lack the expressiveness needed to specify complex views. On the other hand, custom coded visualizations are able to support complex views and enable tight integration with existing systems, however, the construction of such views requires a designer with a high degree of technical skills. Building custom visualizations in this manner in results in views designed close to the technical or solution space rather than the problem space.

Model driven engineering proposes to combat complexity, delay the selection of key architectural decisions and support software customization through problem decomposition and separation of concerns [SK02]. Using an MDE approach to software development, designers work closer to the problem space rather than focusing on the technical aspects of a solution. Since developers are beginning to design and customize software through models

and model transformations [SK02, Sel03, BCT05, BSM+03], we set out to investigate if this approach is feasible for view creation.

In this chapter, we describe Model Driven Visualization (MDV), a process for designing and customizing interactive visualizations using principles from model driven engineering. To demonstrate how MDV can be used for the design and customization of information visualizations, we designed a series of views to answer three questions posed to us by hockey enthusiasts. Section 7.1 presents a relational data-set describing 87 years of NHL statistics that will be used throughout this chapter as a running example. We chose this example since the domain is easy to understand, the data are readily available, and it is familiar to many in Canada. Section 7.2 describes the architecture of MDV. The core concept, the platform independent view model is revisited in Section 7.3, followed by a description of how an MVC framework can be generated for these view models. Section 7.4 outlines how views can be formally customized using models, and the NHL statistics model is used as an example. The chapter also discusses how multiple views can be linked using MDV (Section 7.5) and a brief overview of the available tool support is outlined (Section 7.6). Finally, the chapter concludes (Section 7.7) by summarizing how MDV can be used to customize visualizations with respect to the functional requirements and design recommendations distilled in Chapter 4.

7.1 NHL Statistics: A Running Example

In order to demonstrate to the reader how MDV is used to design and customize visualizations, a running example will be used. Several views of different National Hockey League (NHL) statistics will be created. The data-set is of moderate size consisting of 87 years worth of data, 1,115 team statistics and 31,965 individual player records. We worked with several hockey enthusiasts to design unique views to answer some specific questions (Table 7.1).

Figure 7.1 shows the UML model to which the data conforms. The model contains 7

1. In what years were the most goals scored?
2. Given a player, what teams did he play for throughout his career?
3. Given a player, how many points did he score each year of his career?

Table 7.1: Questions regarding NHL Statistics

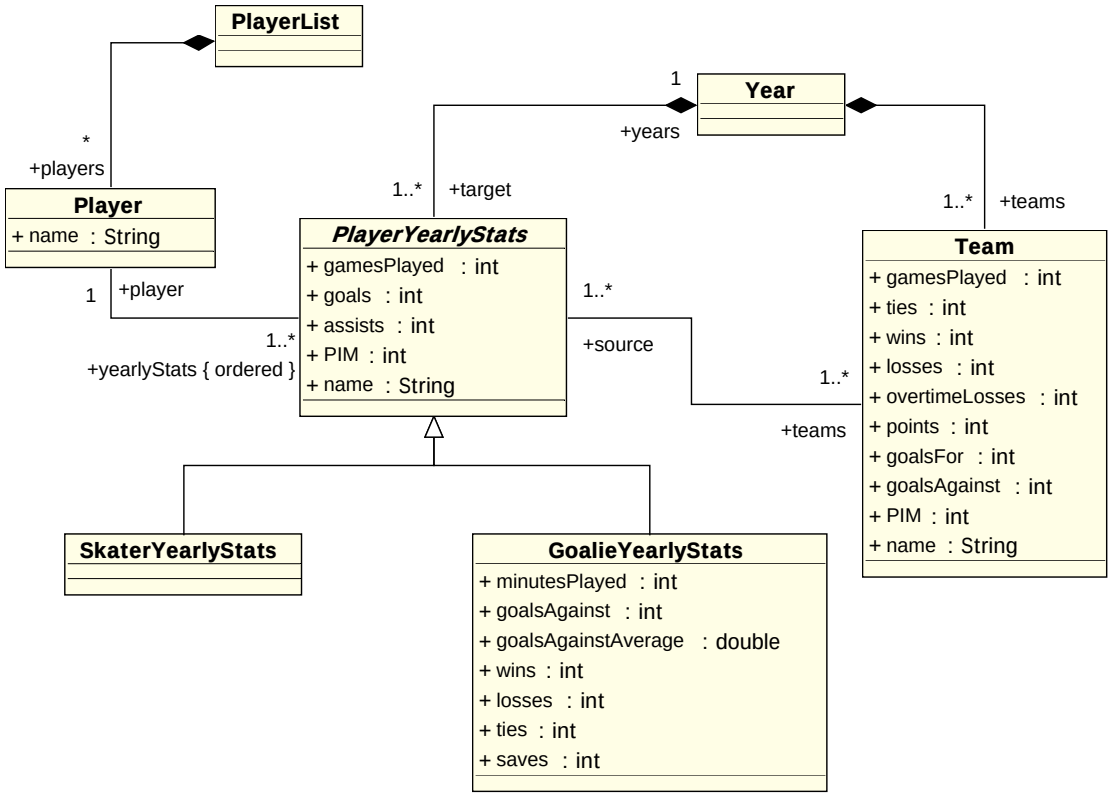

Figure 7.1: NHL Hockey Statistics Model

classes: **Year**, **Team**, **PlayerYearlyStats**, **GoalieYearlyStats**, **SkaterYearlyStats**, **Player** and **PlayerList**. The **Team** and **PlayerYearlyStats** classes are the interesting ones as they contain most of the information. The information was mined from `hockeydb.com`.

7.2 MDV: An Overview

MDE focuses on the tools and technologies to enable engineers to specify solutions in a platform independent manner, and through a series of model transformations, a platform specific solution can be generated. To assist engineers with the design and specification of information visualizations, MDV defines a number of platform independent view models (Chapter 5), and platform specific viewers (Chapter 6). Using MDV, a view can be designed in two different ways. Designers can use the viewers programmatically and customize the view using a content provider. This is intended for designers who are more comfortable using a high-level programming language and an MVC framework.

Another option for designers who wish to specify solutions in a truly platform independent manner, is to specify transformation rules from the data-model to the view-model and use a transformation engine to interpret the rules and render the view. Both these solutions help satisfy the design recommendation that *views should be specified in a familiar language or notation* (DR-3). Also, by specifying the view using a content provider or model transformation language, the *customization options becomes explicitly stated* (DR-6).

Figure 7.2 shows the components of MDV [BSFL06]. As presented in Chapter 5 (and expanded in Appendix A), we have designed and documented several platform independent models and constraints for common viewers (Figure 7.2-A). For each model, a viewer exists (Figure 7.2-B). The concrete viewers are able to use any instance of the view model to render the data (Figure 7.2-C). Eclipse's JFace viewers are leveraged for the tree, list and table controls. The Business Intelligent and Reporting Tools charting engine is used for the charts and Zest [BBS04] is used for the node-link diagrams. Since Zest, BIRT and JFace are used in Eclipse based products (the current target for our technologies), using these

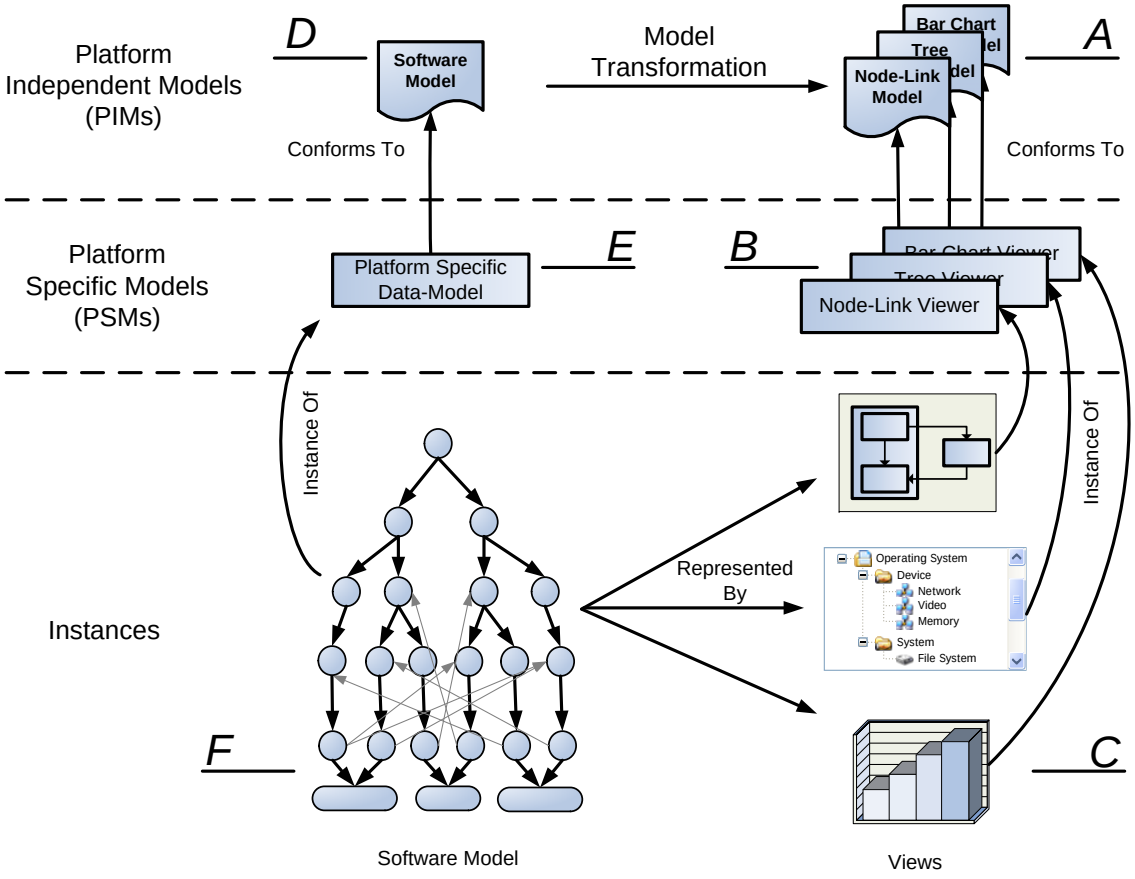

Figure 7.2: Overview of MDV

viewers provides a familiar look and feel to the user, satisfying DR-5: *use existing viewers when available*. Of course, the viewers are not restricted to these rendering engines, and Swing trees or JFreeChart view could be used instead.

In order to generate a view from a data model, the designer first chooses the viewer and then constructs the transformation rules to describe the view. The transformation rules are applied on the instance data (Figure 7.2-F) by matching elements from their corresponding data model (Figure 7.2-D). The rules describe which target elements should be generated. We have tested our MDV framework using both the Atlas Transformation Language [JK06] and oAW's xTend tool [Voe] (described in Section 2.5).

7.3 Generating the Models

The models presented in Chapter 5 present accurate documentation for a viewer and outline the properties that can be configured. In Chapter 6 we described how content providers enable a developer to use these models without tightly coupling their application to a specific view-model. However, the manual construction of the MVC framework is a time consuming task, one that must be repeated for each visualization pattern.

To address the enormous development effort that would be required to create all these content providers, we turned to generative programming to automatically generate the custom content providers from the PIMs [CE00]. By automatically generating the content providers, new models can be added to the collection and leveraged immediately. The code generation facility generates both the content providers and the custom mapping algorithm which uses the content providers to synchronize the view and user models. The Java Emitting Template tool (JET) [Pop04] is used to generate the code. The actual template is available in Appendix B.

In addition to the content providers, EMF generates an XML tool suite. The XML tool suite provides serialization/de-serialization capabilities and an XML Schema for each viewer [W3C04]. Using this approach, a designer can specify a visualization using XML.

```
<viewers xsi:type="vizmodels_5:MapViewer">
  <markers lat="53.5666" lng="-113.5166666"
           label="Edmonton: Edmonton Oilers\
                  City of Champions" imageURL="edmonton_oilers.gif"/>
  <markers lat="34.05" lng="-118.2333"
           label="Los Angeles" imageURL="la_kings.gif"/>
  <markers lat="38.65" lng="-90.6333"
           label="St. Louis" imageURL="stlouis_blues.gif"/>
  <markers lat="40.78333" lng="-73.9666"
           label="New York" imageURL="newyork_rangers.gif"/>
</viewers>
```

List of Listings 7.1: XML Data for the Map View

7.3.1 Running Example: Generating the Map Viewer

To create a visualization using XML, an engineer can serialize their data as an XML document conforming to the view model's schema, and pass the XML document to the viewer for rendering. For example, the XML in listing 7.1 can be passed to the map viewer to produce the view in Figure 7.3[1].

In addition to the XML and programmatic approach, MDV can be leveraged for the creation and customization of visualizations through the use of model-to-model transformations. The remainder of this chapter will be dedicated to this approach to view creation.

7.4 Formalizing View Creation

So far we have focused on how visualization techniques can be formally modeled and how these models can be used within a software system through the use of an MVC framework or shared XML data-sources. While these practical approaches to view creation are suited to many object oriented and imperative programming styles, they still tend to tie a software engineer to a particular language or set of technologies.

To raise the abstraction level of view customization away from the source code and better integrate with systems being designed using MDE, we have investigated if visualizations can be designed and customized using a model transformation language. At this point, we

[1] Google maps and the SWT native browser widget were leveraged to render the data, but any geographical mapping toolkit which supports markers could be used.

Figure 7.3: XML Data Rendered in a Geographical Map

have not seen anyone else who has attempted to use model transformation engines to design and customize visualizations, and it was unclear to us if these transformation engines could be leveraged to support all the customization styles outlined in Section 3.4.

Model transformation languages, like ATL [JK06] or QVT [OMG07a], seem like a reasonable choice for data customization since they concisely represent simple model mappings, a common customization activity. One-to-one mappings between model elements and view elements can be done in a declarative manner. For example, in the case of our hockey example, the creation of a chart highlighting the number of goals Wayne Gretzky scored each year of his career can be specified with two rules.

The first rule (r1) matches all **Players** named Wayne Gretzky. A bar chart is created and a number of visual attributes are configured. The second rule (CreateItem) creates the actual data point and configures the label and icon for each year of Wayne Gretkzy's career.

While this language, or technique, for specifying software solutions may not seem intuitive at first, it is important to note that transformation languages are very common to those

```
1  rule r1 {
2    from  s : Hockey!Player (
3      s.name = 'Wayne Gretzky'
4    )
5    to t : VizModel!SimpleBarChartViewer (
6      chartTitle <- 'Wayne Gretzky\'s Goals by Year',
7      xAxisTitle <- 'Goals',
8      yAxisTitle <- 'Years',
9      showDepth <- true
10   )
11   do {
12     t.items <- s.yearlyStats->collect( e | thisModule.CreateItem(e) );
13     t;
14   }
15 }
16
17 rule CreateItem (s : Hockey!PlayerYearlyStats) {
18   to t : VizModel!SimpleItem (
19     label <- s.year.year,
20     value <- s.goals
21   )
22   do {
23     t;
24   }
25 }
```

List of Listings 7.2: View Mapping to create a Bar Chart

engineers building systems using MDE, the target audience for this work.

Declarative transformation languages work in an additive manner, that is, data are only added to the target if it matches a rule. All data elements that do not match a rule are filtered. While data filtering is an important part of the view customization process it does not address the more complex case of data processing or data deriving operations.

Data abstraction has been approached in a number of ways [Bul02, MU90, SWFM97], and model transformation languages support many of the ideas previously studied. Model transformation tools often provide both a declarative and an imperative component. Declarative transformations are much more concise since they explicitly describe how elements in the source and target models relate to one another. In the previous example (listing 7.2) rule r1 would be considered a declarative rule that "declares" that a bar chart should be created for all hockey players named Wayne Gretzky. The second rule is matched in an imperative manner. The collect statement (line 12) matches all years that Wayne Gretzky played, in order, and invokes the CreateItem rule for each. The second rule could have been

matched in a declarative manner too, however, ATL does not guarantee the order in which declarative rules are matched.

7.4.1 Running Example: Specifying the View

The previous example was straightforward and did not require any significant data processing. In order to demonstrate how data processing can be used in the creation of visualizations, we look at a more complex example. In Section 7.1, three questions were asked about our NHL data (Table 7.1).

The first question asked: **In what years were the most goals scored?** While a simple query could answer the question, a visualization may bring more context and help facilitate a deeper understanding through visual interaction. An example of a visualization that may assist in this case is a treemap. In a treemap, each year could be represented as a box, and nested within each box are players who scored goals that season. Using this approach, it can be seen not only in what years were the most goals scored, but what players lead the league that year. To make use of the treemap visualization to answer our question, a few design considerations must be made: 1) what mappings exist? 2) how do we compute the values? and 3) how will the treemap be structured? To approach the creation of this view, we will decompose the problem into two steps: 1) the mapping of **Player Goals by Year** where a node is created for each Player with the size of the node determined by the number of goals scored by that player, and 2) the grouping of these nodes into a **Year** node.

The first rule (Listing 7.3) shows the declarative mappings that creates a **TreeMapItem** for each **PlayerYearlyStats** element. The label and value of this item are set to the player's name and goals scored respectively. This rule could also be extended to filter all players who did not score a goal, since they are not relevant for this view.

The second rule (Listing 7.4) maps all the **Year** objects and creates a **TreeMapItem** for each. In this transformation, the total number of goals scored that year is computed using the OCL expression on line 5. This particular expression computes the goals scored by all players in that year, and uses the `sum()` operation to add them up. Each of the

```
1  rule r1 {
2          from s : Hockey!PlayerYearlyStats
3          to t : VizModel!TreeMapItem (
4                  label <- s.name,
5                  value <- s.goals
6          )
7  }
```

List of Listings 7.3: Goals to TreeMapItem Transformation

```
1  rule r2 {
2          from  s : Hockey!Year
3          to t : VizModel!TreeMapItem (
4                  label <- s.year,
5                  value <- s.playersStats->collect( e | e.goals )->sum(),
6                  children <- s.playersStats
7          )
8          do {
9                  thisModule.treemapView.children <- t;
10                 t;
11         }
12 }
```

List of Listings 7.4: Year to TreemapItem Transformation

TreeMapItems computed in the previous rule are attached as children by the assignment on line 6.

When executed, this transformation will create a **TreeMapViewer**, with a node for each year, and contained within each year is a node for each player (Figure 7.4). From Figure 7.4, it can be seen that 1992-93 season had the most goals, and if the user expands that node, the players with the most goals that year can be seen.

7.5 Integrating Multiple Views: Snap Points

The visualizations discussed so far are not coordinated with one another. Integrating several views can often lead to more effective visualizations because data can be filtered by first selecting information in one view, e.g. an overview, and narrowing the information space. Pattison describes an architecture which supports multiple coordinated views using the MVC paradigm [PP01]. North describes a method to integrate multiple views using a system called Snap Together Visualizations [NS00]. Snap is applicable to a variety of situa-

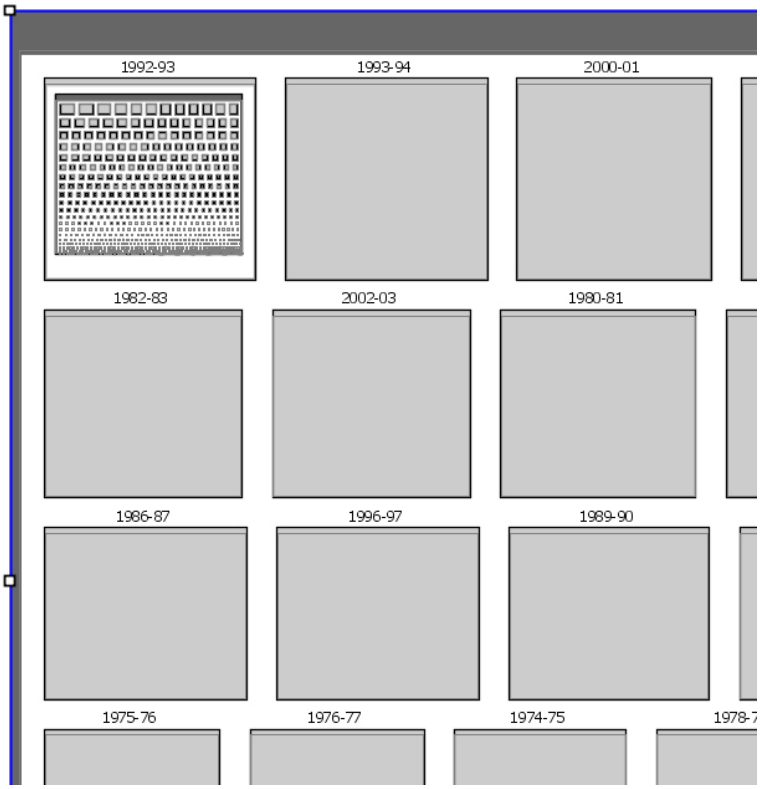

Figure 7.4: Treemap Showing Goals Per Year in the NHL since 1918

tions, however, it cannot be directly applied to MDV. North's work uses relational database queries to generate views and the views are linked through selection and dynamically generated SQL. In order to make use of synchronized views, we have extended his techniques for use in MDV.

