Describing Framework Architectures: more than Design Patterns

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/

Abstract

In this paper we argue for the necessity of an *architectural* description of a framework. We then analyze why design patterns on their own are insufficient for such a description and propose that a variety of complementary forms of documentation are needed to address the requirements of an architectural description. We claim that traditional artifacts of domain analysis and object-oriented design can better capture the architecture of a framework by describing the *design* solutions in the problem *context* at a higher level of granularity than can design patterns.

1 Introduction

More than other software systems, frameworks are intended for extension and reuse; it is therefore essential to describe and document them. Framework cookbooks are now accepted as a necessary form of documentation which enables users to build applications using the framework without requiring a thorough understanding of the framework itself. But what about describing framework *architectures*: since "nobody understands a framework until they have used it" [Joh92], do we need to describe them at all? There are several reasons why such a description is required.

'selling' the framework. Prospective users of the framework want a general understanding of the framework in order to decide if it is appropriate for their needs. This kind of description should show the overall design and the points of variability of the framework.

reuse of architecture. Transmitting a language-independent view of the architecture allows the high-level design of the framework to be reused in implementing it in other languages, or in modifying it for use in other domains.

integration of frameworks. In order to facilitate the construction of systems from several existing frameworks, the architectural assumptions of each framework should be made explicit [GAO95][MB97].

evolution and re-engineering. Having an architectural description of a framework gives us a reference against which we can measure the changes in subsequent versions of the framework. In the same way, the ability to describe the architecture of an application allows us to form hypotheses about the architecture which can be tested in the process of reverse engineering [MN97].

An architectural description of a framework serves a different purpose than a "cookbook" [Joh92] for framework users, or a detailed documentation of the source code to be

used by developers and maintainers. But it is complementary to these and, ideally, can form the basis for them. In this position paper we argue that design patterns by themselves are not sufficient for communicating an architectural view of a framework, and that indeed, no *one* vehicle can fulfill all the requirements for an architectural description outlined above.

2   Using Design Patterns to Describe Architectures

Design patterns [GHJV95] [BMR+96] are touted as a good vehicle for describing software architectures. Because each design pattern presents a solution to a specific design problem, describing the design patterns in a framework can give readers an understanding of the problem that the current design is intended to solve [BJ94].

There are, however, several problems with using design patterns to document software architectures. Consider figure 1, giving a schematic representation of some of the main classes in the HotDraw framework [BJ94] [Joh92], and the design patterns in which these classes participate.
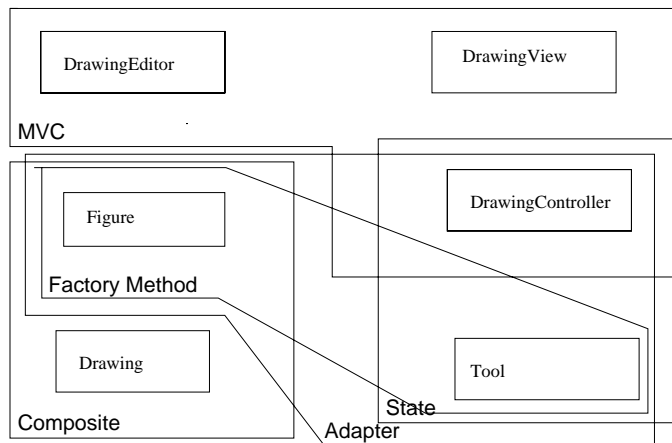

Figure 1: (Some of the) Design Patterns in HotDraw.

Such a diagram, where we omit for simplification information about the role each class plays in the design pattern, illustrates the complexity of understanding a framework architecture as a series of overlapping design patterns. Such a 'covering' of a design with pattern instances [OQC97] may be useful to developers and maintainers of the framework who are familiar with the code, but it does not seem to be a useful description for the purposes listed in the introduction.

'selling' the framework. Design patterns do not tell us what we can do with the framework: such a description doesn't show how we can tailor the framework to create a variety of applications.

reuse of architecture. To reuse an architecture we want to understand the domain-specific architecture that the framework offers and so we must have an understanding of how it models the problem domain, and what are the main components of the system and their relationships. We may choose to apply the same design patterns in our version of the framework, or we may choose to use other techniques or language features to maintain these relationships. A description using design patterns describes the software in too much detail: there may be many design patterns in a framework, but some will be more essential to the architecture of the framework than others.

