Using Recovered Views to Track Architectural Evolution

Tamar Richner
Software Composition Group, Institut für Informatik (IAM)
Universität Bern, Neubrückstrasse 10, 3012 Berne,Switzerland
richner@iam.unibe.ch, http://www.iam.unibe.ch/∼richner/

presented at ECOOP '99 Workshop on OO Architectural Evolution

1 Introduction

Tracking the evolution of a software system through time gives us valuable information. It suggests which parts are likely to remain stable and which 'problem' aspects are likely to change, and it gives us insight into some of the design choices made. In this paper we show how recovered views of succesive versions of the same software system can be used to track evolution. We first briefly describe our approach for recovering views of software applications. We then compare views of two versions of the HotDraw framework. Our objective is to illustrate a number of issues concerning architectural evolution: what is architectural change as opposed to change in general? how can we detect architectural drift? how can we evaluate the relative quality of different architectural solutions? what are guidelines for building evolvable software?

2 Recovering Architectural Views

In order to recover architectural views of an application we use both static and dynamic information, represented as Prolog facts, then analyze this information using Prolog rules in order to create views which show the invocation behavior of the application at different levels of granularity. Figure 1 shows the process of view recovery. Below, we briefly summarize our approach - a more detailed description can be found in [RD99].

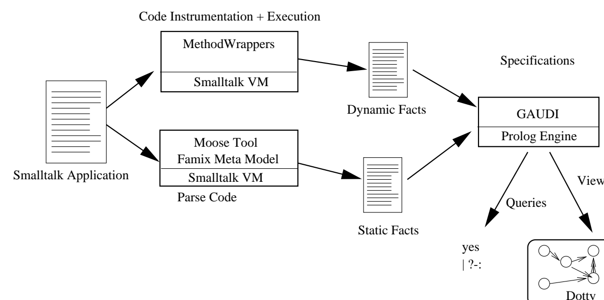

Figure 1: View Recovery from Smalltalk Applications

To obtain static information the source code is parsed and represented in a meta-model for OO software[TD98], developed as part of the Famoos[1] project. This model is then represented as Prolog facts, providing information about classes, methods and the inheritance structure of the application. To obtain dynamic information, Smalltalk applications are first instrumented using Method Wrappers[BFJR98], then executed, producing tracing information in the form of Prolog facts. This information represents the method invocations which occur in the execution of the application. The static and dynamic information is then analyzed using Gaudi, which consists of a set of Prolog rules defining relations and clusters used to create the views (see section 2.1), and of functions to generate views. The dot tool [KN] is used to display the recovered views.

2.1 Specifying Views

To specify a view we use Prolog rules to define: (1) a clustering to components C, and (2) a relation R which specifies whether or not a certain relationship holds between two elements,

For example, to specify a view which shows the invocation relationships between Smalltalk class categories, we define C and R as follows:

```
C allInCategory(Category,ListOfClasses) :-
        setof(Class,class(Class,Category),ListOfClasses).

R invokes(Class1,Class2) :-
        send(_,_,Class1,Instance1,Class2,Instance2,Method).
```

The C rule clusters together all classes which are in the same Smalltalk category, and the R rule defines a relationship between two classes which holds if at least one instance of Class1 invokes a method on at least one instance of Class2. To generate a view that shows the invocations (from an executed scenario) between the Smalltalk class categories of the HotDraw framework we then invoke the query: createView(invokesClass,allInCategory), which generates the views displayed in Figure 2: each node in the graph corresponds to a HotDraw class category and each directed edge $A \rightarrow B$ means that at least one instance of a class in category A invokes a method on an instance of a class in category B. Note that the left hand figure corresponds to HotDraw version 2.5 and the right hand figure to version 3.0.

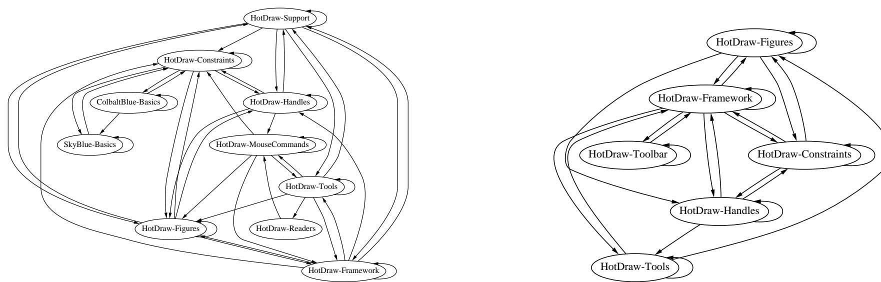

Figure 2: Invocations between categories

This view gives us a coarse idea of the communication between parts of the Hot-

[1]The FAMOOS ESPRIT project investigates tools and techniques for transforming object-oriented legacy systems into frameworks. See *http://www.iam.unibe.ch/∼famoos/*

Draw framework. However, since the category HotDrawFramework groups together some of the main classes, this clustering must be broken down to get a better understanding of the behavior of the application.