Instead of dynamic queries to link views, we provide a mechanism by which interaction designers can annotate models for interconnection. In recognition of the work by North we have called these annotations ***snap points*** [Bul06]. Snap points allow interaction designers to decide where views should be linked by joining the models. For example, in the case of our bar chart view showing only Wayne Gretzky's goals, a listing of all the players could be "snapped" to the bar chart. This snap point indicates that whenever a player is selected in the list, the bar chart should be updated to reflect the selected players statistics.

Figure 7.5 shows the activity diagram for a snap point. Whenever an element is selected, each snap point is resolved. The viewer attached to the snap point is then displayed and seeded with the current selection.

Through MDV, modelers can dynamically create new views by dragging several models and "snapping", or joining, them together. Once the new view coordination model has been designed, transformation rules can be used to specify the data to display.

7.5.1 Running Example: Linking Multiple Views

Questions 2 and 3 in our list of hockey questions can be answered by looking at details of a particular player. Question 2 asks **Given a player, what teams did he play for throughout his career** and question 3 asks **Given a player, how many points did score each year of his career**? In Section 7.4 we designed a view that answered question 3 for a single player, however, we need a more general solution.

To address questions 2 and 3, we will integrate a map and bar chart with a list of players. The bar chart shows the number of points a player scored throughout his career and the map shows where a player played. The map and the bar chart will be snapped to an item in the list, such than when a player is selected, the map and bar chart are populated

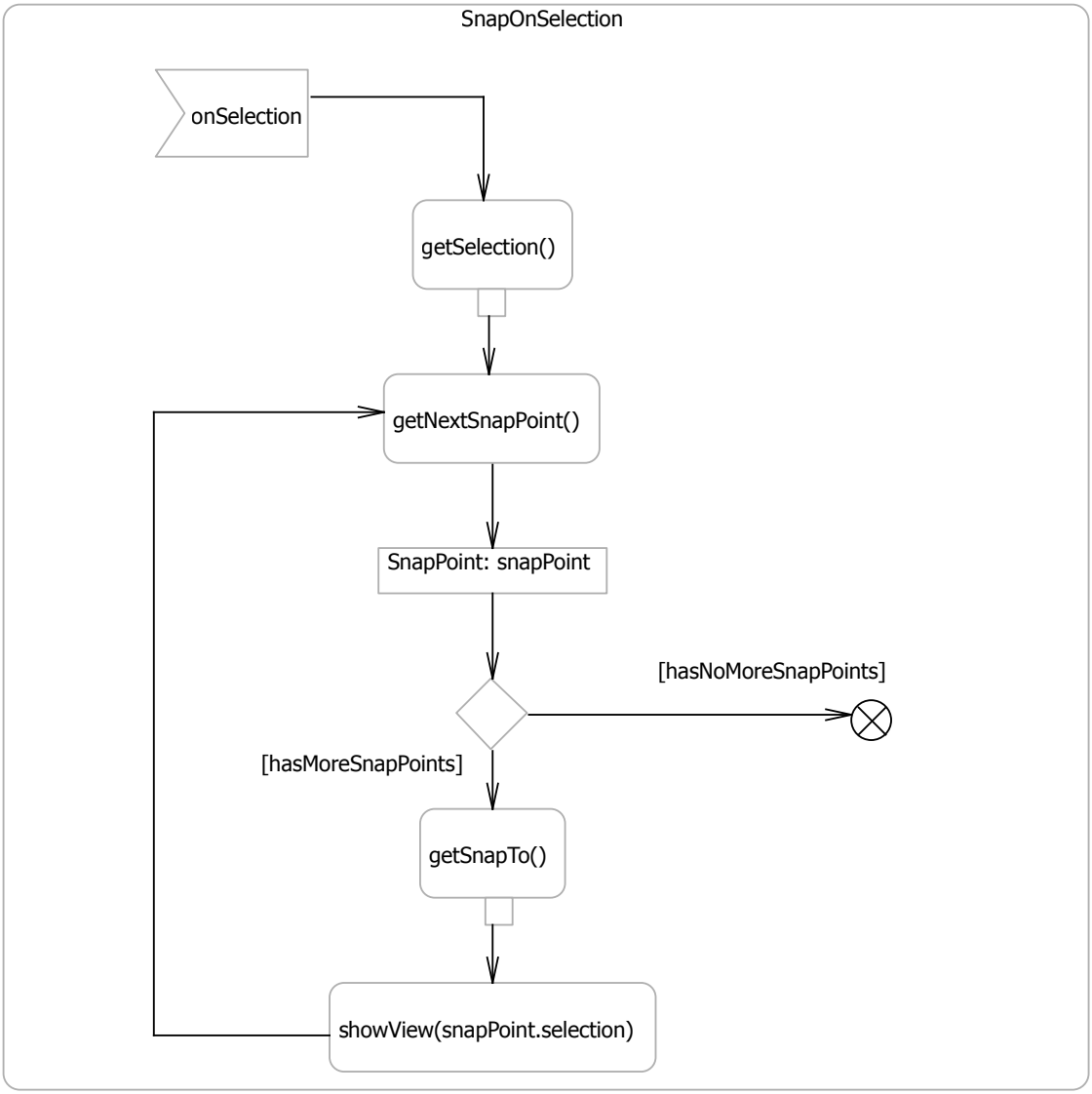

Figure 7.5: Snap Point Activity Diagram

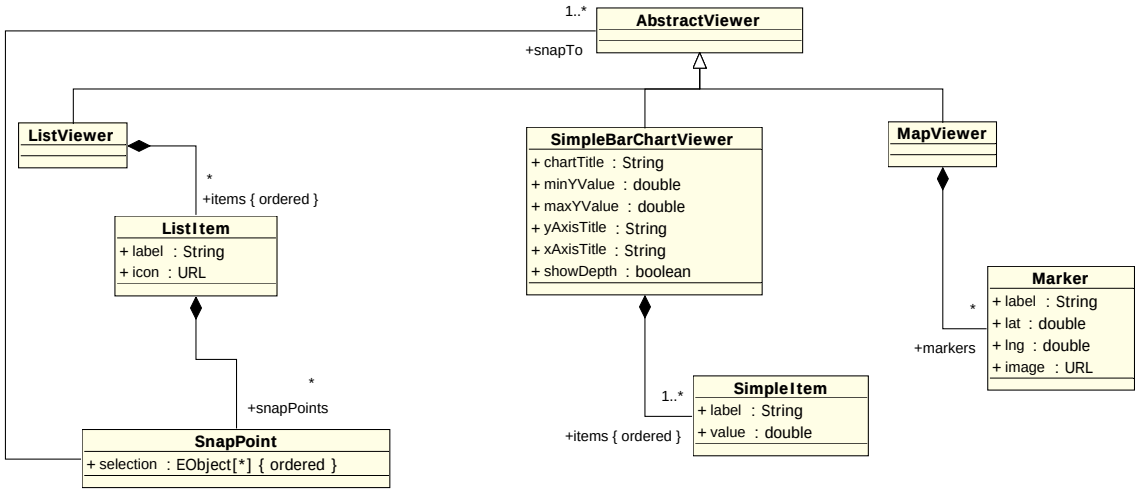

Figure 7.6: Snap Point Model

with information about the player. The model in Figure 7.6 shows how a snap point can be added to the list viewer.

Once the snap point has been added, the customized view can be designed using three transformation rules. The first rule creates a **ListItem**, **BarChart** and a **MapViewer** for each player (Listing 7.5). The list item and chart are both configured to show the player's name, while markers are added to the map and items to the chart. The imperative section of the rule creates the snap point between the list item and the chart and map views.

Markers and bar chart items are mapped from hockey teams and player stats respectively. **BarItems** are created for each **PlayerYearlyStats** object and the value is the sum of goals and assists (line 2 Listing 7.6). A **Map Marker** is created for each team a player played on. After seeding this with a collection of city information (latitude, longitude and team icon) and executing the transformation, the visualization in Figure 7.7 is produced.

The vertical column on the left shows a filtered list with all the players (Figure 7.7-A). When a player is selected, both the map and bar chart (Figure 7.7-B & C) are displayed.

This example shows how multiple views can be integrated within the MDV framework. Views are linked by selection in this case. Other interaction techniques, such as brushing and linking, overview windows and filtering will be the focus of future work.

```
rule Rule1 {
  from s : Hockey!Player
  to   t : VizModel!ListItem (
    label <- s.name
  ),
  b : VizModel!SimpleBarChartViewer (
    items <- s.yearlyStats,
    chartTitle <- s.name +'\'s Points'
  ),
  m : VizModel!MapViewer (
    markers <- s.yearlyStats->
                collect( e | e.teams )->
                collect( f | thisModule.Marker( f ) )
        )
  do {
    thisModule.listView.items <-t;
    t.snapPoint <- thisModule.Snap(
                Tuple{ reference =b, selection = OclUndefined });
    t.snapPoint <- thisModule.Snap(
                Tuple{reference = m, selection = OclUndefined} );
  }
}
```

List of Listings 7.5: ATL Transformation for Integrated View: Rule1

```
lazy rule Marker {
        from s : Hockey!Team
        to t : VizModel!MapMarker (
                label <- s.name
        )
}
rule BarItem {
        from s : Hockey!PlayerYearlyStats
        to t : VizModel!BarItem (
                label <- s.year.year,
                value <- s.goals + s.assists
        )
}
```

List of Listings 7.6: ATL Transformation for Integrated View: Markers and Items

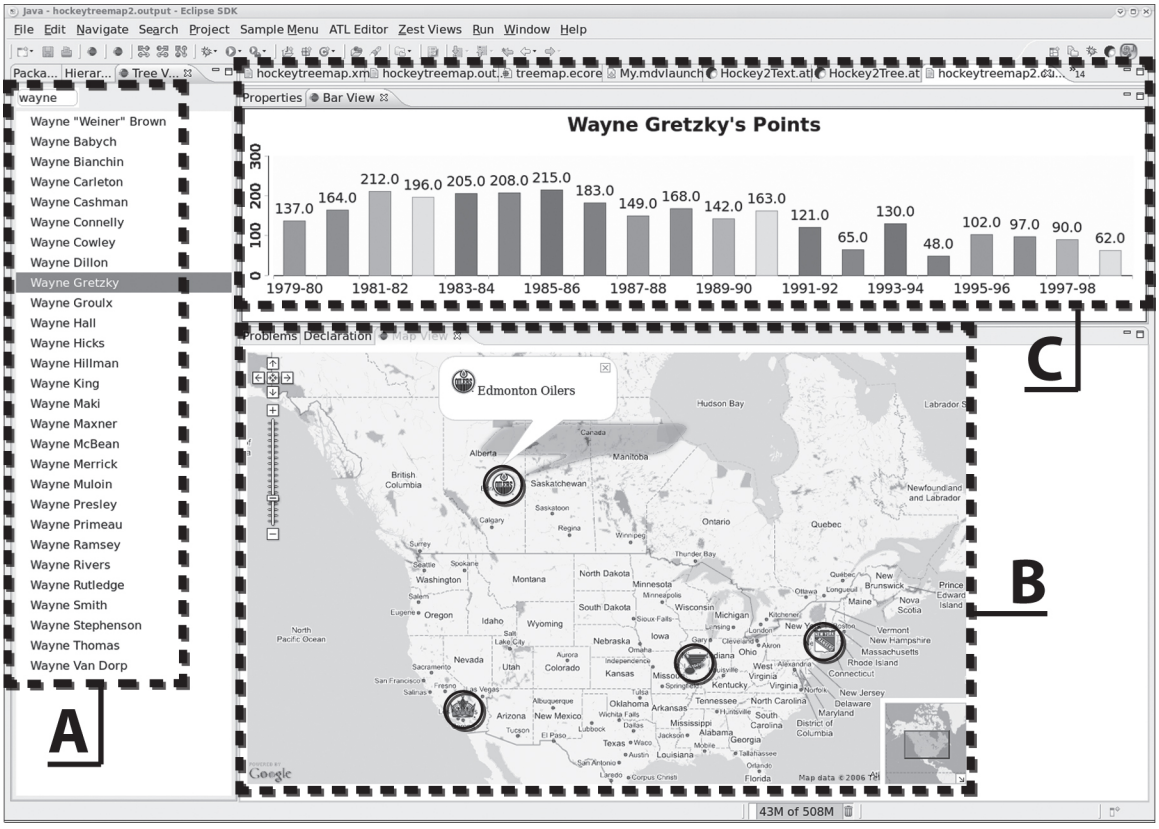

Figure 7.7: Several Coordinated Views of NHL Statistics

7.6 Tool Support

Specifying software transformations is still not a mainstream activity, but with maturing technologies and improved education, developers should become familiar with this engineering technique. There are currently a few basic tools to help design and develop transformations. The Atlas transformation language includes an Eclipse plug-in known as the ATL Development Tools (ADT) [AI04]. ADT includes an editor, outline window and transformation debugger. The editor supports syntax highlighting and on-the-fly compilation with syntax checking and error notification. While the ADT tools provide a solid foundation from which transformation writers can design and implement their rules, there are some features lacking from the system, namely static type checking of the models and content-assist.

In contrast, Open Architecture Ware's (oAW) [Voe] transformation toolkit, xTend, supports content-assist and statically checked models, however, this toolkit lacks debugging compatibilities.

Graphical model mapping, in which users select elements in a source model and assign mappings to elements in a target model are starting to appear [FS07]. These tools can make it easier for designers to see what mappings are possible since both models are presented together.

While tools to help design transformations are maturing, none of the tools integrate the rule writing with transformation execution, meaning that a designer has to explicitly launch the transformation engine and manage the output. To address this issue, we designed a graphical mapping tool that integrates with our visualization environment to assist users when designing views. Our tool presents the data model as a diagram showing the elements, attributes and relationships in a similar manner to a UML class diagram. A list of view models is also shown so the customizer can choose how the data will be rendered. When a view type is chosen, the view model is displayed, and view elements can be "dragged" onto model elements. When completed, the transformation rules are generated and the view is rendered.

Our tool is still in its infancy and was developed to help us better understand the problem of tool development for model mapping. Studies to uncover requirements, improve the design and evaluate the effectiveness of these tools, are important avenues for future research.

7.7 Discussion

Model driven visualization provides model driven engineers with the tools and technologies to integrate interactive visualizations in their systems. By separating the customization and configuration of the view from its underlying model, engineers can explicitly state how their data should be displayed. This formalism provides a number of benefits including:

1) detailed documentation of how the view was constructed, 2) consolidation of all view customizations into a single place, and 3) the opportunity to design model checkers and other tool support for view designers.

Part III

The Evaluation

CHAPTER 8

Evaluation

THE goal of this research is to develop a model driven approach to the design and customization of information visualizations that meets the goals of model driven engineering. In the previous seven chapters, we demonstrated the possibility of creating customized visualizations using a model based approach, but we have yet to present any insight into the practicality of using this technique nor have we discussed when this approach is appropriate. To substantiate our claim that designing and customizing visualizations through a model driven engineering approach is a viable option for tool designers, we revisit the four objectives of this work:

1. Develop a technique for formally modeling visualizations;
2. Develop tool support for our technique;
3. Show how the approach can be used to create key visualizations; and
4. Evaluate our approach against the goals of MDE.

To ensure that we have met these objectives, we evaluate MDV in three ways. In Section 8.1, we discuss how our technique for formally modeling visualizations meets the nine requirements for a visualization customization environment and we describe the tool support that we have developed (objectives 1 and 2). In Section 8.2, we show how MDV can be used to replicate and customize exemplar visualizations (objective 3). We demonstrate this by replicating two of the concrete visualizations from Chapter 4 and further extending and customizing one of them. In Section 8.3, we argue how MDV satisfies the five criteria of a good model (objective 4). In addition to the five criteria, we present some model driven engineering success stories and outline the advantages modeling brings to the process of software development (Section 8.4). Finally, we discuss the limitations of this work (Section 8.5).

8.1 MDV: A Visualization Customization Environment

The first two objectives of this work are to: 1) develop a technique for formally modeling visualizations, and 2) develop tool support for this technique. In Chapter 7, we demonstrated the feasibility of using models to create and customize visualizations, however, for MDV to succeed, it should meet both the functional requirements and design recommendations of a visualization customization environment. These requirements were first discussed in Section 4.5.

8.1.1 Functional Requirements

In Chapters 3 and 4 we explain that three aspects of software can be customized: data, presentation and control, and these aspects can be considered the functional requirements for a visualization customization environment. We also hypothesize that an MDE approach to view creation could meet these requirements. MDE meets these criteria as follows:

1. **Data Customization:** Transformation languages support three types of transforma-

tions, 1) non-ambiguous mapping, 2) selection, and 3) approximation. All three of these transformation types are relevant for customizing data in a visualization [Met05].

The first transformation type, non-ambiguous mapping, associates data items directly to view elements. An example of this type of transformation is the rendering of a node to represent a customer in a customer management system. In this case, each node represents a single customer and each customer is represented by exactly one node. Non-ambiguous mappings are considered one-to-one.

Selection, the second transformation type, maps one or more data elements to view elements only after additional decisions have been made. Filtering is an example of this type of transformation, as only elements that match a particular criterion are shown.

Finally, approximation is the most complex transformation type as there is no semantically equivalent counterparts in the source and target model. In this case, elements in the target have to be approximated and a formal description of how the approximated elements are derived, must be specified. Imperative or operational techniques are often used for this purpose.

The majority of visualizations we analyzed depend on the non-ambiguous mapping and selection transformations. This means that an engineer can often specify a visualization using a series of declarative rules. However, in more complex cases, advanced transformation languages (such as ATL and QVT) support imperative rules to handle data deriving operations.

2. **Presentation Customization:** In MDV, the selection of visual attributes is also performed through a series of model transformations. Most of the visual attributes are a non-ambiguous mapping from data elements to view properties. For example, labels and colours are often chosen to represent particular characteristics of the underlying data.

   The use of models to customize the appearance of a visualization assumes that all

visual attributes can be parameterized. The upfront cost of analyzing and parameterizing the customizable attributes may be infeasible in some systems.

3. **Control / Behaviour Customization:** While MDV is currently limited only to the configuration of a few actions, our approach demonstrates the possibility of using models to configure the control and behaviour of a visualization.

   For example, we demonstrate how multiple views can be linked using SnapPoints (Section 7.5). This technique shows how activity models and model transformations can be combined to create an interactive visualization. While it is feasible to model behaviour in this manner, tool support to generate platform specific interactions from the formal models is still lacking.

8.1.2 Design Recommendations

Three of the design recommendations for a visualization customization environment (efficient view specification, familiar language or notation, and explicit view specification) constrain the functional requirements discussed above. The remaining three recommendations (integrated tool support, provision of view model adapters and use of existing viewers when available) are implementation specific.

1. **Efficient view specification:** Using declarative transformations, designers can map data elements to view elements. For more advanced customizations, transformation languages support an imperative syntax providing the same functionality that general purpose programming languages do. By using declarative rules to map data to view items, designers can quickly prototype new views for testing and evaluation purposes. Using a declarative approach, designers are only responsible for specifying "mapping" rules to describe how the data should be displayed. This is in contrast to a general purpose programming language, with which a developer would have to manually iterate through the data structures, select the desired elements, create the visual artifacts and build the visual model.

2. **Integrated tool support:** Integrated tool support for the creation of highly customized visualizations is important for MDV's success. By implementing our framework using standard tools and technologies, we are able to leverage both existing and future model driven development tools.

   While many transformation languages come with a basic suite of development tools for text based transformations, more research is required to design effective tool support to assist developers while modeling applications, implementing constraints and designing transformations. We have begun preliminary work in this area, including the design of a graphical mapping tool. Future research will not only lead to more advanced visualization customization environments, but to more effective model driven engineering tools in general.

