

Visualization in the Context of Model Driven Engineering

R. Ian Bull
University of Victoria
Victoria, BC
Canada
irbull@cs.uvic.ca

Jean-Marie Favre
LSR-IMAG
University of Grenoble
France
Jean-Marie.Favre@imag.fr

ABSTRACT

Understanding and maintenance of complex information spaces is often supported through visual interfaces. These interfaces must be highly flexible in order to adapt to both the user's role and their current task. Program comprehension tools are one class of tools that make heavy use of information visualization techniques, and Software Engineers use these tools to help understand and maintain software systems. This paper presents a model-driven approach to address the customization requirements of visual user interfaces, and does so in the domain of program comprehension.

1. INTRODUCTION

Understanding large software systems is a difficult task. Over the past 15 years many software visualization tools have been proposed to assist Software Engineers while comprehending large systems. Tools such as Rigi [11] and SHrIMP [14] have been used successfully to browse software systems containing over a million LOC, in part due to their ability to generate interactive views. These tools, based on sophisticated user interfaces with advanced visualization techniques, suffer from a lack of adaptability and flexibility when applied in industrial settings [7]. Often the tools are not able to synchronize the many sources of information that exist for a software system such as source code factbases, requirements information, repositories of documentation, bug reports, and so forth.

It is well known that visual interfaces are most effective when they have been customized for specific users and their needs. If an interface does not display the required information for a given task, or the information displayed does not conform to the user's mental model [12], then the interface is not as useful as it could be.

The software industry requires visualization tools that can be adapted to (1) many different sources of information such as source code, configuration management data, documentation, user requirements, (2) various kinds of visualizations, including source code views, tables, trees, tree maps, graphs, nested-graphs, pie-charts, kiviad-diagrams, and so on, and (3) many different user roles including not only developers but also architects, software testers, members of the quality assurance teams, managers, business architects and so on. Moreover, experience has shown that in order to increase the likelihood of adoption among visualization tools, the tools should not be standalone monolithic applications, but instead they should be integrated in the developer's preferred development environment.

Few visualization tools support the level of customization required to create tailored interfaces, and the tools that do support a high level of customization, do so at the cost of complexity, or like Hy+ [6, 10] they are monolithic and can't be integrated with other tools.

While "coding" is the current way of building visualization tools this method lacks the ease of customization required. This paper advocates for the use of Model Driven Engineering to achieve customizable interfaces by composing small model-driven visualization components. This paper focuses on the use of metamodels and transformations to ease the creation of views from arbitrary models of information. This paper does not address view composition. While the examples in this paper center around program comprehension, the techniques can be generalized to any structured data including knowledge management, business processes, financial data, etc.

The rest of the paper is structured as follows. Section 2 introduces the notion of model-driven visualization. A simple example is presented in Section 3 to illustrate the approach. Section 4 outlines our future work and finally, Section 5 presents some concluding remarks.

2. MODEL DRIVEN VISUALIZATION

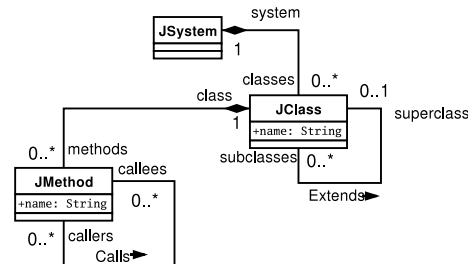


Figure 1: Simple Java MetaModel

2.1 Overview

Model Driven Engineering (MDE) has received a lot of attention in the past few years. More than 50 conferences and workshops have been organized to address the issues related to MDE (see <http://planetmde.org>). This success is due to the fact that MDE is based on very general concepts that can be applied across many different disciplines. The basic set of concept includes Models, Metamodels and Transformations [8]. An in-depth discussion of these concepts is out of the scope of this paper, so these will be illustrated mostly through examples.

Model driven engineering is based on the systematic use of metamodels and transformations. In the context of software visualization, both the information extracted from the software, and information about visualization should be formally defined. While most tools in software engineering are based on implicit metamodels,

it has been recognized recently that information metamodels (also called schemas) should be expressed formally. Figure 1 describes a metamodel for a subset of the Java language.