3 Tracking Evolution: Case Study of HotDraw

HotDraw [BJ94][Bra95] is a framework for semantic graphics editors. It allows for the creation of graphical editors which associate the picture with a data structure - that is, changing the picture also changes the data structure. The HotDraw framework consists of 114 Smalltalk classes and comes with several sample editors. Several changes have been made to the framework over the years, in particular to the implementation of tools and of constraints.

From the documentation we know that HotDraw is based on the Model-View-Controller triad [KP88]: these roles being played by a *drawing*, *drawing editor* and *drawing controller* respectively. It also has a few other basic concepts: *tools* are used to manipulate the drawing which consists of *figures* accessed through *handles*. *Constraints* are used to ensure that certain invariants are met, for example, that two figures connected with a line remain connected if one of the figures is moved.

3.1 Comparing Versions of HotDraw

We compared two versions of HotDraw, version 2.5 and version 3.0, in order to see how the design has evolved. To do this we first obtained static information for each of these versions. To obtain dynamic information we ran the same short scenario on the Drawing Editor application. The scenario we ran consisted of opening the Drawing Editor application, creating a rectangle figure, creating an ellipse figure, selecting both and grouping them, ungrouping them, then closing the Drawing Editor.

Figure 2 gives us an initial view of the communication between the class categories. It shows that us that version 3.0 no longer uses the ColbaltBlue and SkyBlue constraint solvers, and also seems to have no MouseCommands or Readers. Indeed, from some short documentation about the changes made from version 2.5 to version 3.0 we learn that the contraint implementation has changed: instead of using a general constraint solver, it is now implemented through the Smalltalk dependency mechanism. We also learn that the implementation of tools has changed: readers and commands have disappeared, and tools are now implemented as state machines.

1. Matching domain concepts. Though evidently there have been many changes, does this mean that the architecture has changed? We create new views in which we define one component for each of the main domain concepts of HotDraw. For each version we thus define the following components: Drawing, DrawingController, DrawingEditor, Figure, Handle, Tool, Toolbar (ToolPalette in version 2.5) and Constraints. The mapping of classes to components is different for each version: in version 2.5 the tool concept is implemented by classes for Commands, Readers and Tools, whereas in version 3.0 it is implemented by the classes Tool, ToolState and TransitionTable. We then obtain views, see Figure 3, which show how these components communicate.

Matching domain concepts leads to two similar, though not identical graphs. For example, in version 2.5 ToolPalette communicates with Tools, whereas in version 3.0 Toolbar does not communicate with Tools.

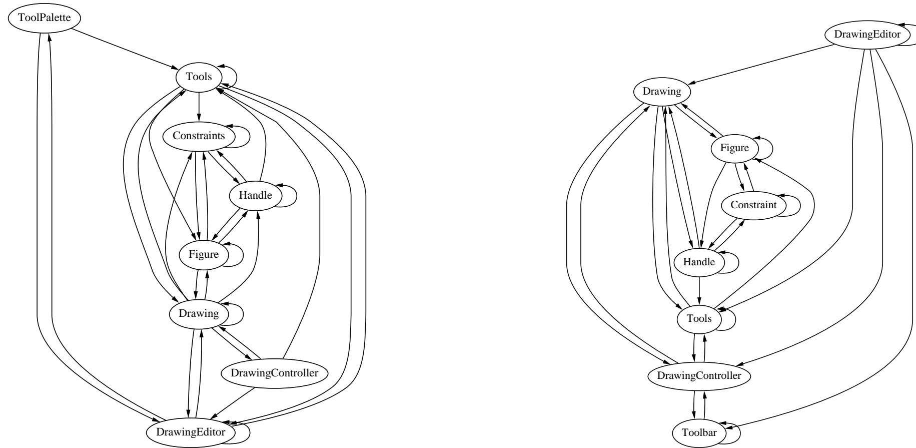

Figure 3: Invocations between components

2. Evaluating architectural quality. We create a view in which noncreation sends are displayed between classes. Since in version 2.5 many more classes are used to implement constraints, we collapse all the constraint related classes to one Constraint component. We want to illustrate schematically the relative complexity of the two versions, see Figure 4.

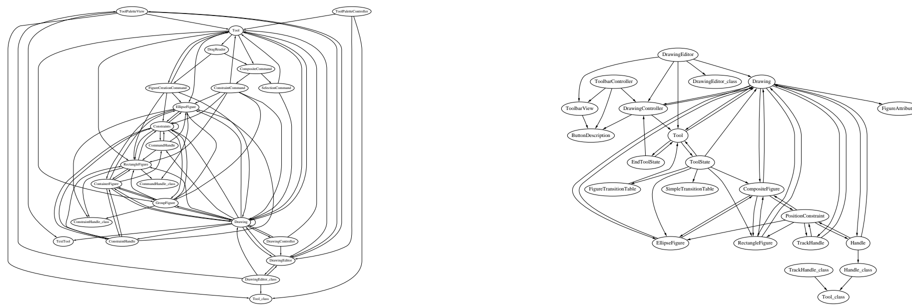

Figure 4: Non-creation invocations between classes

Compared to the graph for version 3.0, the one for version 2.5 is a mess. Can the structure of such invocation graphs be used to evaluate the quality of a certain architecture, given that they both have the same functionality?