3. **Familiar language or notation:** The customization of data, presentation and control / behaviour are specified in languages standardized by the OMG. By leveraging standard technologies, we propose that model driven visualization should easily fit within existing MDE work-practices and integrate well with both current and future modeling tools.

4. **Provision of view model adapters:** Using our generative programming template, an MVC framework can be generated for any platform independent view model. The templates generate content providers in Java, but other languages such as C♯ could also be targeted.

5. **Use of existing viewers when available:** Model based platform specific viewers have been created for SWT. In particular, we used SWT in order to create platform specific instances for 10 of the 12 visualization techniques presented in Chapter 5 and Appendix A. Using the generated model adapters, viewers for other platforms such as Java/Swing or Microsoft .NET, can be built and added to MDV with relative ease.

6. **Explicit view specification:** By designing visualizations through a series of trans-

formations, the specification of the view is decoupled from the view itself. This explicit specification provides the facilities to automatically generate documentation and enhanced tool support, making it easier for future researchers and tool designers to understand what customizations were performed and which parameters were used.

Currently, the specification of data and presentation can be performed using a transformational approach. Control / behaviour can be specified using a combination of transformation tools and activity models. MDV in its current form does not support the automatic transformation of activity diagrams to code, a possible area for future work.

MDV provides an approach to the creation and customization of information visualization that meets each of the functional requirements and design recommendations listed above. Our current implementation helps solidify the degree to which MDV addresses these requirements and highlights areas for future research. Table 8.1 outlines how MDV addresses each of our requirements and what tool support is currently available.

8.2 Creating Exemplar Visualizations

The third objective set out for this research was to show how MDV can be used to create exemplar visualizations. To meet this objective we show how two of the visualizations studied in Chapter 4 can be replicated using MDV. The two views we replicated are: 1) the Jambalaya member-area view, and 2) the plug-in dependency view. We chose the Jambalaya view because we had access to the original data, therefore we could verify that our approach worked as expected. We also demonstrate how this view can be further extended and customized. Finally, the plug-in dependency view was chosen because it was the most complex view we studied.

Requirement	Using MDV	MDV Tool Support Available
Efficient view specification		
Data Customization	Data processing is performed in a concise, declarative syntax.	✓
Presentation Customization	Visual attributes are specified as mappings in the transformation rules.	✓
Control / Behaviour Customization	Activity diagrams and OCL are used to model interactions.	
Integrated tool support	The rich semantics of models more easily facilitates the automatic generation of tools.	
Familiar Language		
Data Customization	ATL & QVT are standard data processing languages for model driven software engineers.	✓
Presentation Customization	The customizable attributes are defined through UML selected using OCL.	✓
Control / Behaviour Customization	OCL, UML Activity diagrams and transformation languages are used for specifying control / behaviour.	✓
Provide view model adapters	The template in Appendix B can be used to generate content providers for any visualization pattern described using EMF.	✓
Use existing viewers when available	Common SWT and Eclipse RCP viewers have been leveraged for 10 of our 12 visualization patterns.	✓
Explicit view specification		
Data Customization	Data processing operations are explicitly stated as transformation rules.	✓
Presentation Customization	Visual attributes are defined explicitly using UML and specifically chosen as part of the transformation.	✓
Control / Behaviour Customization	Interactions are explicitly stated as UML Activity diagrams.	

Table 8.1: Summary of how MDV supports the creation and customization of visualizations.

8.2.1 Jambalaya View

The first view we chose to re-design is one of Allen's [All03] SHriMPBib views. Allen originally used Jambalaya to create a node-link view to display the relationship between researchers and their areas of expertise. Allen called this visualization the **member-area view**.

To re-create this view, we first designed the SHriMPBib model using model driven engineering technologies. We then mined the SHriMPBib data so we could create the same visualizations created by Allen. With both the model and instance-data, the view was re-created as follows. First, the node-link viewer was chosen as this was the same technique chosen by Allen. Second, a series of transformation rules were designed in order to generate the view (Listing 8.1). Finally, the transformation was executed to generate the view.

We now describe the transformation rules (Listing 8.1) in terms of data, presentation and control / behaviour.

Data Customization

Three rules are needed to customize the data for Allen's member-area view in order to create a node-link diagram that shows the relationships between readers and their research area. The first rule creates a graph node for each user in the system. In addition to the graph nodes, this rule also creates an edge from each user to the documents they read. The second rule maps documents to graph nodes and creates an edge from each document to the research areas it is tagged with. The final rule creates a node for each research area. When these three rules are executed on Allen's instance data, the initial graph is created.

Presentation Customization

Once the graph is created, the visual attributes can also be specified through declarative transformation rules, however in this case, no new rules are needed. Instead, the label and

```
1  rule User2Node {
2          from s: shrimpbib!User
3          to t: vizmodel!GraphNode (
4                  label <- s.nick
5          ),
6          edges: distinct vizmodel!GraphEdge foreach ( e in s.documents ) (
7                  src <- s,
8                  dest <- e
9          )
10 }
11 rule Area2Node {
12         from s: shrimpbib!Area
13         to t:vizmodel!GraphNode (
14                 label <- s.name
15         )
16 }
17 rule Document2Node {
18         from s: shrimpbib!Document
19         to t: vizmodel!GraphNode (
20                 label <- s.title,
21                 expandOnMouseOver <- true,
22                 hideText <- true
23         ),
24         edges : distinct vizmodel!GraphEdge foreach( e in s.area ) (
25                 src <- s,
26                 dest <- e,
27                 colour <-#Red
28         )
29 }
```

List of Listings 8.1: Re-creating the Member-Area View

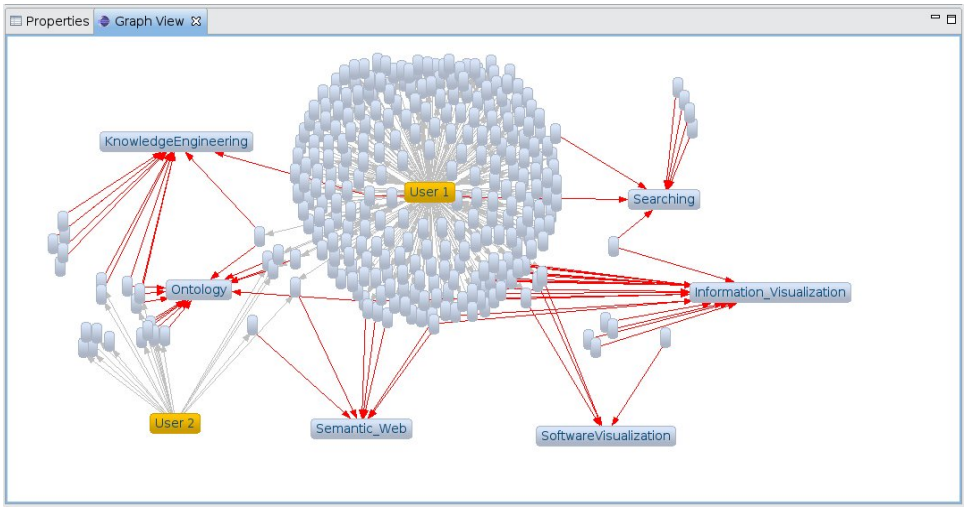

Figure 8.1: MDV: Member Area View

colour for each node and edge are specified by adding additional attribute mappings to the existing rules. In the case of the member-area view, the label names are hidden for the documents as there are over 600 documents listed in the system.

Control / Behaviour

Allen did not customize any control or behaviour for her member-area view. She did specify an initial layout algorithm, which we also specified using a single mapping rule that is executed once (an entry point rule).

Figure 8.1 shows the results of these transformations applied to the anonymized SHriMP-Bib instance data. Allen claims that using this style of visualization, a *"user could discover which group members have expertise in various research areas."* [All03, p.76].

8.2.2 Extending the Member-Area View

While the visualization in Figure 8.1 may help users discover which group members have expertise in various research areas, a simpler view showing only the members and the research areas may be more effective. To demonstrate that model driven visualization can

```
1  rule User2Node {
2          from s: shrimpbib!User
3          using  {
4                  user_areas : Sequence(shrimpbib!Area) = s.documents->
5                                  collect( e | e.area)->flatten();
6                  unique_areas: Set(shrimpbib!Area) = user_areas.asSet();
7          }
8          to t: vizmodel!GraphNode (
9                  label <- s.nick,
10                 colour <-#Orange
11         ),
12         edges2: distinct vizmodel!GraphEdge foreach ( e in unique_areas) (
13                 src <- s,
14                 dest <- e,
15                 label <- user_areas->count( e ).toString()
16         )
17 }
```

List of Listings 8.2: Updated Member Area View Rules

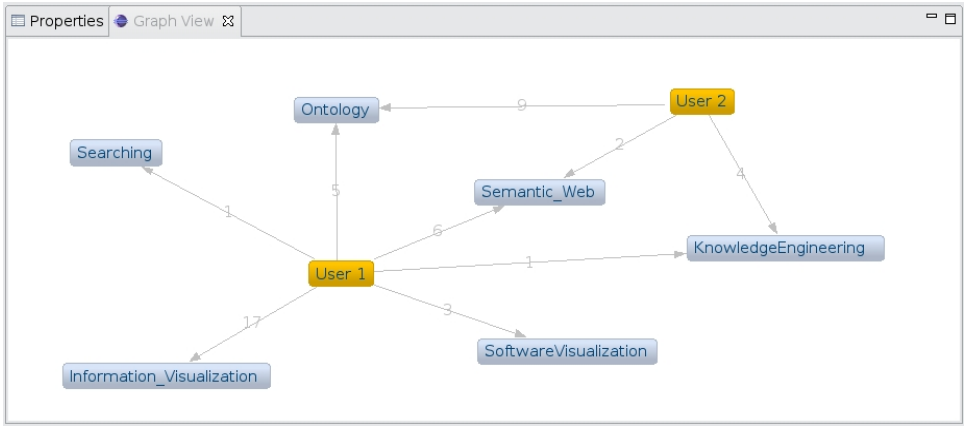

Figure 8.2: Updated Member Area View

also be used as an efficient means of customizing existing views, we extended the view in Figure 8.1 to create a view that directly links researchers to their areas of expertise.

To support this change using MDV, we only needed to change the `User2Node` rule. Instead of creating an edge from each user to their documents, we created an edge from the user to the collection of research areas in which they read papers, and removed the `Document2Node` rule. Listing 8.2 shows the updated `User2Node` rule and Figure 8.2 shows the results.

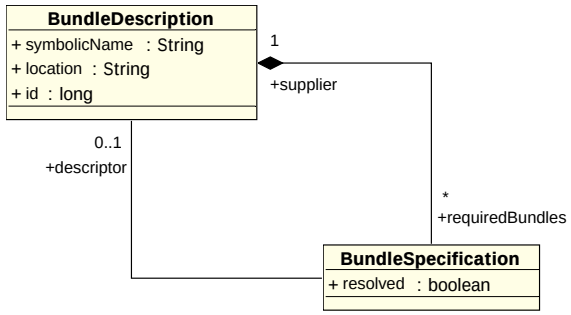

Figure 8.3: Plug-in Model

8.2.3 Plug-in View

The final view chosen for re-design is the plug-in dependency analysis tool, originally discussed in Section 4.4. This view was chosen for three reasons: 1) it possesses all three types of view customizations, 2) an abundance of data are available for testing purposes, and 3) it is a useful view, as there are over 1,000 plug-ins listed on Eclipse Plug-in Central [1]. This view provides a good showcase for the capabilities of MDV. As with the member-area view, we present the creation of the plug-in view in terms of how the data, presentation and control / behaviour are customized.

Data Customization

The data customization for the plug-in view requires both data selection and the derivation of new relationships. The original data-model uses the concept of a "proxy". That is, if plug-in A depends on B, C and D, instead of directly linking to these, plug-in A contains three proxies. Each of these proxies links to the actual plug-in. When the view was designed it was decided that these proxies should not be shown, but instead, a direct connection should be drawn between a plug-in and all dependent plug-ins. Transforming data from this original form (Figure 8.3) to a node link view (Figure 8.4), while removing the proxies and deriving the relationship between the plug-ins, is the first customization we performed.

[1] http://www.eclipseplugincentral.com/

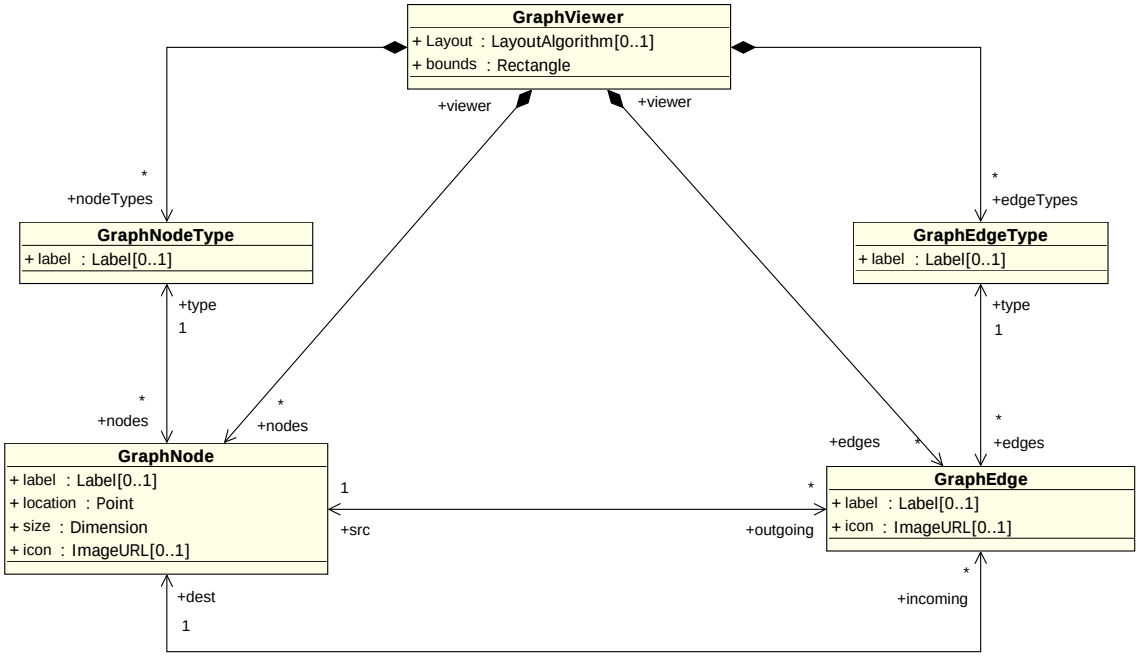

Figure 8.4: Node Link Model

This customization is performed by three transformation rules. The first rule (Rule1 in Listing 8.3) matches the BundleDescription with the name INPUT_PLUGIN and creates a GraphNode. For each required bundle, a GraphEdge is created. The destination of the edge is the BundleDescription for which it is a proxy and the source is the supplier. By matching all plug-ins in the dependency list, a directed-acyclic dependency graph is created. While these rules process the data to create the graph, very little will be displayed as no visual attributes have been defined.

Presentation Customization

To address the fact that the nodes and edges have no visual attributes, colour and label attributes are added to the two rules. The colour of the root node (the one created in Rule1) is set to green, while all other nodes are left as “Default” and the label is set to the bundle name. With these two changes, the graph now has labels and the nodes are coloured appropriately.

```
1  rule Rule1 {
2      from s: pdemodel!BundleDescription (
3        s.symbolicName = INPUT_PLUGIN
4      )
5      to t : vizmodel!GraphNode (
6        outgoing <- s.requiredBundles->collect(e|thisModule.connection(e))
7          )
8      do {
9        thisModule.graphView.nodes <-t;
10     }
11 }
12 lazy rule connection {
13     from s : pdemodel!BundleSpecification
14     to t : vizmodel!GraphEdge (
15       dest <- thisModule.Rule2(s.descriptor)
16     )
17     do {
18       thisModule.graphView.edges <-t;
19     }
20 }
21 unique lazy rule Rule2 {
22     from s: pdemodel!BundleDescription
23     to t : vizmodel!GraphNode (
24       outgoing <- s.requiredBundles->collect( e | thisModule.connection (e) )
25     )
26     do {
27       thisModule.graphView.nodes <-t;
28     }
29 }
```

List of Listings 8.3: Data Customization for Plug-in Visualization

Finally, the layout attribute was set to provide an initial layout algorithm for the view. We chose the directed graph layout since this was the same one used in the PDE viewer we coded by hand.

Control / Behaviour Customization

There are a number of actions available in the plug-in dependency view including: 1) analysis tools to highlight nodes based on a particular criteria, 2) a "focus on node" feature to bring a node into focus when the mouse moves over it, and 3) a screenshot tool.

Each of these behaviours can be modeled, however, behavioral models can become very complex. To demonstrate how behaviour can be modeled, we chose to re-create the interaction that highlights the shortest path between two nodes in the graph. In this case, two nodes are selected and Dijkstra's shortest path algorithm is used to compute the path between these nodes. Finally, this path is highlighted, by updating the visual properties of the affected nodes.

Figure 8.5 shows the activity diagram that describes this, while the transformation that actually performs the shortest path calculation is presented in Appendix D.

8.2.4 Outcome

Using MDV, we have replicated two existing visualizations, and customized and extended one of them. By using MDV we have isolated the configuration and customization code into relatively short transformations. The Jambalaya view is specified using a single transformation of approximately 50 lines and the plug-in view is specified using two transformations, one for customizing the data and presentation (50 line transformation) and one for performing the shortest path analysis and the associated presentation of this path (106 line transformation). The complete ATL code for both these transformations is available in Appendix D.

Table 8.2 summarizes the differences (in number of statements) between the transfor-

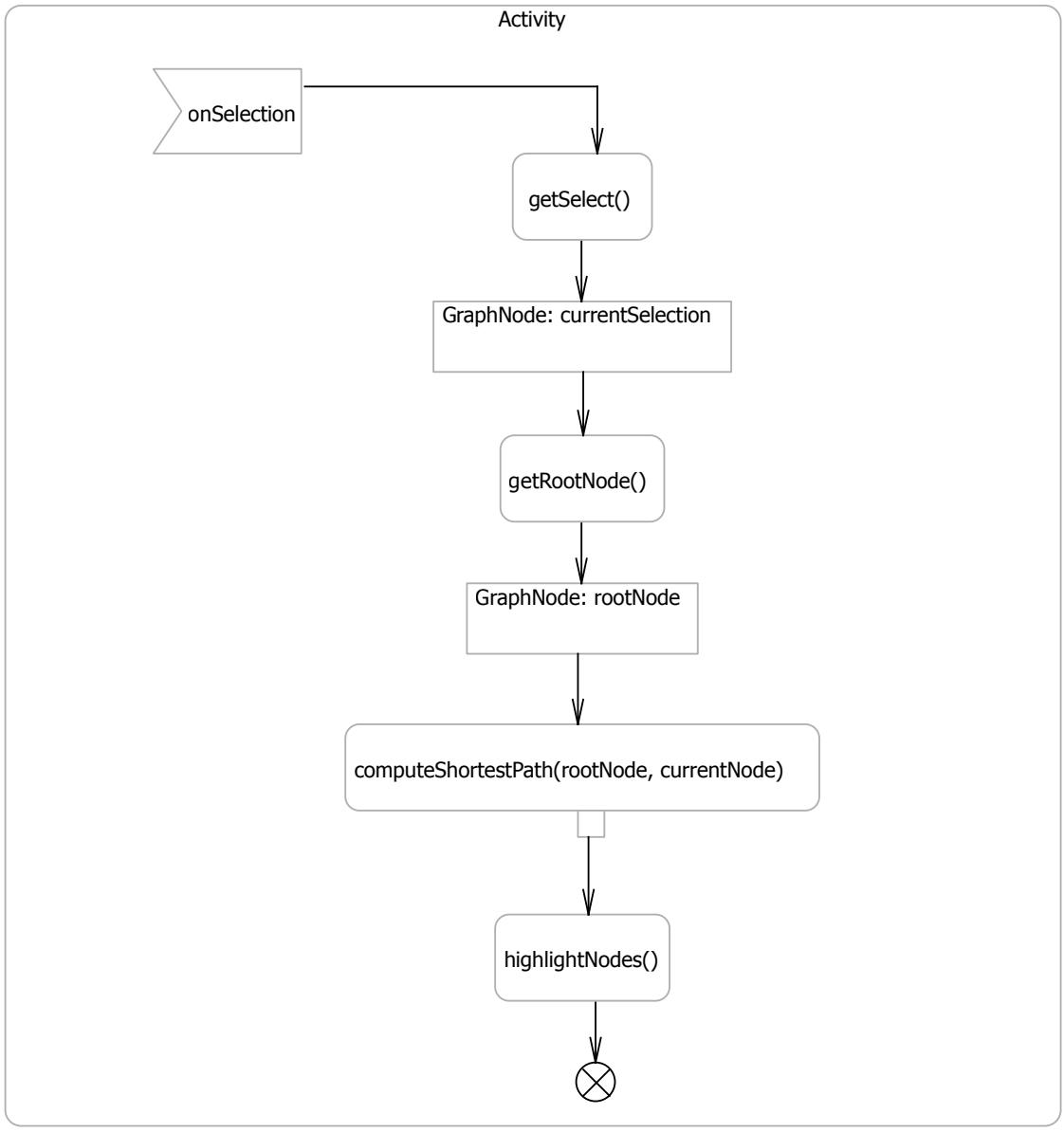

Figure 8.5: Highlight Shortest Path Activity Diagram

	MDV Based Solution	Hand Coded Solution
Member Area View	15	29
Updated Member Area View	12	26
PDEViz	14	21

Table 8.2: MDV Based vs. Hand Coded Solution, Comparison of Statements

```
1  rule Rule1 {
2      from s: pdemodel!BundleDescription (
3        s.symbolicName.startsWith('GUI')
4      )
5      to t : vizmodel!GraphNode
6  }
```

List of Listings 8.4: Declarative Transformation

mation based implementation of the views, and the ones developed using Java.