When building visualization tools, the “visualization metamodel” should also be explicitly defined. If tool is formally defined, then software visualization simply becomes a matter of describing the transformation from one model to another. In the remainder of this section, we present some examples of metamodels and then discusses formal transformations.

2.2 Visualization MetaModels

GUI components designed for visualizing data sets, such as Tree and Table controls, have an implicit **metamodel**. A metamodel formally describes the structure of the information the component can visualize. For example, SHriMP Views [14] a popular nested graph viewer, is excellent at visualizing large hierarchical datasets with non-hierarchical relationships between the nodes. Common visualization tools such as Tree Controls or Space Trees [13] are used extensively to show graph based data in the form of a tree. The metamodels for these components are rarely described formally. The details of how the data must be structured in order to visualize it using one of these tools must be inferred from reading documentation such as *JavaDoc* or *Man Pages*.

Visualization tools are commonly built by researchers as monolithic applications, complete with a parser, application windows, menu system and tool bars. Tools such as these usually have a pre-defined XML schema for the data they can parse. While this provides a formal metamodel, these tools are hard to re-use in existing applications. Since these tools defined their own menu system, tool bars and set of actions, it is difficult to incorporate these visualizations into existing environments. In order to provide a set of reusable components, we advocate that visualization components should adhere to the API specification of the widget toolkit they were implemented with, and provider a similar set of interfaces as the other views in the toolkit. For example, the views should expose item selection, notifications, view instantiation, etc, in the same manner as existing views. For each view, a formal metamodel should also be defined.

Often there are multiple ways to define a data model. In a table view, the table can be defined as several rows, each row containing a fixed number of cells. An equally valid definition would be to define the table as a set of columns, each column containing a fixed number of cells. In order to keep our approach generalizable, several data models have been designed for each component. The implementation of each view will read any model that conforms to one of the pre-defined metamodels. By defining several metamodels for single components, those who use the component can choose the most appropriate metamodel for their data.

2.2.1 Tree View

The Tree View is widely used to present hierarchical data by listing children below their parents. Sub-trees can often be expanded and collapsed. The wide spread adoption of the tree control can partially be attributed to the fact that many widget toolkits include the control as a standard component. Figure 2 shows an example of a Tree View’s metamodel.

2.2.2 Graph View

Graph Views display information as a set of nodes connected with edges to show dependencies among the nodes. Despite the positive

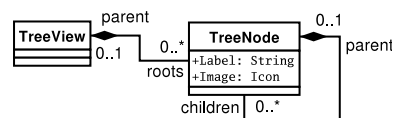


Figure 2: MetaModel for a TreeView

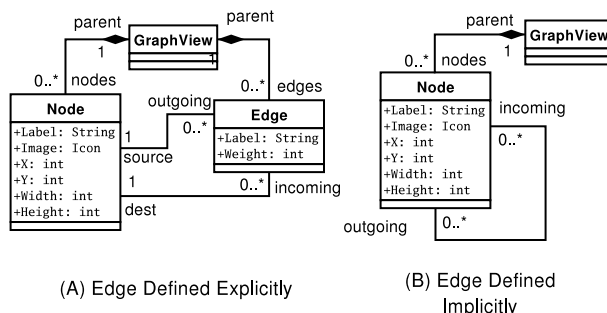


Figure 3: MetaModel for a GraphView

feedback researchers have received from several implementation of graph based views [9], the views have not received wide spread adoption in Common Of the Shelf (COTS) Software. Toolkits exist to help developers create these views [2], however, graph views themselves are not widely available as standard components. The Zest toolkit [5] has been designed to address this problem for SWT, a widget toolkit for Java.

Graph Views can be defined in terms of their nodes and edges or simply defined in terms of their nodes. Using the later approach, the edges can be determined implicitly based on how the nodes are related to one another. An example of each structure is presented in Figure 3.

Metamodels for several other views have also been defined formally. Most notably, metamodels have been defined for Nested Graph Views, Table Views, Matrix Views and Table-Tree Views.