3. Comparing roles. In each version the tool concept is implemented differently - but has the interface changed? We ask about the public interfaces of the tool component, see Figure 5.

We see that version 2.5 has three methods that relate tool to its icon image toolHiLiteSize, toolIconSpacing and toolImageSize, which don't seem to have a correspondence in version 3.0. This reflects the connection between ToolPalette and Tools in version 2.5 (see Figure 3), which is absent in version 3.0.

Figure 5: Public interface of tool component

4 Discussion

In this paper we have briefly sketched how views recovered from different versions of an application can be used to track architectural evolution. Our objective, is however, more to raise some discussion issues concerning architectural evolution.

The purpose of tracking evolution. In reverse engineering an application, tracking the evolution of the software helps us to understand what parts of the software presented problems, and may give insight into design rationale. But tracking evolution also has predictive value. For example, if we would have included still an earlier version of HotDraw in our comparison (version 1.0) we would have seen that the tool concept had yet another implementation. Thus, looking at several versions, we can better predict which aspect of a software system will remain stable, and which is likely to change. Although the application we have studied here is itself a framework, it is clear from the discussion above that tracking the evolution of an application or comparing similar applications is useful in the process of evolving a framework.

The use of rules to detect architectural drift. Rules can be used to codify invariants which should hold in all versions of the software. But interesting architectural invariants are not easy to express with the fine-grained elements of the static and dynamic model. We are currently looking at using dynamic information to identify class collaborations (as in UML); such collaborations could be expressed in rules as invariants [Luc97]. It is much easier to codify rules of programming style [Wuy98] or structural design patterns [KP96] than to codify architectural patterns [BMR+96].

A model for architectural evolution and drift. Our case study suggests that although algorithms changed (as for constraint solving) and patterns used change (as for tools) the domain concepts and their roles remained quite stable. This suggests that the software be viewed as layers which evolve at different rates. Domain concepts and roles (often refered to as the metaphor of the application) remain the most stable. Patterns used change more often: patterns are often solutions for maintaining roles in a flexible way in the face of certain forces. Algorithms used are even more volatile. This layering is not an architectural style [Gal98] - it is a way to think about the software. It can also be a way to package the software, as possible with Java interfaces.

Evaluating the quality of an architecture In this paper we compared views of an application for a specific execution scenario, and saw that the views of HotDraw version 3.0 was simpler than that for HotDraw version 2.5. This raises the question of

how to evaluate the relative merits of one arhcitecture over another. What are good measurements for comparing versions, for comparing maintainability and flexibility of the software?

Building evolvable software. Can we learn something from tracking software evolution, such as guidelines for building applications which evolve gracefully? [RJ98].

Acknowledgements This work has been funded by the Swiss Government under Project no. NFS-2000-46947.96 and BBW-96.0015 as well as by the European Union under the ESPRIT programme Project no. 21975.

References

[BFJR98] J. Brant, B. Foote, R. E. Johnson, and D. Roberts. Wrappers to the rescue. In *Proceedings of ECOOP'98*, LNCS 1445, pp. 396–417, 1998.

[BJ94] K. Beck and R. Johnson. Patterns generate architectures. In *Proceedings ECOOP'94*, LNCS 821, pp. 139–149. Springer-Verlag, July 1994.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P.Sommerlad, and M. Stahl. *A System of Patterns*. J. Wiley & Sons, 1996.

[Bra95] J. Brant. Hotdraw, 1995. Master's Thesis, University of Illinois.

[Gal98] G. H. Galal. A note on object-oriented software architecting. In *ECOOP'98 Workshop Reader*, LNCS 1543, pp. 46–47, 1998.

[KN] E. Koutsofios and S. C. North. *Drawing graphs with dot*. AT & T Bell Laboratories, Murray Hill, NJ.

[KP88] G. E. Krasner and S. T. Pope. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *JOOP*, pp. 26–49, Aug. 1988.

[KP96] C. Kramer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Proceeding of WCRE '96*. IEEE, Nov. 1996.

[Luc97] C. Lucas. *Documenting Reuse and Evolution with Reuse Contracts*. PhD thesis, Vrije Universiteit Brussel, 1997.

[RD99] T. Richner and S. Ducasse. Recovering high-level views of object-oriented applications by combining static and dynamic information. In H. Yang and L. White, editors, *Proceedings ICSM'99*, pp. –pages yet unknow. –publisher yet unknown, Sept. 1999.

[RJ98] D. Roberts and R. Johnson. Patterns for evolving frameworks. In R. Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design 3*, pp. 471–486. Addison-Wesley, 1998.

[TD98] S. Tichelaar and S. Demeyer. An exchange model for reengineering tools. In *ECOOP'98 Workshop Reader*, LNCS 1543, 1998.

[Wuy98] R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of TOOLS USA '98*. IEEE Computer Society Press, Aug. 1998.