By recreating existing visualizations, and customizing one of them, we demonstrated that MDV is a practical approach for the creation and customization of information visualizations. To demonstrate that MDV offers an MDE approach to view creation, we next evaluate MDV from the perspective of model driven engineering.

8.3 Validating the Goals of Model Driven Engineering

The final objective of MDV is to support view creation in a manner compatible with the goals of model driven engineering. As Selic explains, a good model embodies the following five characteristics [Sel03]: 1) abstraction, 2) understandability, 3) accuracy, 4) predictiveness, and 5) inexpensiveness. We discuss how MDV fits within each of these subjective dimensions.

Abstraction: Abstraction refers to a model's ability to reduce the details of the system it represents. The declarative nature of a transformation language makes understanding mappings much easier than it would be in a high level programming language. To illustrate this, let us compare the declarative transformation code in Listing 8.4 to the same functionality written in a high-level programming language (Listing 8.5). In Listing 8.5, the developer is responsible for iterating through all the model elements, filtering and selecting objects, and explicitly creating and attaching the view model elements. Using a declarative transformation language (Listing 8.4) this can be expressed much more concisely, as an engineer needs only to specify the **from** and **to** targets.

Understandability: Using MDV, the design and customization of a view has been com-

```
1  PDEModel model = getModel();
2
3  GraphModel graphModel = getGraphModel();
4  List descriptions = model.getBundleDescriptions();
5  Iterator descriptionIterator = descriptions.iterator();
6
7  while(descriptionIterator.hasNext()) {
8      BundleDescription description = descriptionIterator.next();
9      if ( description.getSymbolicName().startsWith('GUI') ) {
10         GraphNode node = new GraphNode();
11         graphModel.addNodeToModel(node);
12     }
13 }
```

List of Listings 8.5: Transformation Written in Java

pletely decoupled from the data and viewer specification. By separating these components, the view specifications is not littered with unnecessary complexities and the designer can see (often from a class diagram or a few transformation rules) how the view is designed.

Accuracy: The models provide more accuracy and make the views easier to reproduce than if the views were designed using a traditional approach. This is achieved as the model explicitly states how a visualization is created, which attributes are selected and how edge cases are handled. Instead of describing a view through a screenshot and ambiguous prose, MDV makes the design decisions explicit.

Predictiveness: Since models explicitly describe the semantics, they are easier to validate. For example, constraints such as the number of nodes that can be realistically rendered in a node link diagram, or the number of bars that can be effectively displayed in a chart, can be specified on the model. Once specified, these constraints can be tested before the view is designed. This is similar to the types of testing which physical models are subjected to in traditional engineering disciplines.

Inexpensiveness: Analyzing the "cost" of a model is highly subjective, however, it is clear that when a formal model that represents a visualization is used, designers are able to more easily experiment with ideas and quickly make changes. In our own experience, we have been able to construct views in a matter of minutes, that before, would have taken a day or two to complete. Also, by sharing models and transformation rules, designers can begin to mix and match components in order to design a completely customized visualization.

8.4 Modeling in Practice

As software systems continue to evolve, and more complex, interconnected systems are sought, managing the planning, development and maintenance of these systems will continue to challenge software engineers. While software development is an inherently complicated process [FPB87], a model driven approach can help alleviate some of the accident complexity.

Model driven engineering techniques have successfully been applied in some of the world's most challenging domains. Motorola has claimed that "*Model-driven engineering has dramatically increased both the quality and reliability of software developed in our organization, as well as the productivity of our software engineers* [WW06]". In contrast to conventional software development, Motorola believes that the formalisms enabled by MDE, mitigates ambiguities during requirements engineering, removes the highly error-prone process of manually translating design documents into code, and maintains better synchronization between the code and the design documents [WW06].

The UK National Health Service Information Authority (NHSIA) leverages modeling technologies because of the platform independent capabilities of MDE. Due to the volatile nature of health care requirements, it was important that the NHSIA maintain a single specification. This specification was then used to drive a number of different technologies and architectures [Rai05].

The areospace industry has also made heavy use of modeling in many mission critical software systems. Some of the reasons cited for the use of modeling include, lower maintenance costs, improved stakeholder insight, requirements traceability, rapid prototype development and better component reuse [IS05]. While the areospace industry is very careful about publishing internal practices, they have claimed to have successfully applied model driven engineering techniques in projects ranging from less than 10 team members, to projects with over 100 members working with 1000's objects [IS05].

Many businesses are reluctant to publish their successes with model driven engineer-

ing technologies as they often view their use of MDE as a competitive advantage [Sel06]. However, with industrial track sessions at many popular conferences, and many quality open source modeling projects, the MDE advantage is starting to become known. More engineers are turning to software modeling for reasons such as, a more streamlined development process, component reuse, and a better separation between the technical solution and the specification. MDV will assist these engineers with a highly integrated and relevant approach to the design and customization of information visualizations.

8.5 Limitations

Even though we have designed several visualizations using this approach, model driven visualization is not a silver bullet for view design. There are a few limitations which must be considered before using models to design views.

The first limitation of our approach is that everything must be modeled. That is, a formal model must exist for all views one wishes to create and one must have a model for one's data. Our intended audience is software engineers who are already making use of MDE for their applications, so the data models should already exist. As for the view models, we have provided a small collection of view models that we or others may expand on in the future.

When a view model does not exist, the engineers should consider whether or not the cost of constructing these view models will ever be recovered. Roberts and Johnson argue that before a framework is designed, three instances should first be created [RJ97]. This is likely a good rule of thumb for creating view models.

The second limitation of MDV is that declarative transformation languages are quite different from traditional high level programming languages. Designing solutions through a series of transformations is unlikely something that all software engineers will feel comfortable doing. Again, we would expect that at least one person on a development team would have experience designing software through model transformations.

The third limitation of our work, and MDE in general, is the lack of tool support. Software development is a challenging activity, and while modeling can help alleviate some of this difficultly, there is still not a good selection of tools that provides cognitive support to software modelers.

Finally, as discussed in Chapter 8, modeling the control / behavioural aspects of a view is still an understudied area. This will be the primary focus of our future work.

CHAPTER 9

Conclusions

WITH open-membership organizations such as the OMG defining modeling standards, and open-source implementations of those standards emerging, model driven engineering is becoming a viable option for any software engineer. To ensure that modeling can benefit all aspects of software, researchers have studied ways to apply model driven engineering techniques across the different components of a software system. To further the advancements of model driven technology, we have demonstrated the power and feasibility of a model based customization environment for information visualizations. A number of benefits of model driven visualization have been discussed, including better documentation, automatic generation of model adapters, improved tool support enabled through richer semantics, and a concise format for specifying simple views with the power of an imperative language for more complex interactions.

We have successfully applied model driven visualization to construct a number of new views. In addition, we have replicated existing visualizations that were originally designed

using high-level programming languages and specialized tools. Using our approach, not only are the view customizations more concise, they have been explicitly stated and successfully decoupled from the implementation details. This decoupling improves design, maintenance and evolution for the following reasons:

1. **Formally specified views:** All of the customization options and design choices are now formally specified, making it easier for tool designers and future researchers to reproduce successful visualizations. Specifying views in this manner will allow the view to more easily evolve as requirements are refined.

2. **Centralized view specification:** The ways in which data, presentation and control / behaviour are customized is now stated explicitly. This will make it easier for those maintaining the software to modify or extend existing visualizations.

3. **Better documentation:** Formal view models provide precise documentation about the types of elements and attributes that can be configured for a particular view. This is in contrast to complex frameworks in which the configurable attributes often become hidden as there is no standard way to present customizable options. By explicitly stating the configurable attributes, software engineers can more easily understand how a visualization technique can be customized.

4. **Lowers a barrier to adoption:** Many visualization techniques are first designed for a single purpose. By describing a view in terms of its components and visual attributes, other designers are able to hypothesize about how the view may fit their data model. Then by writing mapping rules, designers can generate a prototype and test their hypothesis. This means that software engineers can rapidly prototype and test their tools, tightening the feedback loop in user-centered design.

9.1 Future Work

While we have successfully created a number of interactive visualizations using a model driven engineering approach, this work only scratches the surface of possibilities. Our work has focused on the static data structures that constitute the specification of a view model, but languages, such as UML, support a much broader scope of modeling options. For example, the use of interaction diagrams could potentially play a key role in the definition and customization of view control and behaviour.

As is often the case, this research poses more questions then it answers. We will present some of the avenues that future researchers can focus on, and offer guidance to those building on this work.

9.1.1 Study the evaluation of API design for a graphing toolkit

One of the goals of Zest is to define an easy to use graphing Application Programmer Interface (API). We attempted to reach this goal by mimicking the SWT API, the toolkit for which Zest is intended. However, in our work we did not evaluate our success in this area, and our only measure is the kind words we receive from users of our toolkit. An important area for future research is a better evaluation of the design of a graphing API. Much work has been done evaluating the ease of use of a user interface, but little work has actually focused on the ease of use of an API.

9.1.2 MDE Tool Support

An important part of any framework is the tool support available. Effective tool support can greatly improve productivity and software engineers are beginning to demand high quality tools. In order to help create and customize visualizations, we need to provide more than a modeling process, we also must supply tools that provide cognitive assistance to the engineers. Some examples include: better mapping tools for designing transformations, better tools for selecting models, and tools to evaluate the model mappings.

9.1.3 Generating Interactivity Through UML Activity Diagrams

While we have modeled the interactivity of a visualization using UML activity diagrams, this has only been performed for documentation purposes. Determining the feasibility of using existing code generation technologies to generate these interactions remains a future research topic.

9.1.4 Other ways to integrate multiple views

In addition to improving the modeling of interactivity, we intend to explore other ways in which multiple views can be integrated. In Chapter 7, we showed how multiple views can be "snapped" together, but there are a number of user interaction techniques such as brushing and linking, filtering, and overview + detail, which MDV could also support. Finding an approach that supports the modeling of different interaction techniques that make use of multiple views, would be a useful addition to this work.

9.1.5 Finer Granularity for View Models

Another limitation of this work is that the view models must be specified before a visualization can be generated, and the upfront cost of developing these models may never be recovered. However, many of the view models we specified have a number of common components that can likely be abstracted. If properly abstracted, the view designer could use these components to first assemble a platform independent view model, and then use this model to create a platform specific instance.

One possibility for specifying the relationships between components is through the use of feature models [CHE05]. "Feature modeling is a notation and an approach for modeling commonality and variability in product families [CW07]." Feature models could be used to specify the mandatory / optional features, groups of features, and implies and excludes relationships for the collection of elements that make up a visualization. Using these feature models, a designer could quickly combine the elements in a meaningful way, and if the

propositional formula generated from the feature maps holds true, a platform independent view model could be generated.

9.1.6 Improved MDE Education

Finally, we believe we can leverage our experience with model driven visualization to provide better education for software modeling at the university level. Many industry leaders, including IBM and Microsoft, are embracing MDE as an approach to software development. If this trend continues, it will be important that software engineering curriculums contain a model based component. This curriculum should include course work on generative programming, model transformations, model constraints, model based data design, model based UI design and of course, model driven visualization.

9.2 Contributions

Several contributions have emerged as a result of this work. These results impact both the information visualization and model driven engineering communities. In addition, this research directly contributes to the software engineering community as a whole.

9.2.1 Contributions to the Information Visualization Community

Model driven visualization has made a number of important contributions to the field of information visualization. First, the taxonomy and collection of view models provides a framework in which other view designers can place their work, making information visualization techniques and tools available to a wide variety of domains. By separating the view from the data it displays, and documenting the techniques as a set of patterns, software engineers working in a variety of domains can make use of these techniques.

Secondly, the use of formal models within the specification of a visualization provides future designers, and those who customize visualizations, a powerful language they can use

to share their results. Recreating views, even with access to the original developers, proved to be a challenge for a number of reasons: 1) the customization options were often hidden in source code, 2) the visualization tools were not available, leaving only ambiguous prose and a single screen-shot, and 3) the design decisions, including failed attempts to create the view, were never documented. Using a formal specification for view customization means that visualizations can be more easily evolved and the views re-created.

Finally, a general purpose graph based visualization toolkit has been designed for Eclipse. While other toolkits do exist for Java based applications, we believe that by providing a toolkit that uses the same design patterns, and a familiar API, should help improve the adoption of more interactive visualizations. Furthermore, including our visualization toolkit as part of the yearly Eclipse release means that Zest is not just a research prototype, but a toolkit readily available to all Eclipse users.

9.2.2 Contributions to the Model Driven Engineering Community

In addition to the information visualization community, this work contributes to model driven engineers. Model driven visualization presents another example of how MDE can be leveraged in the design and implementation of an important aspect of today's software, interactive visualizations. Model driven engineers are designing models for different aspects of software systems, but until now, models of interactive visualizations have been largely ignored.

In particular, formal, platform independent view models, help validate the power of modeling as these models demonstrate that formalisms can be applied to the design of information visualizations. UML class diagrams and OCL constraints provide a concise way to describe the types of data, and the customizable properties of a view model.

In addition to helping validate the power of modeling, this work has helped drive requirements, helped identify defects, and provided several tests cases for a number of MDE related areas. In particular, this research has directly contributed code to the Atlas Transformation Language (ATL), the Eclipse Modeling Framework (EMF) and Eclipse's imple-

mentation of the Object Constraint Language (OCL).

Finally, this work presents a code generation technique for automatic creation of an MVC framework and set of model adapters. As more models are designed, developers are constantly faced with the integration and synchronization of multiple models. While transformation languages help define these integration points, the reality is that not all systems have a properly defined model in a language such as UML, Ecore or MOF. Our generative approach helps software engineers adapt their existing Java classes to work with a formal model.

9.2.3 Contributions to the Software Engineering Community

Finally, model driven visualization contributes to the software engineering community as a whole. In particular, MDV underscores the importance of separating concerns in a software system. The separation of concerns is not only an effective way to help software maintainers, it also defines sound component boundaries in which researchers and developers can work.

Model driven visualization makes these boundaries precise for the domain of information visualization. Using MDV, and solid software engineering principles, researchers can now conduct research as to the effectiveness of individual view models, and document their findings as formal constraints. By formally capturing knowledge about visualizations in a platform independent manner, the power of information visualization will more easily stand the test of time.

References

[Ada06] Eytan Adar. GUESS: A Language and Interface for Graph Exploration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 791–800, Montreal, Quebec, Canada, 2006. 42

[AI04] Freddy Allilaire and Tarik Idrissi. ADT: Eclipse Development Tools for ATL. In *Proceedings of the Second European Workshop on Model Driven Architecture (MDA) with an Emphasis on Methodologies and Transformations (EWMDA-2)*, pages 171–178, Canterbury, England, 2004. 127

[AIwMJ+77] Christopher Alexander, Sara Ishikawa, Murray Silverstein with Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language: Towns, Buildings, Construction*. Oxford, 1977. 81

[AK03] Colin Atkinson and Thomas Kühne. Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5):36–41, October 2003. 20

[Ala03] Harith Alani. TGVizTab: An Ontology Visualization Extension for Protege. In *Proceedings of Knowledge Capture 2003 - Workshop on Visualizing Information in Knowledge Engineering*, Sanibel Island, FL, 2003. 32

[Alf] Alfred Inselberg. Home of parallel coordinates. http://www.math.tau.ac.il/ aiisreal/. 203

[All03] Mary Margaret Allen. Empirical evaluation of a visualization tool for knowledge engineering. Master's thesis, University of Victoria, 2003. 54, 138, 140

[All08] OSGi Alliance, 2008. http://osgi.org. 70

[AS94] Christopher Ahlberg and Ben Shneiderman. Visual information seeking: tight coupling of dynamic query filters with starfield displays. In *Proceedings of the SIGCHI Conference on Human factors in Computing Systems (CHI'94)*, pages 313–317, New York, NY, USA, 1994. 199

[AWT] AWT Abstract Widget Toolkit. website. http://java.sun.com/products/jdk/awt/. 34

[Bal81] Robert Balzer. Transformational implementation: An example. *IEEE Transactions on Software Engineering*, SE-7(1):3–14, January 1981. 11

[BBS04] R. Ian Bull, Casey Best, and Margaret-Anne Storey. Advanced Widgets for Eclipse. In *Proceedings of the 2nd Eclipse Technology eXchange (ETX'04)*, pages 6–11, 2004. 70, 114

[BC87] Richard A. Becker and William S. Cleveland. Brushing Scatterplots. *Technometrics*, 29(2):127–142, May 1987. 204

[BCT05] Alan W. Brown, Jim Conallen, and Dave Tropeano. Introduction: Models, Modeling, and Model-Driven Architecture (MDA). In *Model-Driven Software Development*, pages 1–16. Springer Berlin Heidelberg, 2005. 15, 112

[Bec99] Kent Beck. *eXtreme Programming eXplained*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999. 18

[Ben01] Ben Shneiderman and Martin Wattenberg. Ordered Treemap Layouts. In *Proceedings of the IEEE Symposium on Information Visualization 2001 (InfoViz'01)*, pages 73–78, 2001. 93

[Ber81] Jacques Bertin. *Graphics and Graphic Information Processing*. Walter de Gruyter, 1981. 32, 37, 200

[BETT94] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry: Theory and Applications*, 4:235–282, 1994. 90, 108

[Béz01] Jean Bézivin. From Object Composition to Model Transformation with the MDA. In *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)*, pages 350–354, 2001. 20

[Béz05] Jean Bézivin. On the Unification Power of Models. *Software and Systems Modeling*, 4(2):171–188, 2005. 1, 17

[BGH00] Ivan T. Bowman, Michael W. Godfrey, and Richard C. Holt. Connecting Architecture Reconstruction Frameworks. *Information and Software Technology*, 42(2):91–102, 2000. 61

[BGH+05] Sven Burmester, Holger Giese, Martin Hirsch, Daniela Schilling, and Matthias Tichy. The Fujaba Real-Time Tool Suite: Model-Driven Development of Safety-Critical, Real-Time Systems. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, pages 670–671, May 2005. 31

[BH02] Luciano Baresi and Reiko Heckel. Tutorial Introduction to Graph Transformation: A Software Engineering Perspective. In *Proceedings of the First International Conference on Graph Transformation*, pages 402–429, 2002. 25

[BHB99] Ivan T. Bowman, Richard C. Holt, and Neil V. Brewster. Linux as a Case Study: Its Extracted Software Architecture. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 555–563, Los Angeles, CA, May 1999. 60

[BHLV94] Francois Bodart, Anne-Marie Hennebert, Jean-Marie Leheureux, and Jean Vanderdonckt. A Model-based Approach to Presentation: A Continuum from Task Analysis to Prototype. In *Proceedings of Interactive Systems: Design, Specification, and Verification (DSV-IS'94)*, pages 25–39. Eurographics, June 1994. 29

[BHvW00] M. Bruls, K. Huizing, and J. van Wijk. Squarified Treemaps. In *Proceedings of Joint Eurographics and IEEE TCVG Symposium on Visualization (TCVG'00)*, pages 33–42, 2000. 93