2.3 Visualization Transformations

In order to visualize a partial dataset using one the structures described above, application programmers must develop data traversal algorithms and apply them to their data model. The traversal algorithms are used to walk the data structure, extracting the “interesting” information and using this information to populate the views. While traversing the data structures, some information is discarded and new information is deduced. For example, a common Java model will include interactions between Methods. However, when attempting to understand a section of source code, a developer may wish to see how the classes relate to one another. In order to do this, the method nodes must be discarded and the relationships between them lifted to the Class level. Using a declarative language for these transformations is often more concise, and easier to maintain, then writing the traversal algorithms in source code.

2.4 Techniques

A framework for Model Driven Visualization has been implemented within Eclipse [1]. The models have been designed using EMF [4]. EMF was used because of its ability to generate working code from a model, and because it is able to generate XML serialization / de-

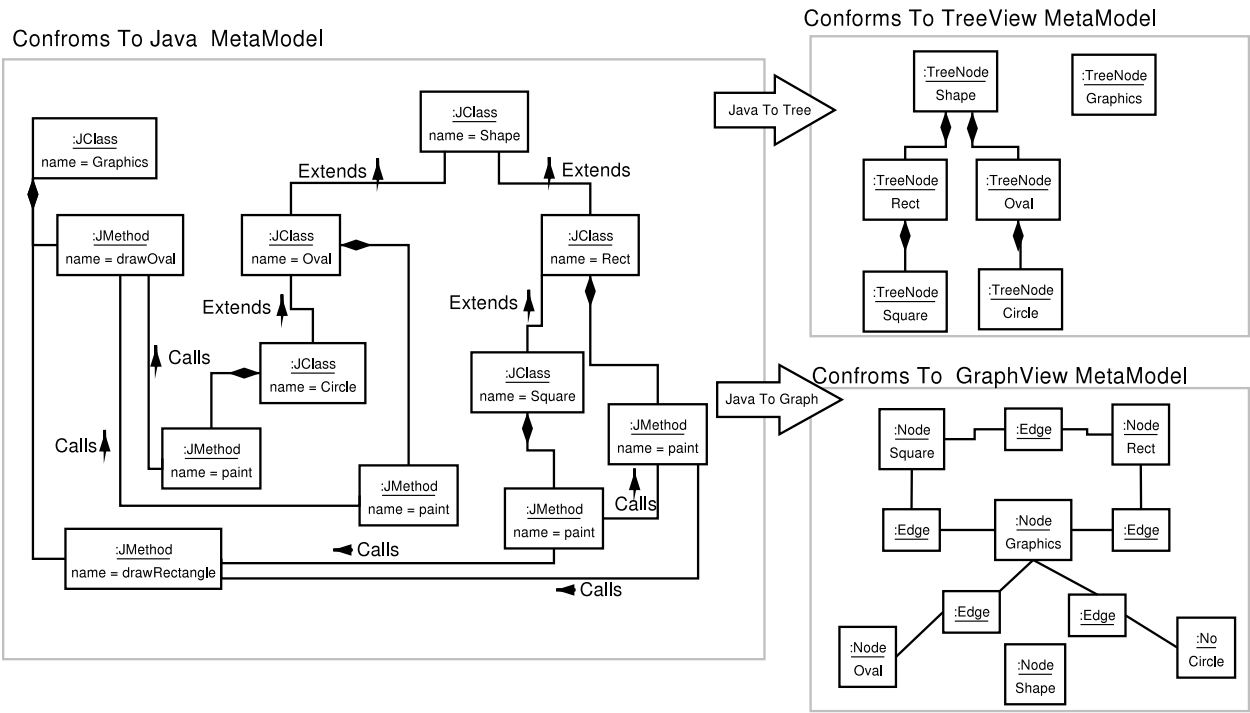


Figure 4: Model Transformation

serialization capabilities and an XSD. Each of the visualizations described in Section 2.2 have also been implemented for Eclipse using SWT. The JFace Tree and Table controls were used for the Tree and Table views, while the Zest project was used for the graphical views.