[BIR] The Business Integration and Reporting Toolkit. http://www.eclipse.org/birt. 36, 46, 51

[BMG00] Benjamin B. Bederson, Jon Meyer, and Lance Good. Jazz: An Extensible Zoomable User Interface Graphics Toolkit in Java. In *UIST*, pages 171–180, 2000. 35

[BMS+08] Christopher Bennett, Del Myers, Margaret-Anne Storey, Daniel M. German, David Ouellet, Martin Salois, and Phillipe Charland. A survey and evaluation of tool features for understanding reverse engineered sequence diagrams. *To appear in Wiley's Journal of Software Maintenance and Evolution (PCoda Special Issue)*, 2008. 109

[Böc05] Günter Böckle. Introduction to Software Product Line Engineering. In *Software Product Line Engineering*, pages 3–18. Springer Berlin Heidelberg, 2005. 12

[Boe88] B. W. Boehm. A Spiral Model of Software Development and Enhancement. *Computer*, 21(5):61–72, May 1988. 17

[BSFL06] R. Ian Bull, Margaret-Anne Storey, Jean-Marie Favre, and Marin Litoiu. An Architecture to Support Model Driven Software Visualization. In *Proceedings of the 14th International Conference on Program Comprehension (ICPC'06)*, pages 100–106, June 2006. 114

[BSM02] Casey Best, Margaret-Anne Storey, and Jeff Michaud. Designing a component-based framework for visualization in software engineering and knowledge engineering. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, pages 323–326, 2002. 36, 41, 97, 103, 107

[BSM+03] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework*. Addison Wesley, 2003. 14, 81, 112

[BSW02] Benjamin B. Bederson, Ben Shneiderman, and Martin Wattenberg. Ordered and Quantum Treemaps: Making Effective Use of 2D Space to Display Hierarchies. *ACM Transactions on Graphics*, 21(4):833–854, October 2002. 93

[Bul] R. Ian Bull. Eclipse Project: PDE Dependency Visualization. retrieved January 2008 from http://www.eclipse.org/pde/incubator/dependency-visualization/index.php. 68

[Bul02] R. Ian Bull. Abstraction Patterns for Reverse Engineering. Master's thesis, University of Waterloo, 2002. 119

[Bul06] R. Ian Bull. Integrating Dynamic Views Using Model Driven Development. In *Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research (CASCON'06)*, page 17, 2006. 123

[Cas85] Albert F. Case. Computer-Aided Software Engineering (CASE): Technology For Improving Software Development Productivity. *ACM SIGMIS Database*, 17(1):35–43, 1985. 11

[CDMS01] James R. Cordy, Thomas R. Dean, Andrew J. Malton, and Kevin A. Schneider. Software Engineering by Source Transformation - Experience with TXL. In *Proceedings of First IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'01), 2001*, pages 168–178, 2001. 24

[CE00] Krysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000. 116

[CE04] Brendan Cleary and Chris Exton. CHIVE – A Program Source Visualisation Framework. In *Proceedings of the 12th IEEE International Workshop on Program Comprehension (IWPC'04)*, 2004. 42

[CFKW95] Yih-Farn R. Chen, Glenn S. Fowler, Eleftherios Koutsofios, and Ryan S. Wallach. Ciao: A graphical navigator for software and document repositories. In *Proceedings of International Conference Software Maintenance, ICSM'95*, pages 66–75. IEEE Computer Society, 1995. 60

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *Proceedings of OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003. 26

[CH06] K. Czarnecki and S. Helsen. Feature-Based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–645, 2006. 25, 26

[CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing Cardinality-Based Feature Models and their Specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005. 155

[Chi00] Ed H. Chi. A Taxonomy of Visualization Techniques using the Data State Reference Model. In *Proceedings of the IEEE Symposium on Information Vizualization 2000*, pages 69–75, 2000. 38

[CI90] Elliot J. Chikofsky and James H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. In *IEEE Software*, volume 7, pages 13–17, Jan. 1990. 60

[CM97] Stuart K. Card and Jock Mackinlay. The Structure of the Information Visualization Design Space. In *Proceedings of the 1997 IEEE Symposium on Information Visualization (InfoVis '97)*, pages 92–99, 1997. 38

[CMR92] Mariano Consens, Alberto Mendelzon, and Arthur Ryman. Visualizing and Querying Software Structures. In *Proceedings of International Conference on Software Engineering (ICSE'92)*, pages 138–156, 1992. 41, 42

[CMS99] Stuart K. Card, Jock D. Mackinlay, and B. Shneiderman. *Readings in Information Visualization*. Morgan Kaufmann Publishers, 1999. 3, 32, 33, 34, 39, 40, 49, 73

[CW07] Krzysztof Czarnecki and Andrzej Wasowski. Feature Diagrams and Logics: There and Back Again. In *Software Product Line Conference, 2007. SPLC 2007. 11th International*, pages 23–34, 10-14 Sept. 2007. 155

[DFAB97] Alan Dix, Janet Finlay, Gregory Abowd, and Russell Beale. *Human-Computer Interaction, second edition*. Prentice Hall, 1997. 35, 77

[Die07] Stephan Diehl. *Software Visualization*. Springer, 2007. 38, 74

[Dij59] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numensche Mathematik*, 1:269–271, 1959. 72

[ECL] Eclipse. http://www.eclipse.org. 7, 46

[ERG02] Peter Eklund, Nataliya Roberts, and Steve Green. OntoRama: Browsing RDF Ontologies using a Hyperbolic-style Browser. In *Proceedings of the First International Symposium on Cyber Worlds*, pages 405–411, 2002. 32

[Fav01] Jean-Marie Favre. G^{see}: a generic software exploration environment. In *Proceedings of 9th International Workshop on Program Comprehension (IWPC'01)*, pages 233–244, Toronto, ON, May 2001. 42

[Fav04a] Jean-Marie Favre. Towards a Basic Theory to Model Model Driven Engineering. In *Workshop on Software Model Engineering (WiSME'04)*, page N/A, 2004. 100

[Fav04b] Jean-Marie Favre. CacOphoNy: Metamodel-Driven Software Architecture Reconstruction. In *Proceedings of 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 204–213, November 2004. 1, 6

[FB95] George W. Furnas and Benjamin B. Bederson. Space-Scaled Diagrams: Understanding Multiscale Interfaces. In *Proceedings of Human Factors in Computing Systems (CHI'95)*, pages 234–241, Denver, Colorado, 1995. 2, 36

[FC01] Liliana Favre and Silvia Clérici. A Systematic Approach to Transform UML Static Models to Object-Oriented Code. In *Unified Modeling Language: Systems Analysis, Design and Development Issues*. IGI Global, April 2001. 22

[FES03] J. Favre, Jacky Estublier, and Remy Sanlaville. Tool adoption issues in very large software company. In *Proceedings of 3rd Workshop on Adoption Centric Software Engineering (ACSE'03)*, 2003. 74

[FGDTS06] Robert B. France, Sudipto Ghosh, Trung Dinh-Trong, and Arnor Solberg. Model-Driven Development Using UML 2.0: Promises and Pitfalls. 39(2):59–66, February 2006. 30

[FHK+97] Pat Finnigan, Richard C. Holt, Ivan Kalas, Scott Kerr, Kostas Kontogiannis, Hausi A. Müller, John Mylopoulos, Stephen Perelgut, Martin Stanley, and

Kenny. Wong. The Software Bookshelf. *IBM Systems Journal*, 36(4):564–593, November 1997. 21, 46, 60, 63

[FKO98] L. Feijs, R. Krikhaar, and R. Van Ommering. A Relational Approach to Support Software Architecture Analysis. *Software-Practice and Experience*, 28(4):371–400, 1998. 62

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison-Wesley, 1999. 18

[FP02] J. Fekete and C. Plaisant. Interactive Information Visualization of a Million Items. In *Proceedings of IEEE Symposium on Information Visualization*, pages 117–124, October 2002. 36, 92

[FPB87] Jr. Frederick P. Brooks. No Silver Bullet Essence and Accidents of Software Engineering. *Computer*, 20(4):10–19, 1987. 1, 149

[FS07] Sean M. Falconer and Margaret-Anne D. Storey. A cognitive support framework for ontology mapping. In *ISWC/ASWC*, pages 114–127, 2007. 128

[GB03] Erich Gamma and Kent Beck. JFace – User Interface Frameworks. In *Contributing to Eclipse Principles, Patterns and Plug-Ins*, page 335. Addison Wesley, 2003. 105

[GEF] Graphical Editor Framework. Website. http://www.eclipse.org/gef. 35, 103, 107

[GGL05] Lars Grunske, Leif Geiger, and Michael Lawley. A Graphical Specification of Model Transformations with Triple Graph Grammars. *Lecture Notes in Computer Science, Springer Berlin / Heidelberg*, 3748:284–298, 2005. 26

[GGZ+05] Lars Grunske, Leif Geiger, Albert Zndorf, Niels Van Eetvelde, Pieter Van Gorp, and Daniel Varro. Using Graph Transformation for Practical Model-

Driven Software Engineering. In *Model-Driven Software Development*, pages 91–117. Springer Berlin Heidelberg, 2005. 16

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. 71, 81

[GK07] Joel Greenyer and Ekkart Kindler. Reconciling TGGs with QVT. In *Model Driven Engineering Languages and Systems*, pages 16–30. Springer Berlin Heidelberg, 2007. 27

[GL00] Michael W. Godfrey and Eric H. S. Lee. Secrets from the Monster: Extracting Mozilla's Software Architecture. In *Proceedings of Second Symposium on Constructing Software Engineering Tools (CoSET'00)*, page N/A, Limerick, Ireland, June 2000. 63

[GMF] The Graphical Modeling Framework. http://www.eclipse.org/gmf. 29

[Gre06] Jack Greenfield. Bare-Naked Languages or What Not to Model. October 2006. Available at: http://msdn2.microsoft.com/en-us/library/bb245772.aspx. 30

[Gru95] Thomas R. Gruber. Toward Principles for the Design of Ontologies used for Knowledge Sharing. *International Journal of Human-Computer Studies*, 43(5-6):907–928, November 1995. 54

[GS93] David Garlan and Mary Shaw. An Introduction to Software Architecture. In *V. Ambriola and G. Tortora, editors, Advances in Software Engineering and Knowledge Engineering*, pages 1–39, Singapore, 1993. World Scientfic Publishing Company. 60

[GSCK04] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools*. John Wiley Publishing Inc., 2004. 10

[GTK] GTK+ The GIMP Toolkit. website. http://www.gtk.org/. 34

[GXL] GXL The GXL Homepage. Website. http://www.gupro.de/GXL. 21, 64

[Har88] David Harel. On Visual Formalisms. *Communications of the ACM*, 31(5):514–530, May 1988. 36

[HCL05] Jeffrey Heer, Stuart K. Card, and James A. Landay. Prefuse: A Toolkit for Interactive Information Visualization. In *Proceedings of the SIGCHI conference on Human factors in Computing Systems*, pages 421–430, 2005. 103

[HH02] Ahmed E. Hassan and Richard C. Holt. Architecture Recovery of Web Applications. In *In Proceedings of 24th International Conference on Software Engineering (ICSE'02)*, pages 349–359, 2002. 63

[HMM00] Ivan Herman, Guy Melançon, and M. Scott Marshall. Graph Visualization and Navigation in Information Visualization: A Survey. *IEEE Transactions on Visualization and Computer Graphics*, 6:24–43, 2000. 6, 32

[Hol] GROK Richard C. Holt. The Grok Programming Language. http://plg.uwaterloo.ca/ holt/papers/grok-intro.html. 60, 62

[Hol97] Richard C. Holt. An introduction to TA: The Tuple-Attribute Language. Available at `http://plg.uwaterloo.ca/~holt/papers/ta.html`, 1997. 21, 60, 64

[Hol99] Richard C. Holt. Software Architecture Abstraction and Aggregation as Algebraic Manipulations. In *Proceedings of the 1999 Conference of the Center for Advanced Studies on Collaborative Research (CASCON'99)*, page 210, 1999. 41

[HS05] Brian Henderson-Sellers. UML the Good, the Bad or the Ugly? Perspectives from a panel of experts. *Software and Systems Modeling*, 4(1):4–13, February 2005. 30

[HSW00] Ric Holt, Andy Schürr, and Andreas Winter. GXL: Towards a Standard Exchange Format. In *Proceedings of 2000 Working Conference on Reverse Engineering (WCRE-00)*, pages 162–171, Brisbane, Australia, November 2000. 64

[Hud] Randy Hudson. Eclipse bug (130386): Almost impossible to browse problems using plug-in validation. Retrieved January 2008 from http://bugs.eclipse.org/130386. 68

[Ins85] Alfred Inselberg. The plane with parallel coordinates. *The Visual Computer*, 1(4):69–91, December 1985. 203

[IS05] Robert G. Pettit IV and Julie A. Street. Lessons Learned Applying UML in the Design of Mission Critical Software. *UML 2004 Satellite Activities, Lecture Notes in Computer Science*, 3297:129–137, 2005. 149

[JBK06] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In *Proceedings of the fifth international conference on Generative programming and Component Engineering*, pages 249–254, 2006. 24

[JFr] JFreeChart. website. available at http://www.jfree.org/. 35

[JGr] JGraph. website. http://www.jgraph.com. 35, 103

[JK06] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. *Lecture Notes in Computer Science, Springer Berlin / Heidelberg*, 3844:128–138, 2006. 25, 27, 116, 118

[JS91] Brian Johnson and Ben Shneiderman. Treemaps: A Space-Filling Approach to the Visualization of Hierarchical Information Structures. In *Proceedings of IEEE Conference on Visualization*, pages 284–291, 1991. 36, 57, 58, 92

[JUN] JUNG. Java Universal Network/Graph Framework. available at http://jung.sourceforge.net/. 103, 107

[KC98] Rick Kazman and S. Jeromy Carrière. Playing detective: Reconstructing software architecture from the available evidence. *Automated Software Engineering*, 6:107–138, 1998. 60, 63

[Ken96] Kenny Wong. Rigi User's Manual: Version 5.4.3, November 1996. 21

[Ken02] Stuart Kent. Model Driven Engineering. In *Proceedings of 3rd international Conference on Integrated Formal Methods*, pages 286–298, 2002. 11

[Kie06] Holger Michael Kienle. *Building Reverse Engineering Tools with Software Components*. PhD thesis, University of Victoria, 2006. 74

[KLM04] Holger M. Kienle, Marin Litoiu, and Hausi A. Müller. Using Components to Build Software Engineering Tools. In *Proceedings of the Fourth International Workshop on Adoption-Centric Software Engineering (ACSE 2004)*, pages 36–42, Edinburgh, Scotland, 2004. 74

[Kon] Konqueror - Web Browser, File Manager - and more! website. http://www.konqueror.org/. 94

[Kön05] Alexander Königs. Model Transformation with Triple Graph Grammars. In *Proceedings of Model Transformations in Practice Satellite Workshop of MODELS 2005*, 2005. 26

[Kos03] Rainer Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Main-*

tenance and Evolution: Research and Practice, 15(2):87 – 109, April 2003. 6

[Kov98] Srdjan Kovecevic. Uml and user interface modeling. In *Lecture Notes in Computer Science*, volume 1681, pages 253–266. Springer-Verlag Berlin Heidelberg, 1998. 1, 29

[KR03] Vinay Kulkarni and Sreedhar Reddy. Separation of Concerns in Model-Driven Development. *IEEE Software*, 20(5):64–69, 2003. 1, 2

[LMSW03] Rob Lintern, Jeff Michaud, Margaret-Anne Storey, and Xiaomin Wu. Plugging-in visualization: experiences integrating a visualization tool with Eclipse. In *Proceedings of the 2003 ACM Symposium on Software visualization*, pages 47–58, San Diego, 2003. 46, 103

[LSW01] Carola Lange, Harry M. Sneed, and Andreas Winter. Comparing graph-based program comprehension tools to relational database-based tools. In *Proceedings of 9th International Workshop on Program Comprehension (IWPC'01)*, pages 209–218, May 2001. 41

[Mac86] Jock Mackinlay. Automating the Design of Graphical Presentations of Relational Information. *ACM Transactions on Graphics*, 5(2):110–141, April 1986. 42

[MDH01] Andrew J. Malton, Tom R. Dean, and Richard C. Holt. Union Schemas as a Basis for a C++ Extractor. In *Proceedings of 8th Working Conference on Reverse Engineering (WCRE'01)*, pages 59–67, 2001. 60

[Met05] Andreas Metzger. A Systematic Look at Model Transformations. In *Model-Driven Software Development*, pages 19–33. Springer Berlin Heidelberg, 2005. 16, 26, 133

[MG03] Paul Mutton and Jennifer Golbeck. Visualization of Semantic Metadata and Ontologies. In *Proceedings of Seventh International Conference on Information Visualization (InfoViz'03)*, pages 300–305, 2003. 32

[Mic03] Jeffrey William Michaud. A Software Customization Framework. Master's thesis, University of Victoria, 2003. 11, 12, 13, 39

[MJ82] Daniel D. McCracken and Michael A. Jackson. Life Cycle Concept Considered Harmful. 7(2):29–32, April 1982. 30

[MK88] Hausi A. Müller and Karl Klashinsy. Rigi: A system for programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering (ICSE'88)*, pages 80–86, Singapore, April 1988. 21, 60, 63, 64

[MK93] Lil Mohan and R. L. Kashyap. A Visual Query Language for Graphical Interaction with Schema-Intensive Databases. In *IEEE Transactions on Knowledge and Data Engineering*, volume 5, pages 843–858, 1993. 42

[MMC02] Jonathan I. Maletic, Andrian Marcus, and Michael L. Collard. A Task Oriented View of Software Visualization. In *Proceedings of the First International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'02)*, pages 32–41, 2002. 44

[MNS01] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380, 2001. 60

[MS95] Alberto Mendelzon and Johannes Sametinger. Reverse Engineering by Visualizing and Querying. *Software – Concepts and Tools*, 16:170–182, 1995. 42

[MU90] Hausi A. Müller and James S. Uhl. Composing Subsystem Structures using (k,2)-partite Graphs. In *Proceedings of International Conference on Soft-*

ware Maintenance (ICSM'90), pages 26–29, San Diego, California, November 1990. 119

[Mye89] Brad A. Myers. User-Interface Tools: Introduction and Survey. *IEEE Software*, 6(1):15–23, 1989. 29

[NCS02] Chris North, Nathan Conklin, and Varun Saini. Visualization Schemas for Flexible Information Visualization. In *Proceedings of the 2002 Symposium on Information Visualization*, pages 15–22, 2002. 41

[NFM] N. F. Noy, R. W. Fergerson, and M. A. Musen. The knowledge model of Protege-2000: Combining interoperability and flexibility. In *Proceedings of 2nd International Conference on Knowledge Engineering and Knowledge Management (EKAW'00)*, pages 17–32, Juan-les-Pins, France, 2000. 46, 54

[Nor98] Donald A. Norman. *The Design of Everyday Things*. New York: Basic Books, 1998. 40

[NS00] Chris North and Ben Shneiderman. Snap-together visualization: A user interface for coodinating visualizations via relational schemata. In *Advanced Visual Interfaces*, pages 128–135, 2000. 41, 121

[NW04] Steve Northover and Mike Wilson. *SWT: The Standard Widget Toolkit, Volume 1 (The Eclipse Series)*. Addison Wesley, 2004. 34, 103, 107

[OMG06a] OMG/MOF. Mof core specification, v2.0, (formal/2006-01-01), 2006. available at http://www.omg.org. 17, 20

[OMG06b] OMG/OCL. OCL Specification, v2.0, (Formal/2006-05-01), 2006. available at http://www.omg.org. 22

[OMG07a] OMG/QVT. MOF – Query / View / Transformation (QVT), 2007. available at http://www.omg.org. 27, 118

[OMG07b] OMG/UML. Unified Modeling Language (UML), version 2.1.1, (Formal/2007-02-03), 2007. available at http://www.omg.org. 17, 19

[PBS93] B.A. Price, R.M Baecker, and I.S. Small. A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993. 32, 38

[PE00] Magnus Penker and Hans-Erik Eriksson. *Business Modeling With UML: Business Patterns at Work*. John Wiley, 2000. 1

[Pen93] David Allan Penny. *The Software Landscape: A Visual Formalism for Programming-In-The-Large*. PhD thesis, University of Toronto, 1993. 60, 97