The Atlas Transformation Language (ATL) [3] was used to specify the transformations. ATL is a transformation language for MDE and is able to translate EMF models using both declarative and imperative constructs. The transformations are described as a set of transformation rules and the ATL virtual machine uses these rules to generate an output model from a given input model.

3. CASE STUDY

In order to demonstrate our approach, we have built a small class browser for Eclipse. Two views were created, one to show the class hierarchy, and the other to show method invocation between classes. A Java metamodel was first defined using EMF, and then using ATL, transformations were written to translate the Java model to a view model. Finally, using our interactive visualization toolkit, the view models were rendered.

The Java metamodel presented earlier (Figure 1) defines the *JSystem*, *JClass* and *JMethod* elements. *JMethods* are related to one another through the *calls* reference and *JClasses* reference each other through the *extendedBy* dependency. The left side of Figure 4 shows a Java model which conforms to the Java metamodel described in Figure 1. The two horizontal arrows represent the model transformations. The two views on the right of Figure 4 show two translated models. The one on top conforms to the TreeView metamodel (Section 2.2.1) and the one below conforms to the GraphView metamodel (Section 2.2.2).

The transformation from the Java Model to the view models is per-

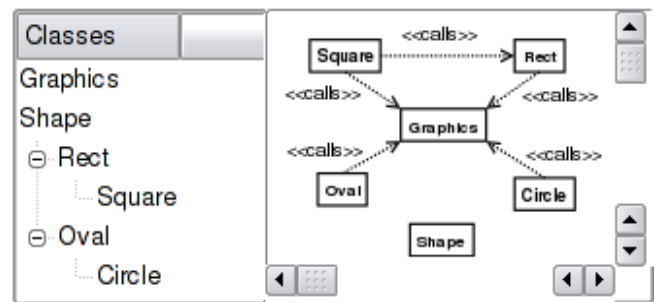


Figure 5: Class Dependency Tool

```

1 rule JSystem2TreeView {
2   from s : java!System
3   to
4     t : treeview!TreeView (
5       roots <- java!JClass.allInstances()->
6         select( superclass->isEmpty() )
7     )
8 }
9
10 rule JClass2TreeNode {
11   from c : java!JClass
12   to
13     n : treeview!TreeNode (
14       label <- c.name,
15       children <- c.extendedBy
16     )
17 }

```

Listing 1: Java to Tree Translation

```

1  rule JSystem2GraphView {
2    from s : java!JSystem
3    to
4      out : graphview!GraphView (
5        entities <- java!JClass.allInstances(),
6        edges<-java!JClass.allInstances()->
7          collect(e|thisModule.resolveTemp(e, 'e'))
8      )
9  }
10 rule JClass2NodeAndEdges {
11   from c : java!JClass
12   to
13     n : graphview!Node (
14       label <- c.name
15     ),
16     e : distinct graphview!Edge
17       foreach(singleCall in c.methods
18         ->collect( m|m.calls )->flatten()) (
19         source <- n,
20         destination <- singleCall.parent
21       )
22 }

```

Listing 2: Java to Graph Translation

formed using ATL. Listing 1 shows the ATL transformation responsible for translating the Java model to a Tree View model (Section 2.2.1). The main rule matches the Java System and creates the `TreeView`'s Root Node. All the classes with no superclass are listed as roots. The other rule matches the `JClass` elements. Each of these elements are translated to `TreeView`'s Node and the Label for each node is simply the Class name. Finally, the children of a given element are determined by *extendedBy* relationship in the model.

The second transformation (Listing 2) shows a translation from the Java model to a Graph Model (Section 2.2.2). The Graph Model shows dependencies among classes if there is a *call* relationship between them. The `JSystem2GraphView` rule matches the Java System and creates a `GraphView` Root Node. The edges and entities of the graph are attached to the *GraphView*. The second rule creates all the *Nodes* and *Edges*. Each node directly maps to a `JClass` element in the Java model. For each method call an edge is created. The edge is lifted to the `JClass` level such the *source* of the edge is the `JClass` that contains the *JMethod*, and the destination is the `JClass` of the *JMethod* being invoked.