[PGB02] C. Plaisant, J. Grosjean, and B.B. Bederson. Spacetree: Supporting exploration in large node link tree, design evolution and empirical evaluation. In *Proceedings of IEEE Symposium on Information Visualization, 2002 (InfoViz'02)*, pages 57–64, Boston, October 2002. 36

[PHP03] Darius Pfitzner, Vaughan Hobbs, and David Powers. A unified taxonomic framework for information visualization. In *Proceedings of the Asia-Pacific symposium on Information visualization - Volume 24*, pages 57 – 66, 2003. 38

[Pop04] Remko Popma. Jet tutorial (write code that writes code), 2004. available at http://www.eclipse.org. 116, 209

[PP01] Tim Pattison and Matthew Phillips. View coordination architecture for information visualisation. In Peter Eades and Tim Pattison, editors, *Australian Symposium on Information Visualisation, (invis.au 2001)*, Sydney, Australia, 2001. ACS. 121

[Pra71] Terrence W. Pratt. Pair Grammars, Graph Languages and String-to-Graph Translations. *Journal of Computer and Systems Sciences*, 5:560–595, 1971. 26

[Rai05] Chris Raistrick. Applying MDA and UML in the Development of a Healthcare System. *UML 2004 Satellite Activities, Lecture Notes in Computer Science 3297*, pages 203–218, 2005. 149

[RC94] Ramana Rao and Stuart K. Card. The table lens: merging graphical and symbolic representations in an interactive focus + context visualization for tabular information. In *Proceedings of Conference on Human Factors in Computing Systems*, pages 318–322, Boston, Massachusetts, United States, 1994. 36

[Rei01] Steven P. Reiss. An Overview of Bloom. In *Proceedings of the 2001 ACM Workshop on Program Analysis for Software Tools and Engineering*, pages 2–5, 2001. 42

[Rei02] Steven P. Reiss. A Visual Query Language for Software Visualization. In *Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)*, pages 80–82, 2002. 42

[RJ97] Don Roberts and Ralph E. Johnson. Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks. In *Pattern Languages of Program Design 3*. Addison Wesley, 1997. 150

[RLS+03] Derek Rayside, Marin Litoiu, Margaret-Anne Storey, Casey Best, and Robert Lintern. Visualizing flow diagrams in websphere studio using shrimp views. *Information Systems Frontiers*, 5:161–174, 2003. 2, 32

[RMC91] George G. Robertson, Jock D. Mackinlay, and Stuart K. Card. Cone Trees: animated 3D visualizations of hierarchical information. In *Proceedings*

of the Conference on Human Factors in Computing Systems: Reaching Through Technology, pages 189–104, 1991. 36

[Rob05] Naomi B. Robbins. *Creating More Effective Graphs*. John Wiley, 2005. 191, 192

[Roy70] W. W. Royce. Managing the Development of Large Software Systems: Concepts and Techniques. In *Proceedings of IEEE Computer Society Press WESTCON (Chapter 4)*, 1970. 17

[Sch] A. Schurr. Specification of graph translators with triple graph grammars. In *In Proceedings of the 20 International Workshop on Graph-Theoretic Concepts in Computer Science, Herrsching, Germany, June 1994. Springer Verlag.* 26

[Sch06] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006. 14, 15

[SDBP98] John Stasko, John Dominque, Marc H. Brown, and Blaine A. Price, editors. *Software Visualization: Programming as a Multimedia Experience*. Massachusetts Institute of Technology, 1998. 74

[Sei03] Ed Seidewitz. What Models Mean. *IEEE Software*, 20:26–32, October 2003. 19

[Sel03] Bran Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003. 1, 4, 11, 112, 147

[Sel06] Bran Selic. Model-Driven Development: Its Essence and Opportunities. In *Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 313–319, 2006. 11, 150

[SFM99] Margaret-Anne D. Storey, F. David Fracchia, and Hausi A. Müller. Customizing a Fisheye View Algorithm to Preserve the Mental Map. *Journal of Visual Languages & Computing*, 10(3):245–267, June 1999. 97

[Shn96] Ben Shneiderman. The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In *IEEE Visual Languages*, number UMCP-CSD CS-TR-3665, pages 336–343, College Park, Maryland 20742, U.S.A., 1996. 38, 81

[SHNM05] Matthew Scarpino, Stephen Holder, Stanford Ng, and Laurent Mihalkovic. *SWT/JFace In Action*. Manning Publications Co., 2005. 102, 105

[Sii00] Harri Siirtola. Direct manipulation of parallel coordinates. In *Proceedings of IEEE International Conference on Information Visualization (InfoViz'00)*, pages 373–378, 2000. 205

[SK02] Shane Sendall and Wojtek Kozaczynski. Model Transformation the Heart and Soul of Model-Driven Software Development. Technical report, Ecole Polytechnique Fédérale de Lausanne, 2002. 17, 24, 111, 112

[SKBS97] Danny Soroker, Michael Karasick, John Barton, and David Streeter. Extension Mechanisms in Montana. In *Proceedings of the 8th Israeli Conference on Computer-Based Systems and Software Engineering*, pages 119–128, 1997. 74

[SLVA97] Janice Singer, Timothy C. Lethbridge, N. Vinson, and Nicolas Anquetil. An examination of software engineering work practices. In *Proceedings of the 1997 Conference of the Center for Advanced Studies on Collaborative research (CASCON'97)*, pages 209–223, 1997. 74

[SM96] M.-A. D. Storey and H. A. Müller. Graph Layout Adjustment Strategies. In *Lecture Notes in Computer Science*, volume 1027, pages 487–499. Springer Verlag, 1996. 91

[SMS+01] Margaret-Anne Storey, Mark Musen, John Silva, Casey Best, Neil Ernst, Ray Fergerson, and Natasha Noy. Jambalaya: Interactive visualization to enhance ontology authoring and knowledge acquisition in protege. In *Proceedings of Workshop on Interactive Tools for Knowledge Capture, K-Cap-2001*, Victoria, BC, 2001. 32, 46, 103

[Sne07] Harry M. Sneed. The Drawbacks of Model-Driven Software Evolution. IEEE CSMR 07 - Workshop on Model Driven Software Evolution (MoDSE2007), March 2007. 30

[Som04] Ian Sommerville. *Software Engineering 7*. Pearson Education Limited, 2004. 15, 77

[Ste46] S. S. Stevens. On the Theory of Scales of Measurement. *Science*, 103(2684), 1946. 82

[Sto98] Margaret-Anne D. Storey. *A Cognitive Framework for Describing and Evaluating Software Exploration Tools*. PhD thesis, Simon Fraser University, 1998. 44

[SWFM97] M.-A.D. Storey, K. Wong, F.D. Fracchia, and H.A. Müller. On integrating visualization techniques for effective software exploration. In *Proceedings of IEEE Symposium on Information Visualization (InfoVis'97)*, pages 38–45, Phoenix, AZ, October 1997. 32, 119

[SwJBHH06] Tomas Stahl, Markus Völter with Jorn Bettin, Arno Hasse, and Simon Helsen. *Model-Driven Software Development*. John Wiley and Sons Ltd., 2006. 17

[TH96] Vassilios Tzerpos and R.C. Holt. A Hybrid Process for Recovering Software Architecture. In *Proceedings of the 1996 Conference of the Center for Advanced Studies on Collaborative research (CASCON'96)*, page 38, 1996. 63

[TM04] Melanie Tory and Torsten Möller. Rethinking visualization: A high-level taxonomy. In *Proceedings IEEE Symposium on Information Visualization, 2004. INFOVIS 2004*, pages 151 – 158, 2004. 38

[TMWW93] Scott R. Tilley, Hausi A. Müller, Michael J. Whitney, and Kenny Wong. Domain-Retargetable Reverse Engineering. In *Proceedings of Conference on Software Maintenance (ICSM'93)*, pages 142–151, 1993. 13

[Tuk77] John W. Tukey. *Exploratory Data Analysis*. Addison Wesley Publishing Company, 1977. 42

[Twe97] Lisa Tweedie. Characterizing Interactive Externalizations. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems, 1997*, pages 375–382, 1997. 38

[TWSM94] Scott R. Tilley, Kenny Wong, Margaret-Anne D. Storey, and Hausi A. Müller. Programmable Reverse Engineering. *Journal of Software Engineering and Knowledge Engineering*, 4(4):501–520, 1994. 13, 41

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notes*, pages 26–36, 2000. 1, 24

[VK06] Markus Völter and Bernd Kolb. Best Practices for Model-to-Text Transformations. In *In Eclipse Summit Europe, Modeling Symposium, 2006.*, 2006. 27

[Voe] Markus Voelter. open Architecture Ware. http://www.openarchitectureware.org/. 14, 24, 116, 128

[VWvH+07] Fernanda B. Viégas, Martin Wattenberg, Frank van Ham, Jesse Kriss, and Matt McKeon. Many Eyes: A Site for Visualization at Internet Scale. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1121–1128, 2007. 36, 46, 49

[W3C99] W3C. XSL Transformations (XSLT) Version 1.0, 1999. available at http://www.w3c.org. 26

[W3C04] W3C. XML Schema (XSD) Version 1.1, 2004. available at http://www.w3.org/XML/Schema. 17, 116

[War04] Colin Ware. *Information Visualization: Perception for Design*. Morgan Kaufmann Publishers, 2004. 32

[Was90] Anthony I. Wasserman. Tool integration in software engineering environments. In *Proceedings of the International Workshop on Environments on Software Engineering Environments*, pages 137 – 149, Chinon, France, 1990. Springer-Verlag New York. 11, 12, 39

[Wat98] Martin Wattenberg. Map of the market, 1998. SmartMoney.com, http://smartmoney.com/marketmap. 94

[Wat99] Martin Wattenberg. Visualizing the stock market. In *Proceedings of ACM CHI 99, Extended Abstracts*, pages 188–189, 1999. 32, 94

[Weg02] Hans Wegener. Agility in Model-Driven Software Development? Implications for Organization, Process and Architecture. OOPSLA 2002 Workshop on Generative Techniques in the Context of Model-Driven Architecture, November 2002. 15, 30

[WMSL04] Xiaomin Wu, Adam Murray, Margaret-Anne Storey, and Rob Lintern. A reverse engineering approach to support software maintenance: version control knowledge extraction. In *Proceedings 11th Working Conference on Reverse Engineering*, pages 90–99, November 2004. 94

[Won00] Kenny Wong. *The Reverse Engineering Notebook*. PhD thesis, University of Victoria, 2000. 75

[Wu03] Xiaomin Wu. Visualization of version control information. Master's thesis, University of Victoria, 2003. 64

[WW06] Thomas Weigert and Frank Weil. Practical Experiences in Using Model-Driven Engineering to Develop Trustworthy Computing Systems. In *Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC'06)*, pages 208–217, 2006. 149

[Xul07] Xulin Zhao and Ying Zou and Jen Hawkins and Bhadri Madapusi. A Business-Process-Driven Approach for Generating E-Commerce User Interfaces. *Model Driven Engineering Languages and Systems, LNCS, Springer Berlin / Heidelberg*, 4735:256–270, 2007. 29

[ytdc] yWorks: the diagramming company. yfiles. available at http://www.yworks.com. 103

[Zha03] Kang Zhang. *Software Visualization: From Theory to Practice*. Kluwer Academic Publishers, 2003. 74

APPENDIX A

Visualization Patterns

IN Chapter 5, we introduced four visualization patterns. In this appendix we expand on our catalog and introduce another eight. Included in this appendix are visualization patterns for: 1) a list, 2) a line chart, 3) a pie chart, 4) a map, 5) a heat-map, 6) a table, 7) parallel coordinates, and 8) a tree.

List

Intent

A list viewer is ideal for representing a single collection of items. For a small number of elements (< 20) lists can show all the elements on the screen. For more elements, scrolling or fisheye views could be used. Lists can also be sorted to indicate rank or to improve navigation.

Motivation

One of the most basic information visualizations is the list viewer. This viewer displays each element sequentially in a scrollable view, with an optional filtering mechanism and the elements can be sorted based on some criteria. Many instances of this viewer also support both a label and an icon for each element in the view. The view usually scales remarkably well and it has been used to represent many thousands of elements.

While many implementations of the list viewer scale remarkably well, lists rarely support lazy loading. If they do support lazy loading, then this is available at the cost of quickly navigating the entire view.

Lists are one of the most utilized views in computer interfaces and they are often used in conjunction with other views to support overview+detail.

Formal Model

Figure A.1 presents the formal model for this viewer. This model defines two classes, the **ListViewer** itself and the **ListItems**. Each item can have a label and icon and is parented to the list viewer through an ordered list.

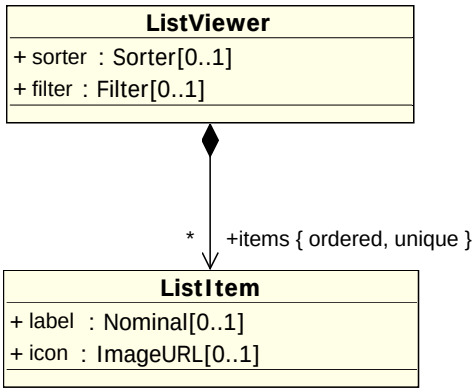

Figure A.1: List Viewer – Formal Model

Configurable Attributes

The text and icon for each of the list items can be customized to provide context or represent information about the element. In addition to the name and icon, the elements can be ordered based on some criteria. The elements can either be sorted to indicate rank, in which case the elements would be ordered by a property, or to improve navigation, in which case the elements may be ordered alphabetically.

Configurable User Interactions

In order to support navigation, four techniques are commonly used: scrolling, distortion-oriented views, quick search and filtering. Scrolling is usually available in list views without modification. Distortion-oriented views, such as a fisheye lenes, work by showing the entire set of elements and magnifying part of the display. Quick search is often activated as a user beings to type. In this case, the first element to match the character typed is highlighted. Finally, filtering can be achieved in a number of ways. One of the more common ways is to include a filter box, and as a user enters characters, only the elements which match the expression typed are retained in the view.

Known Uses

List views are often used as the overview in “Overview+Detail” displays. This implies that the list view is often the users first interaction with a system and once an element (or set of elements) is selected, another detailed view is brought into focus.

Line Chart

Intent

Like the bar chart, line charts are commonly available in charting packages and are used both for a single collection of items or groups of items. Unlike bar charts, line charts are more appropriate when the elements have both an interval value and an ordinal label. This way, the items can be ordered in a meaningful way.

Motivation

When faced with a set of ordinal tuples, especially if the tuples can be sorted by one of the values such that the tuples are step-wise increaseing, a line chart may be an appropriate means of representation. In Table A.1, the Age column could be sorted such that the values were step-wise increasing and Player 1 and Player 2 could each be considered a category. Figure A.2 shows how this data would appear when rendered using a line chart.

Age	Player 1's Goals	Player 2's Goals
19	51	43
20	55	48
21	92	54
22	71	70
23	87	85
24	73	45
25	52	19
26	62	44
27	40	69
28	54	17
29	40	69

Table A.1: Goals vs. Age

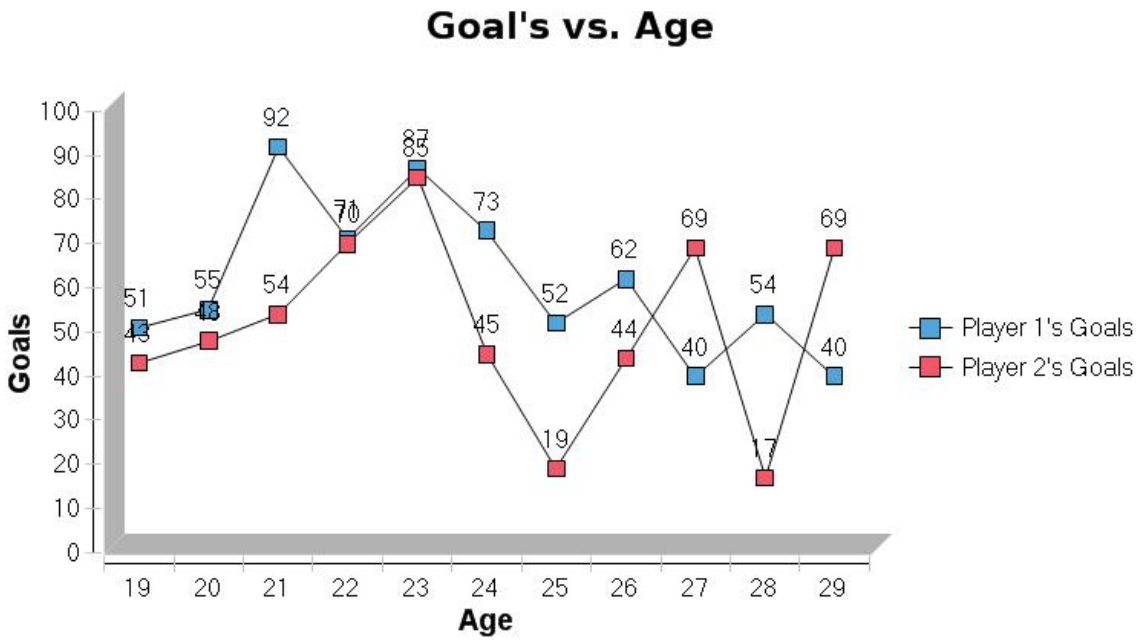

Figure A.2: Age vs. Goals Example

A bar chart could also be used to represent this information, but since all attributes are ordinal values and the tuples are ordered such that one of the values (age) is steap-wise increasing, a line chart better represents the connection between the values.

Formal Model

A line chart can be used to represent items or a group of items. We only documented the categorized line chart, however, a simple line chart would be very similar to the simple bar chart on page 85. Like the bar chart, the categorized line chart consists of the viewer class, the category, the item and the values. The only difference is that the line chart uses ordinal data for its label (or X-Axis).

The constraints specified on the categorized line chart are similar to those on the categorized bar chart. Each category must have a value for each item and each item must have a value for each category.

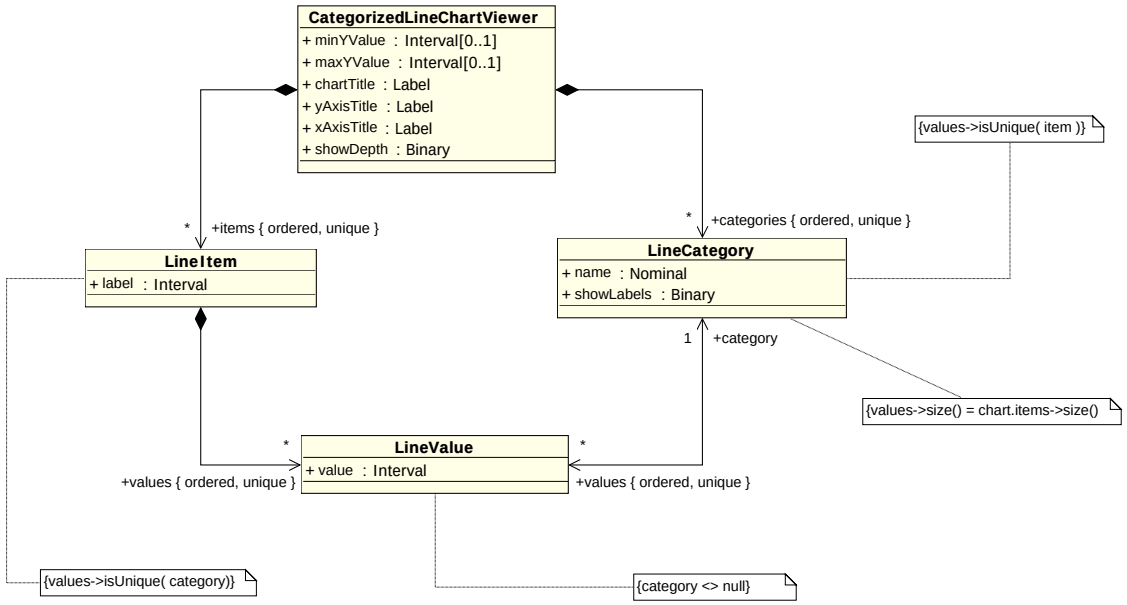

Figure A.3: Line Chart Viewer – Formal Model

Configurable Attributes

Colour and order are the two most common configurable attributes for a line chart. Colour can be used to distinguish between items and it can also be used to highlight items bringing them to the user's attention. The order of the elements should be determined by the items label data, however, a custom sorter could be used instead.