Once the transformations is completed, our Visualization Toolkit is used to read the view models and present the user with the Class dependencies. Figure 5 shows the output of this tool.

4. FUTURE WORK

In the interest in saving space, only simple examples have been presented here. Model-based visualization components are commonly more sophisticated. Our approach has been used to generate views using charts, interactive graphs, interactive nested graphs, time-lines, spectrographs, interactive tree-maps and others. We are currently continuing our work by creating a complete library of visualization metamodels and model-driven visualization components. The metamodels will be included in Zoomm, the International Zoo of Metamodels, Schemas and Grammars. Model-based visualization components are also to being developed for integration within Eclipse using the EMF framework.

5. CONCLUSION

In this paper, Model Driven Engineering techniques have been applied in the context of software visualization tools. The techniques presented here are not only applicable to software, but can be generalized to any information space. While GUI components found in traditional widget toolkits are well suited for visualizing small to medium, rather "flat" information, visualization tools are very good at exploring large datasets with complex structures. This paper shows how metamodels can be used for describing the structure of the visualization tool. Designing these metamodels is a difficult task, but once these models have been created, the explicit mappings between datasets and the model-based visualization components can be easily described. The work in this paper focusses on single visualization components, although we realize that software exploration requires combining and synchronizing multiple views. Clearly, sophisticated exploration environments in the future will only be achieved by integrating know-how from Software Exploration community, Model Driven Engineering community and the User Interface community.

6. REFERENCES

- [1] Eclipse. <http://www.eclipse.org>.
- [2] Graphical Editor Framework. Website. <http://www.eclipse.org/gef>.
- [3] 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)*, Canterbury, England, 2004. Computing Laboratory, University of Kent, Canterbury, Kent CT2 7NF, UK.
- [4] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework*. Addison Wesley, 2003. <http://www.eclipse.org/emf>.
- [5] R. Ian Bull, Casey Best, and Margaret-Anne Storey. Advanced Widgets for Eclipse. In *Proceedings of the 2nd Eclipse Technology Exchange*, pages 6–11, 2004.
- [6] Mariano Consens, Alberto Mendelzon, and Arthur Ryman. Visualizing and Querying Software Structures. In *Proc of International Conference on Software Engineering*, 1992.
- [7] J.-M Favre, F Duclos, J Estublier, R. Sanlaville, and J.-J. Auffret. Reverse engineering a large component-based software product. In *Proc. of Fifth European Conference on Software Maintenance and Reengineering*, pages 95–104, Lisbon, Portugal, 2001.
- [8] J.M. Favre. Towards a basic theory to model model driven engineering. In *Workshop on Software Model Engineering (WISME)*, Lisboa, Portugal, 2004.
- [9] Ivan Herman and Guy Melançon adn M. Scott Marshall. Graph Visualization and Navigation in Information Visualization: A Survey. *IEEE Transactions on Visualization and Computer Graphics*, 6:24–43, 2000.
- [10] Alberto Mendelzon and Johannes Sameting. Reverse Engineering by Visualizing and Querying. *Software – Concepts and Tools*, 16:170–182, 1995.
- [11] Hausi A. Müller and Karl Klashinsky. Rigi: A system for programming-in-the-large. In *Proc. of the 10th Intl. Conference on Software Engineering (ICSE-10)*, pages 80–86, Singapore, April 1988.
- [12] Donald A. Norman. *The Design of Everyday Things*. New York: Basic Books, 1998.
- [13] C. Plaisant, J. Grosjean, and B.B. Bederson. Spacetree: Supporting exploration in large node link tree, design evolution and empirical evaluation. In *Proc. Of INFOVIS 2002. IEEE Symposium on Information Visualization, 2002*, pages 57–64, Boston, October 2002.
- [14] M.-A. D. Storey and H. A. Müller. Manipulating and documenting software structures using SHriMP views. In *Proceedings of the 1995 International Conference on Software Maintenance (ICSM '95) (Opio (Nice), France, October 16-20, 1995)*, 1995.