Configurable User Interactions

Interactive charting engines support operations such as mouse over or selection. Through scripts these interactions can be configured to highlight elements or perform some other user defined operation.

Known Uses

Line charts are commonly available in business and scientific charting tools like Excel. Google also supports an API for creating line charts for the web [1].

[1] http://code.google.com/apis/chart/

Pie Chart

Intent

Pie charts are useful for presenting ratios between a collection of items. If a group of items have an attribute that represent ratios between one another, a pie chart may be ideal.

Motivation

When studying the ratios between a set of ordinal data points, it may be desirable to present the proportions graphically. For example, in the case of program test cases, which can complete in 1 of four ways, *success*, *warning*, *failure* and *exception*, a table of results is often hard to interpret, especially if the number of test cases is large. On the other hand, a pie chart could be used to summarize the four categories. In this view a pie slice would show the number of tests that completed within a given category.

It should be pointed out however, that humans have a relatively hard time distinguishing between small discrepancies in the size of a pie slice, and if this is important, other views should be considered [Rob05]. If a pie chart must be used, ensure that the segments are ordered by size (largest to smallest) in a clockwise direction. Another option is to label each pie slice with its size.

Given a percentage of the pie chart, the size of the slice can be calculated by the following formula:

$$angle = percentage \div 100 \times 360^{\circ} \tag{A.1}$$

Formal Model

Since pie charts are not commonly used to represent categories (or series) of data, their formal model is fairly straight forward. The pie chart model contains two classes, the **PieChartViewer** and the **PieItem**. Each item has a value and a label. Many rendering engines support a 3D look, as well as the option to show labels and a chart title. Figure A.4 shows the formal model as a UML class diagram.

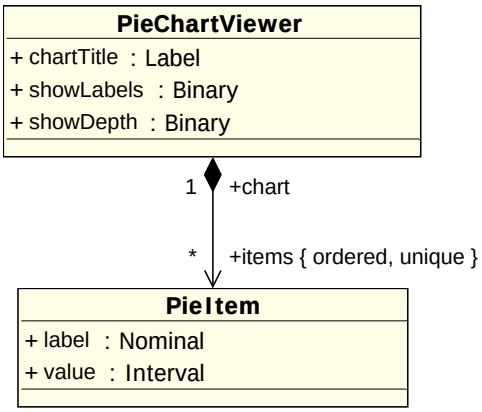

Figure A.4: Pie Chart Viewer – Formal Model

Configurable Attributes

The colour and position of each slice can be configured in a pie chart, although, it has been suggested that pie slices be ordered from largest to smallest in a clockwise direction [Rob05]. If the slices represent unique categories, distinct colours should be chosen. If, on the other hand, the slices represent related categories, then changing the colour value (the amount of white displayed) helps reenforce he connection.

Known Uses

Pie charts, despite the fact that they are often hard to interpret, are commonly used. They are most commonly used to reinforce large discrepancies.

Map

Intent

By mapping markers onto a map, humans can perceive proximity as well as geographical positioning. The map viewer is intended for data which is inherently location based.

Motivation

Figure A.5: Map of Paris

Understanding the relative distance between locations while considering natural barriers is difficult, especially to someone who is not familiar with the geography of the area. Victoria and Vancouver are only about 80 Kilometers apart, but the Strait of Georgia separating them severely impedes the travel time.

A map view helps users understand the distances between points, receive directions and better understand the locations of places by viewing them relative to known reference points. A first time traveler to Paris may not know where the Champs-Élysées is, but by viewing it relative to the Eiffel Tower or the Arc de Triomphe may provide the needed

context.

The Map Viewer is intended for use whenever a set of points (longitude and latitude) are available. A high quality map viewer implementation should support both an overview of the region (world, continent, country, province, city, etc..) and provide the needed facilities to zoom in for more detail. Many mapping applications show both an abstract view (showing streets, parks, water, etc...) and a satellite view (showing actual images of the area).

Formal Model

The map viewer model (Figure A.6) is comprised of two classes, the **MapViewer** and the **Marker**. Each marker must provide coordinates for both the longitude and latitude and optionally can provide a label or icon for each marker. The map viewer itself supports a centering point and default zoom level.

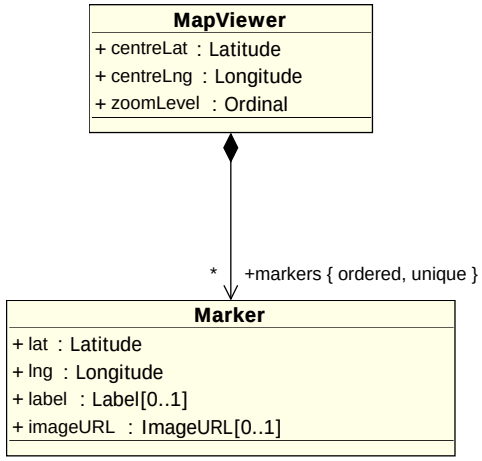

Figure A.6: Map Viewer – Formal Model

Configurable Attributes

Depending on the underlying rendering engine, a number of visual attributes can be configured. Both the label and icon for each marker can be customized, along with tools such as an overview window and scale. Additionally, a map may support a title and initial position.

Configurable User Interactions

User interactions such as panning and zooming help users keep context when using a large map. If the zoom level is too fine, a user may not know exactly what they are looking at, but by zooming out a broader view is presented. Other interactions such as selection can often be used to give more detail about a particular location.

Known Uses

A number of freely available on-line mapping applications exist. This includes Google and Yahoo Maps, MapQuest and Microsoft Live Maps. Google also has a rich client mapping tool called Google Earth.

Heat-Map

Intent

Since computer screens are inherently two-dimensional, using them to represent more that two dimensions always results in a compromise. Heat-maps use an X- and Y-axis for the first two dimensions, and colour value for the third.

Motivation

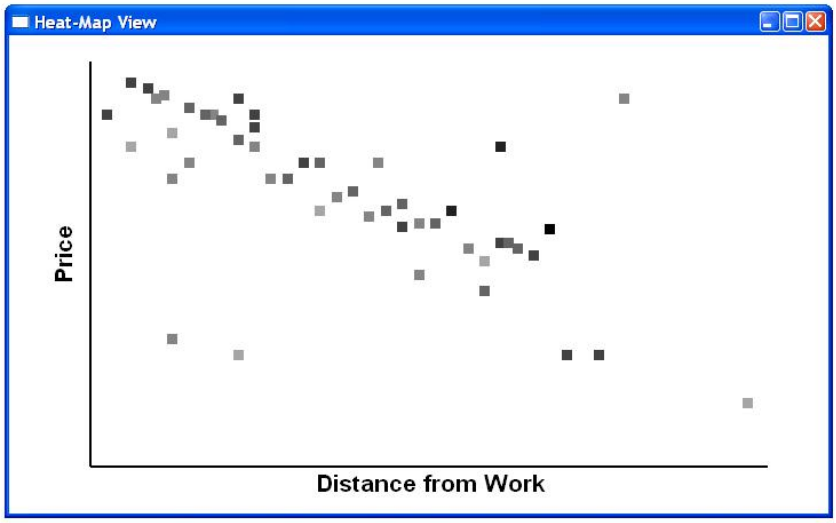

Figure A.7: Presentation of Trivariate Data

Understanding the relationships between trivariate data, such as price of a home, distance from work and number of bedrooms may seem like a relatively simple task. After all, we do live in a three dimensional world. While there are some three dimensional virtual reality displays available, these are often quite expensive and not readily available to the average application developer.

A heat-map may be an appropriate visualization, especially if data is unique among two dimensions. If the data is not uniquely defined but its first two dimensions a combing function must be used to calculate the colour. For example, in the case of 45 houses represented in figure A.7, if multiple houses had the exact same price and distance, then we choose the colour based on the maximum number of bedroom among all intersecting elements.

Using this display, we are able to see that it is possible to find a house relatively close to work without paying a fortune although none of these houses had "lots" of bedrooms.

Formal Model

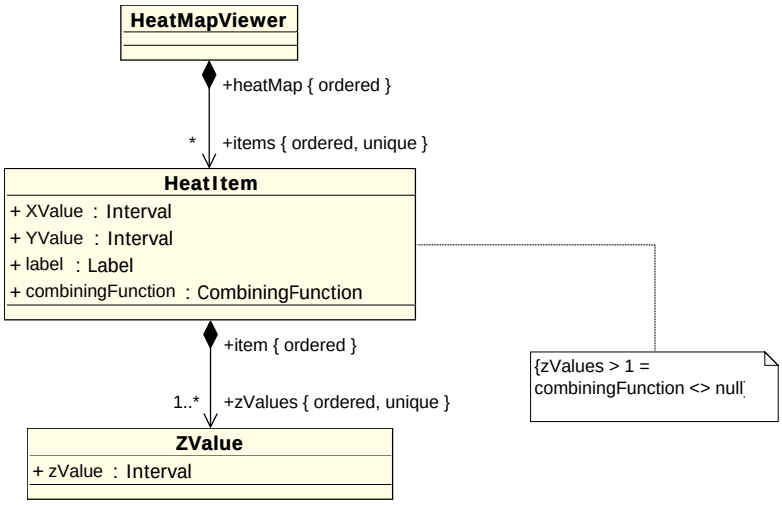

Figure A.8: Heat-Map Viewer – Formal Model

The formal model for this view contains three classes, the **HeatMapViewer**, **HeatItem** and the **ZValue** (Figure A.8). The viewer contains a number of heat items, or locations on the heat-map. Each of these items can have 1 or more ZValues. Each item can also have an optional icon or label. The most interesting thing about this model is the **CombingFunction** which is mandatory if the item has more than 1 ZValue.

Configurable Attributes

Labels and colours are configurable attributes for a heat-map. However, while colours can be chosen the colour value, the amount of white, is usually reserved for the result of the combing function.

Unless each heat item has one and only one zValue, a combining function must be configured. In the case of our house distance / price comparison, the colour value may be chosen to represent the maximum number of bedrooms. Other combining functions may include minimum or average value.

Configurable User Interactions

Interactions can be added to a heat-map in the form of selection listeners or mouse-overs, so that when a user selects a data-point, more information is displayed. Interaction can also be used to help users distinguish multiple zValues from a given item on the map.

Filtering can be used as an effective mechanism to hide elements which do not meet a given criteria. This is similar to the approach used in the film finder application [AS94].

Known Uses

Heat maps are often used in conjunction with geographical maps to show the value of a particular variable for that region. Housing prices by province, city or region is one such example.

Table

Intent

While the use tables to visualize information has existing long before computer, the interactivity supported by a computer has made these user interface elements one of the most widely used mechanisms for rendering multi-dimensional data. Using a table, each column represents a dimension, while each row represents a unique N-Tuple. In addition to being extremely versatile, the table widget is often a standard component in a graphical user interface toolkit.

Motivation

No single approach works well for all multi-dimensional data. Understanding how 4 or more variables are related across a list of elements and using that data to make decisions is highly domain dependent. Tables provide a "best effort" general means of visualizing multi-variant data. The table is ideal for showing a variety of dimensions for a set of items, and through row / column reordering and sorting, tables can assist users throughout the decision making process [Ber81].

A table can support multi-variant data by representing each elements as a row (or column) and all the attributes as cells in that row (or column). Sorting and rearranging rows and columns can help users compare attributes. According to Bertin, reordering rows and columns in a table can greatly approve information comprehension [Ber81].

Formal Model

Tables can be seen as a list of rows, each with a list of cells, or a list of columns each with a list of cells. Both representations are valid and the choice depends on the internal

representation that works best. For the purpose of this work, we have designed the formal model after the former, that is, a list of rows each with a list of cells. Figure A.9 shows the formal model for the table view.

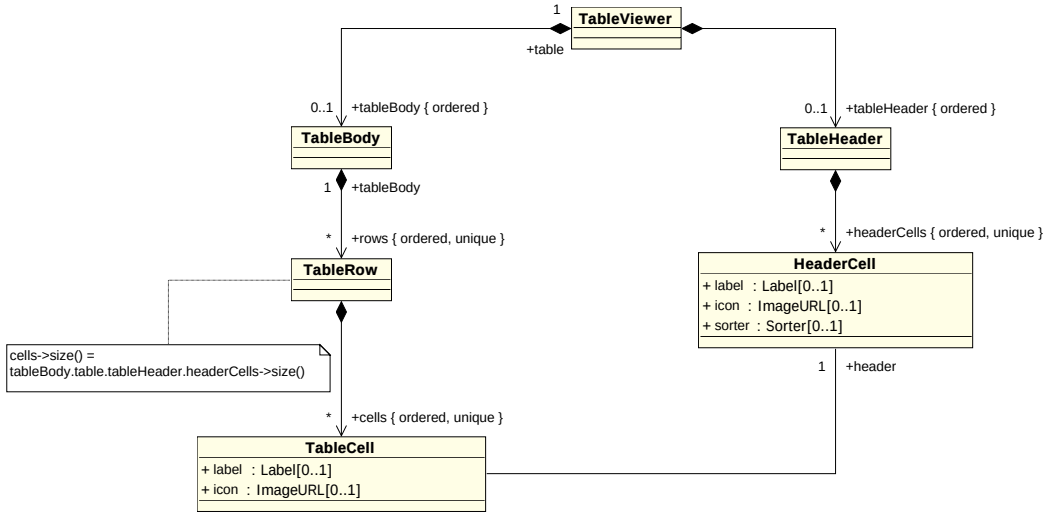

Figure A.9: Table Viewer – Formal Model

The table viewer model contains 6 classes. The **TableViewer** is the root of the model, and it contains both a **TableBody** and **TableHeader**. The TableHeader contains a number of elements each representing a column header.

The TableBody contains 0 or more rows, each with a number of cells. Each cell may contain an icon or label and the column header can have an optional sorter, which sorts all rows by the values in that column.

The only constraints specified on the this model is that each row much have the same number of cells as all other rows. In a sparse table, some of these elements can be left blank, that is, no label or icon would be assigned.

Configurable Attributes

The label and image for each cell in a table can be customized and in some systems, table cells can even contain rich user interface controls. Early internet applications even used

tables nested within tables to render complex layouts.

In addition to the visual attributes, table row and column ordering can also be customized.

Configurable User Interactions

Many interactive tables support column sorters by "clicking" on the column you wish to sort. Some libraries allow users to reorder entire rows or columns by dragging the headers around. These types of interactions make it much easier to compare variables, by reorganizing a table to better suite the data.

Tables also routinely support tool-tips, mouse hovers and selection events (both single cell selection and group selection), making tables a good interface control for overview in overview+detail interfaces.

Known Uses

Tables are natively supported by almost all mainstream user interface toolkits. Tables are also widely used on the web, as they have been part of the HTML standard since version 1.0. In addition to this, tables are the primary control for spreadsheet applications.

Parallel Coordinates

Intent

While tables can present multi-variant data, it is hard to see the results and comprehend the data. This is because a table presents all the data, and do not use any visual cues to distinguish or group elements.

Parallel coordinates [Ins85] is a two-dimensional techniques to visualize multidimensional data. It works by plotting an n-dimensional tuple across a set of parallel axis.

Motivation

The idea of parallel coordinates was originally conceived by a PhD student working studying multi-dimensional geometry. Frustrated with the fact that multi-dimensional geometry did not support the same visual representation that two and three dimensional geometry did, Inselberg designed the parallel coordinate system [Alf, Ins85]. Using parallel coordinates, an attributes are mapped to a single vertical axis. Then each element is plotted with respect to each of these attributes.

The parrel coordinate visualization can help spot anomalies and indicate trends among variables.

Formal Model

Our parallel coordinate formal model consists of four classes. The **ParallelCoordinateViewer**, the **Item**, the **Attribute** and the **Value**. The viewer contains a number of items and a number of attributes. Each item has a single value for each attribute.

There are two constraints that together state that each item must be mapped to exactly 1 attribute and each attribute must have a value for each item.

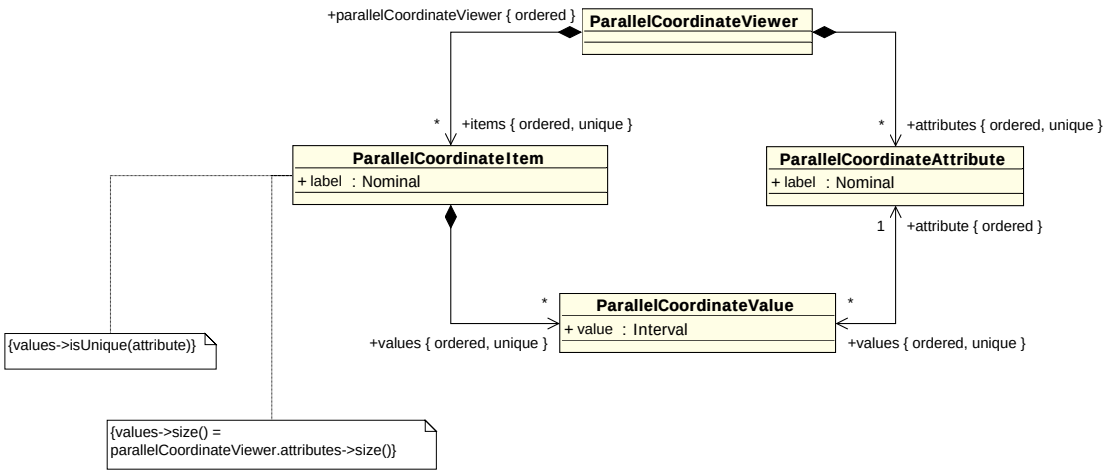

Figure A.10: Parallel Coordinate Viewer – Formal Model

Configurable Attributes

Each dimension and each elements (lines) can be configured with a label and colour. Line thickness and other visual cues can be used to bring attention to an item or dimension. As well, the axis for each dimension can be configured with a max, min, scale origin value.

Configurable User Interactions

The user interactions supported by the parallel coordinate system can greatly improve the understandability of the data. A common technique is attribute and/or item filtering. By filtering items based on some criteria, or a brushing technique [BC87], the view can be tailored to only show the relevant data.

One of the problems with the parallel coordinate view is the difficulty users have relating attributes when they are not situated next to one another. To assist users with this problem, the parallel coordinate view should support attribute reordering.

Known Uses

Parallel coordinates where first designed to study properties of multi-dimensional geometry, but the ideas have since been applied to domain of information visualization. One example of how parallel coordinates have been applied to interactive information visualization is the design of a tool to help consumers compare features of automobiles [Sii00]. The parallel coordinate system was used to plot 406 makes of automobiles across 9 dimensions.

Tree

Intent

Like the list and table viewer, the tree viewer is usually a standard component in most user interface libraries, making it a very common visualization technique in today's software systems. Interactive trees are also popular because they support lazy loading, that is, items are not loaded into the view until explicitly requested. This is why a file system explorer can handle up to one million files or more without incurring a performance penalty.

Motivation

The tree viewer organizes data in a hierarchical manner, directly attaching child nodes to their parents, and showing / hiding these child nodes through interactions with the parent element. Through lazy loading, trees have successfully represented data structured of over a million nodes while still providing an effective interface for data navigation. While trees are prevalent in many software systems, they only work well if a user remembers the path to the data. That is, if a user cannot remember what system the left ventricle is in, navigating becomes difficult.

Some tree viewers support a filtering mechanism, however, the lazy loading is often sacrificed when this is enabled since the entire structure must be loaded if it is to be searched.

Tree viewers also support selection, mouse overs and the ability to expand / collapse children. Because the tree viewer is such a common component in almost all widget toolkits, users have become familiar with them and the interactions they support.

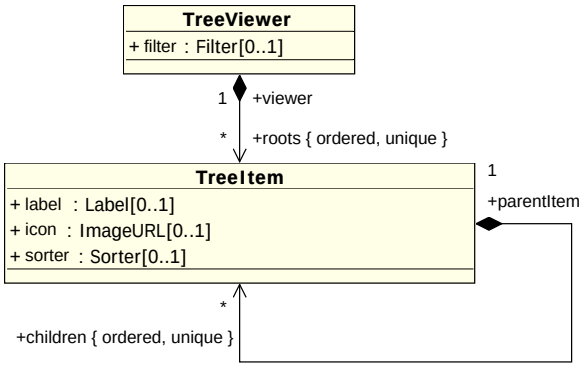

Figure A.11: Tree Viewer – Formal Model

Formal Model

The formal model for the tree viewer contains two classes, the viewer itself (**TreeViewer**) and the **TreeItem**. Each item can be configured with a label and an icon, and each item can contain zero or more children. A sorter can be placed on each item to configure how the child item are sorted. The viewer contains an optional filtering mechanism.

Configurable Attributes

Most tree viewers do not support a wide variety of configurations. Usually labels and icons can be customized for each element in the tree, along with font style and colour. While it may seem limiting, one of the advantages of the tree viewer is how consistent it looks and behaves across platforms.

Configurable User Interactions

Some of the more common configurable user interactions supported by a tree viewer are selection, mouse over and expand / collapse. A “collapse all” action is also a common to help restore the initial state of the view. Some tree viewers even support an expand all operation to show all the descendants, however, designers should be cautious when providing this functionality since many graphical interfaces will not be able to handle the

load if there are too many items to display.

Known Uses

The tree viewer is available in almost all widget toolkits and used in countless systems including directory listings, file browsers and integrated development environments.

APPENDIX B

Java Emitting Templates

THE following templates are used to generate model adapters for view models. Using these templates, future modelers can extend our collection of view models and generate adapters for their clients. These particular templates were designed to run on the Java Emitting Template (JET) tool [Pop04].

```
1  <%@ jet
2      package="org.vizmodels.generator.jet.gen"
3      class="GenContentProvider"
4      imports="java.util.*
5          org.eclipse.emf.ecore.EClass
6          org.vizmodels.generator.jet.gen.AdapterGenerator
7          org.eclipse.emf.ecore.EReference
8          org.eclipse.emf.ecore.EAttribute"
9
10     %>
11     <% GenContext context = (GenContext)argument; %>
12     <% context.beginNewFile(); %>
13     <% EClass rootClass = context.getViewer(); %>
14     <% String name = rootClass.getName(); %>
15     <% String className = context.createContentProviderClassName(rootClass); %>
16 package <%= context.createPackageName(rootClass) %>;
17
18 import org.eclipse.jface.viewers.IContentProvider;
19 <% context.markImportLocation(stringBuffer); %>
20
21 /**
22  * <%= className %> is the model adapter for the <%= name %>.
23  *
24  * By implementing this interface, you can adapt your model to <%= name %>.
25  */
26 public interface <%= className %> extends IContentProvider {
27     /*** Structural Content ***/
28     <% Set<EClass> classes = context.getClasses(rootClass); %>
29     <% for( EClass eClass : classes ) { %>
30       <% List<EReference> references = context.getContainmentReferences(eClass); %>
31       <% for ( EReference reference : references ) { %>
32         <% if ( context.getCardinality(reference) == GenContext.CARDINALITY_MANY  ) { %>
33     public Object[] <%= context.createMethodName(reference) %> (Object <%=context.
           createParamName(reference)%> ) ;
34         <% } else { %>
35     public Object <%= context.createMethodName(reference) %> (Object <%=context.
           createParamName(reference)%> ) ;
36         <% } %>
37       <% } %>
38     <% } %>
39
40     /*** Non-Structural Content ***/
```

```
41   <% for( EClass eClass : classes ) { %>
42     <% List<EReference> references = context.getNonContainmentReferences(eClass); %>
43     <% for ( EReference reference : references ) { %>
44       <% if ( context.getCardinality(reference) == GenContext.CARDINALITY_MANY  ) { %>
45     public Object[] <%= context.createMethodName(reference) %> (Object <%=context.
           createParamName(reference)%> ) ;
46       <% } else { %>
47     public Object <%= context.createMethodName(reference) %> (Object <%=context.
           createParamName(reference)%> ) ;
48       <% } %>
49     <% } %>
50   <% } %>
51
52     /*** Labels and Data Content ***/
53   <% for( EClass eClass : classes ) { %>
54     <% List<EAttribute> attributes = context.getAttributes(eClass); %>
55     <% for ( EAttribute attribute : attributes) { %>
56       <% if ( context.getCardinality(attribute) == GenContext.CARDINALITY_MANY  ) { %>
57     public <%= context.getReturnTypeNonBuiltIn(attribute)%>[] <%= context.createMethodName
           (attribute) %> (Object <%=context.createParamName(attribute)%> ) ;
58       <% } else { %>
59     public <%= context.getReturnTypeNonBuiltIn(attribute)%> <%= context.createMethodName(
           attribute) %> (Object <%=context.createParamName(attribute)%> ) ;
60       <% } %>
61     <% } %>
62   <% } %>
63 }
64
65 <% context.emitSortedImports(); %>
```

List of Listings B.1: Jet Transformation for generating Content Providers)

```
1
2  <%@ jet
3       package="org.vizmodels.generator.jet.gen"
4       class="GenContentProvider"
5       imports="java.util.*
6            org.eclipse.emf.ecore.EClass
7            org.vizmodels.generator.jet.gen.AdapterGenerator
8            org.eclipse.emf.ecore.EReference
9            org.eclipse.emf.ecore.EAttribute"
10
11      %>
12      <% GenContext context = (GenContext)argument; %>
13      <% context.beginNewFile(); %>
14      <% EClass rootClass = context.getViewer(); %>
15      <% String name = rootClass.getName(); %>
16      <% String className = context.createContentProviderClassName(rootClass); %>
17  package <%= context.createPackageName(rootClass) %>;
18
19  import org.eclipse.jface.viewers.IContentProvider;
20  <% context.markImportLocation(stringBuffer); %>
21
22  /**
23   * <%= className %> is the model adapter for the <%= name %>.
24   *
25   * By implementing this interface, you can adapt your model to <%= name %>.
26   */
27  public interface <%= className %> extends IContentProvider {
28      /*** Structural Content ***/
29      <% Set<EClass> classes = context.getClasses(rootClass); %>
30      <% for( EClass eClass : classes ) { %>
31        <% List<EReference> references = context.getContainmentReferences(eClass); %>
32        <% for ( EReference reference : references ) { %>
33          <% if ( context.getCardinality(reference) == GenContext.CARDINALITY_MANY  ) { %>
34      public Object[] <%= context.createMethodName(reference) %> (Object <%=context.
            createParamName(reference)%> ) ;
35          <% } else { %>
36      public Object <%= context.createMethodName(reference) %> (Object <%=context.
            createParamName(reference)%> ) ;
37          <% } %>
38        <% } %>
39      <% } %>
40
```

```
41      /*** Non-Structural Content ***/
42    <% for( EClass eClass : classes ) { %>
43      <% List<EReference> references = context.getNonContainmentReferences(eClass); %>
44      <% for ( EReference reference : references ) { %>
45        <% if ( context.getCardinality(reference) == GenContext.CARDINALITY_MANY  ) { %>
46      public Object[] <%= context.createMethodName(reference) %> (Object <%=context.
            createParamName(reference)%> ) ;
47        <% } else { %>
48      public Object <%= context.createMethodName(reference) %> (Object <%=context.
            createParamName(reference)%> ) ;
49        <% } %>
50      <% } %>
51    <% } %>
52
53      /*** Labels and Data Content ***/
54    <% for( EClass eClass : classes ) { %>
55      <% List<EAttribute> attributes = context.getAttributes(eClass); %>
56      <% for ( EAttribute attribute : attributes) { %>
57        <% if ( context.getCardinality(attribute) == GenContext.CARDINALITY_MANY  ) { %>
58      public <%= context.getReturnTypeNonBuiltIn(attribute)%>[] <%= context.createMethodName
            (attribute) %> (Object <%=context.createParamName(attribute)%> ) ;
59        <% } else { %>
60      public <%= context.getReturnTypeNonBuiltIn(attribute)%> <%= context.createMethodName(
            attribute) %> (Object <%=context.createParamName(attribute)%> ) ;
61        <% } %>
62      <% } %>
63    <% } %>
64  }
65
66  <% context.emitSortedImports(); %>
```

List of Listings B.2: Jet Transformation for Synchronizing Content Providers)

APPENDIX C

ATL Transformations

WHILE all the ATL transformations presented in this dissertation are accurate, occasionally name spaces were truncated, or uninteresting rules removed for readability. The transformations listed below are the complete ATL transformations used to produce the all of our interactive visualizations.

```
1  module HOCKEY2View; -- Module Template
2  create Output : VizModel from Input : Hockey;
3
4  rule r1 {
5          from  s : Hockey!Player (
6                          s.name = 'Wayne Gretzky'
7          )
8          to t : VizModel!SimpleBarChartViewer (
9                  chartTitle <- 'Wayne Gretzky\'s Goals by Year',
10                 xAxisTitle <- 'Goals',
11                 yAxisTitle <- 'Years',
12                 showDepth <- true
13         )
14         do {
15                 t.items <- s.yearlyStats->collect( e | thisModule.Item(e) );
16                 t;
17         }
18 }
19
20 rule Item (s : Hockey!PlayerYearlyStats) {
21         to t : VizModel!SimpleItem (
22                         label <- s.year.year,
23                         value <- s.goals
24                 )
25         do {
26                 t;
27         }
28 }
```

List of Listings C.1: ATL Transformation: Hockey Model to Bar Chart (from Section 7.4)

```
1  module hockey2treemap; -- Module Template
2  create Output : VizModel from Input : Hockey;
3
4  helper def: treemapView: OclAny = OclUndefined;
5
6  entrypoint rule start() {
7          to t : VizModel!TreeMapViewer
8          do {
9                  thisModule.treemapView <- t;
10         }
11 }
12
13 rule r1 {
14         from  s : Hockey!Year
15         to t : VizModel!TreeMapItem (
16                 label <- s.year,
17                 value <- s.playersStats->collect( e | e.goals )->sum() + 0.0,
18                 children <- s.playersStats
19         )
20         do {
21                 thisModule.treemapView.children <- t;
22                 t;
23         }
24 }
25
26 rule r2 {
27         from s : Hockey!PlayerYearlyStats (
28                 s.goals > 0
29         )
30         to t : VizModel!TreeMapItem (
31                 label <- s.name,
32                 value <- s.goals +0.0
33         )
34 }
```

List of Listings C.2: ATL Transformation: Hockey Model to TreeMap (from Section 7.4.1)

```
1  module snapTest; -- Module Template
2  create Output : VizModel from Input : Hockey;
3
4  helper def: listView : OclAny = OclUndefined;
5
6  entrypoint rule start() {
7          to t : VizModel!ListViewer
8          do {
9                  thisModule.listView <- t;
10         }
11 }
12 rule r1 {
13         from s : Hockey!Player
14         to   t : VizModel!ListItem (
15                         label <- s.name
16                 ),
17         b : VizModel!SimpleBarChartViewer (
18                 items <- s.yearlyStats,
19                 chartTitle <- s.name +'\'s Points'
20         ),
21         m : VizModel!MapViewer (
22                 markers <- s.yearlyStats->collect( e | e.teams )->flatten()->asSet()->
                       collect( f | thisModule.Marker( f ) )
23         )
24
25         do {
26                 thisModule.listView.items <-t;
27                 t.snapPoint <- thisModule.Snap(
28                         Tuple{ reference =b,
29                                 selection = OclUndefined });
30                 t.snapPoint <- thisModule.Snap( Tuple{reference = m, selection =
                       OclUndefined} );
31
32         }
33 }
34
35 lazy rule Marker {
36         from s : Hockey!Team
37         to t : VizModel!Marker (
38                 label <- s.name
39         )
40 }
41
42 rule BarItem {
43         from s : Hockey!PlayerYearlyStats
44         to t : VizModel!SimpleItem (
45                 label <- s.year.year,
46                 value <- s.goals + s.assists
47         )
48 }
49
50 lazy rule Snap {
51   from a : TupleType(reference:MOF!EObject, selection:MOF!EObject)
52   to s : VizModel!Snap (
53     snapTo<-a.reference
54   )
55 }
```

List of Listings C.3: ATL Transformation: SnapPoint Transformatoin (from Section 7.5)

APPENDIX D

Evaluation Transformations

In this appendix, we present all the ATL transformations used in our evaluation (Chapter 8). This includes transformations for:

- Transformation for creating the plug-in dependency view (Listing D.1);
- Transformation for highlighting the shortest path between two nodes (Listing D.2);
- Transformation for generating the member-area view (Listing D.3); and
- Transformation for extending the member-area view (Listing D.4)

```
1  module PDEView; -- Create Plugin View
2  create OUTPUT : vizmodel from INPUT : pdemodel;
3  helper def: graphView: OclAny = OclUndefined;
4
5  rule r1 {
6          from s: pdemodel!Root
7          to t: vizmodel!GraphViewer
8          do {
9                  thisModule.graphView <- t;
10         }
11 }
12
13 rule r2 {
14         from s: pdemodel!BundleDescription (
15                 s.symbolicName = INPUT_PLUGIN
16         )
17         to t : vizmodel!GraphNode (
18                 label <- s.symbolicName,
19                 colour <- #Green,
20                 outgoing <- s.requiredBundles->collect( e | thisModule.connection ( e) )
21         )
22
23         do {
24                 thisModule.graphView.nodes <-t;
25         }
26 }
27
28  lazy rule connection {
29         from s : pdemodel!BundleSpecification
30         to t : vizmodel!GraphEdge (
31                 dest <- thisModule.r3(s.descriptor)
32         )
33
34         do {
35                 thisModule.graphView.edges <-t;
36         }
37 }
38
39 unique lazy rule r3 {
40         from s: pdemodel!BundleDescription
41         to t : vizmodel!GraphNode (
42                 label <- s.symbolicName,
```

```
43                  outgoing <- s.requiredBundles->collect( e | thisModule.connection ( e) )
44          )
45          do {
46                  thisModule.graphView.nodes <-t;
47          }
48  }
```

List of Listings D.1: ATL Transformation: Create Plug-in Dependency View

```
1  module ShortestPath; -- Module Template
2  create OUTPUT : vizmodel from INPUT : vizmodel;
3
4
5  helper def: computePreviousPath( previous: Map(vizmodel!GraphNode, Integer), dest:
       vizmodel!GraphNode ) :
6          Sequence(vizmodel!GraphNode) =
7          if previous.get(dest) = OclUndefined then
8                  Sequence{dest}
9          else
10                 thisModule.computePreviousPath( previous, previous.get(dest) )->union(
                       Sequence{dest})
11         endif;
12
13
14 rule shortestPath(nodes : vizmodel!GraphNode, src: vizmodel!GraphNode, dest:vizmodel!
       GraphNode) {
15         using {
16                 dValues : Map(vizmodel!GraphNode, Integer) = Map{};
17                 previous : Map(vizmodel!GraphNode, Integer) = Map{};
18         }
19         do {
20                 for ( n in nodes ) {
21                         if ( n = src ) {
22                                 dValues <- dValues.union(Map{(n, 0)});
23                         }
24                         else {
25                                 dValues <- dValues.union(Map{(n, 100000)});
26                         }
27                 }
28                 for ( head in nodes ) {
29                         for ( edge in head.incoming ) {
30                                 if ( dValues.get(head) + 1 < dValues.get(edge.src) ) {
31                                         previous <- previous.union(Map{(edge.src, head)});
32                                         dValues <- dValues.union(Map{(edge.src, dValues.
                                               get(head) + 1)});
33                                 }
34                         }
35                 }
36                 thisModule.computePreviousPath(previous, dest);
37         }
38 }
```

```
39
40 helper def: createQueue(src: vizmodel!GraphNode, dest:vizmodel!GraphNode) :
41         OrderedSet(vizmodel!GraphNode) =
42                         let
43                         subPath : OrderedSet(vizmodel!GraphNode) = src.incoming->iterate(
                                edge ; elements : OrderedSet(vizmodel!GraphNode) = OrderedSet
                                {} |
44                                         elements->union(thisModule.createQueue(edge.src,
                                                dest))
45                         )
46                         in
47                         if src = dest
48                         then
49                                 OrderedSet{dest}
50                         else
51                                 if subPath->size() > 0
52                                 then
53                                         (OrderedSet{src})->union(subPath)
54                                 else
55                                         OrderedSet{}
56                                 endif
57                         endif;
58
59 helper def : getNode(s: Set(vizmodel!GraphNode), name: String) :
60         vizmodel!GraphNode =
61                 s->select( p |p.label = name)->first();
62
63 helper def: highlight : Set(vizmodel!GraphNode) = Set{};
64
65 rule r1 {
66         from s:  vizmodel!GraphViewer
67         using {
68                 dest : vizmodel!GraphNode = thisModule.getNode(vizmodel!GraphNode.
                        allInstances(), 'org.eclipse.mylyn.pde.ui');
69                 src : vizmodel!GraphNode = thisModule.getNode(vizmodel!GraphNode.
                        allInstances(), 'org.eclipse.text');
70         }
71         to t: vizmodel!GraphViewer (
72                 nodes <- s.nodes,
73                 edges <- s.edges
74         )
75         do {
```

```
76                  thisModule.highlight <- thisModule.shortestPath(thisModule.createQueue(src
                        ,dest)->asSequence(),src,dest);
77          }
78  }
79
80  rule r2 {
81          from s : vizmodel!GraphNode
82          to t: vizmodel!GraphNode (
83                  label <- s.label,
84                  colour <- if thisModule.highlight->includes(s)
85                                          then
86                                                  #Red
87                                          else
88                                                  #Default
89                                          endif
90          )
91  }
92
93  rule r3 {
94          from s : vizmodel!GraphEdge
95          to t: vizmodel!GraphEdge (
96                  src <- s.src,
97                  dest <- s.dest,
98                  colour <- if thisModule.highlight->includes(s.src) and thisModule.
                        highlight->includes(s.dest)
99                          then
100                                 #Red
101                         else
102                                 #Default
103                         endif
104
105         )
106 }
```

List of Listings D.2: ATL Transformation: Shortest Path Transformation

```
1  module MemberAreaView; -- Member Area View
2  create OUTPUT : vizmodel from INPUT : shrimpbib;
3
4  helper def: graphView: OclAny = OclUndefined;
5
6  rule r1 {
7          from s: shrimpbib!User
8          to t: vizmodel!GraphNode (
9                  label <- s.nick
10         ),
11     edges: distinct vizmodel!GraphEdge foreach ( e in s.documents ) (
12                 src <- s,
13                 dest <- e
14         )
15         do {
16                 thisModule.graphView.nodes <- t;
17                 thisModule.graphView.edges <- edges;
18         }
19 }
20
21 rule area {
22         from s: shrimpbib!Area
23         to t:vizmodel!GraphNode (
24                 label <- s.name
25         )
26         do {
27                 thisModule.graphView.nodes <- t;
28         }
29 }
30
31 rule r2 {
32         from s: shrimpbib!Document
33         to t: vizmodel!GraphNode (
34                 label <- s.title,
35                 fisheyeNode <- true,
36                 hideText <- true
37         ),
38         edges : distinct vizmodel!GraphEdge foreach( e in s.area) (
39                         src <- s,
40                         dest <- e,
41                         colour <-#Red
42         )
```

```
43          do {
44                  thisModule.graphView.nodes <- t;
45                  thisModule.graphView.edges <- edges;
46          }
47  }
48
49  entrypoint rule r2() {
50          to t: vizmodel!GraphViewer
51          do {
52                  thisModule.graphView <- t;
53          }
54  }
```

List of Listings D.3: ATL Transformation: SHriMPBib Member Area View

```
1  module MemberAreaView2; -- Member Area View 2
2  create OUTPUT : vizmodel from INPUT : shrimpbib;
3
4  helper def: graphView: OclAny = OclUndefined;
5
6  rule r1 {
7          from s: shrimpbib!User
8          using  {
9                  user_areas :Sequence(shrimpbib!Area) = s.documents->collect( e | e.area)->
                       flatten();
10                 unique_areas: Set(shrimpbib!Area) = user_areas.asSet();
11         }
12         to t: vizmodel!GraphNode (
13                 label <- s.nick,
14                 colour <-#Orange
15         )
16     edges2: distinct vizmodel!GraphEdge foreach ( e in unique_areas) (
17                 src <- s,
18                 dest <- e,
19                 label <- user_areas->count( e ).toString()
20         )
21         do {
22                 thisModule.graphView.nodes <- t;
23                 thisModule.graphView.edges <- edges2;
24         }
25 }
26
27 rule area {
28         from s: shrimpbib!Area
29         to t:vizmodel!GraphNode (
30                 label <- s.name
31
32         )
33         do {
34                 thisModule.graphView.nodes <- t;
35         }
36 }
37
38 entrypoint rule r2() {
39         to t: vizmodel!GraphViewer
40         do {
41                 thisModule.graphView <- t;
```

```
42              }
43  }
```

List of Listings D.4: ATL Transformation: SHriMPBib Member Area View 2