In other words: for representing architecture, "not all patterns are equal". There are two reasons for this:(a) some *kinds* of GoF patterns are more *architectural* than others

in the sense that they describe a relationship between entities which could be seen as an infrastructure for a complete application (e.g. Model-View-Controller pattern for GUIs), rather than a relationship which is based solely on inheritance. GoF behavioral patterns are candidates for this category. [BMR⁺96] makes a distinction between architectural patterns and general purpose patterns. (b) the *granularity* at which a pattern is applied also determines if it is essential or non-essential to the architecture of the framework. That is, components of the architecture need not map one-to-one on the classes of the framework. So patterns which involve classes which are invisible in the architecture are also superfluous to the architectural description.

integration of frameworks. If we want to combine several frameworks, certain assumptions about the architecture of the framework must be understood. Many have to do with the context in which the framework is expected to be used: what does it require as lower-level infrastructure, and what kinds of applications can it serve the infrastructure for. Issues such as the handling of control, concurrency, persistence and distribution are also critical [GAO95]. Though design patterns can be used to document such aspects [MRB98], many of these issues are overlooked in framework documentation.

evolution and re-engineering. Frameworks evolve continuously, and it is important to be able to measure the extent of architectural drift of a framework. But is a change in design pattern use a sign that the architecture has changed? Perhaps a different design pattern or some other technique is now being used to maintain the same relationship. A notation which allows to express the architectural style of the framework allows us to track the evolution of the framework. It also allows us to test hypotheses about the way the framework is structured.

Communicating a framework design through patterns is certainly preferable to documenting each class and its responsibilities individually, and is an important view for developers. But it suffers from two main problems: its *incompleteness* - design patterns can not document all the design decisions emanating from the requirements, and its *fine-granularity* - design patterns overly describe the system, without revealing a more global view of the system.

3 Architecture: Design in Context

Software architecture has been defined as the decomposition of the software into the main design elements (the components) of the system and the interactions among these components (the connectors), and the rules or conventions governing their assembly [SG96]. We subscribe to the view that no *one* set of components and connectors can fully describe a software architecture: the architecture of an application can be described by a set of different views, and the mappings between these views. Each view will have a different definition of a component and a connector. Kruchten [Kru95], for example, proposes four views: logical view, process view, physical view and development view, which are elaborated through the use of scenarios, representing the important variations of the systems functionality. A variety of strategies and techniques are used to map views on each other. A similar approach is also taken in several object-oriented design methods [Boo91]. Aspect-oriented programming also recognizes the need to reconcile different aspects of a system, and proposes an approach for combining these at the programming language level. We are more concerned with gaining an overall understanding of the system and making explicit the assumptions and rules used in its construction.

What is often presented as the architecture of an application is the strategy used to solve the foremost design problem. But other side-issues and the policies used to resolve them also form part of the architecture and may document the hidden assumptions which come

3

up in the integration of frameworks. It is therefore unlikely that one notation or method can be used to express all aspects of an application's architecture and to satisfy the four points outlined in the introduction.

A variety of complementary forms of documentation help an engineer to understand the architecture of the system. Both the *what* - the context: what is the application domain, what are its main elements and their relationships, what problems must be tackled in building the framework architecture - and the *how* - the solution: how have the framework designers chosen to tackle these problems - must be addressed.

application domain model. This is a description of the main elements of the problem domain and how they relate to each other. It is the result of a domain analysis and is usually a first step in object-oriented design: identifying domain concepts and making the relationship between them explicit.

design space for the domain. Each problem in the domain can be solved using a variety of approaches. In a design space each problem represents an axis along which are possible or recommended solutions [DMNS97]. A design space may also include some collected wisdom about choosing solutions in the particular domain. Understanding the design space allows us to situate the particular framework as a point in this space.

examples. This is particularly useful to illustrate what can be done with the framework: the kinds of applications that can be created using it. Whereas a design space for the domain helps to situate the framework with respect to other approaches and so documents the *fixed* elements of the framework, examples illustrate the *variability* in the framework.

architectural patterns. The strength of design patterns for program understanding is that they tap into familiar metaphors - and so their names give us a handle on understanding them before we delve into the details of their structure and collaborations. In describing architectures, the power of metaphor is evident - blackboard, pipe and filter, for example, are architectural styles [SG96] for which we have an intuitive understanding. The kind of patterns that present us with a metaphor for a complete architectural view, or a large part of one, are ones which could be most useful in describing framework architectures. Such metaphors can succinctly describe components, the relationships between them, and at the same time provide us with familiar scenarios for their behavior.

scenarios. Scenarios are instances of use-cases. In object-oriented design these are used to elucidate requirements and derive class responsibilities. They are invaluable in understanding an architecture because they give a concrete illustration of the relationships of the components to each other. Design patterns also describe abstract scenarios, but these are small scenarios with only two or three objects as participants. More useful are scenarios which show how several components (i.e. granularity greater than classes) interact to assure one variation of the system's functionality.

These forms of description aim to give an understanding of the system at a coarse granularity and can complement more detailed forms of documentation such as design patterns and class hierarchies. Whereas architectural patterns and scenarios address more the *how* - the design solution - domain models and design spaces set up a context to describe the *what.* For 'selling' the framework, positioning the framework in a design space and showing examples of applications are probably most useful. For the reuse of the architecture, domain models, design space description and architectural patterns are appropriate as a starting point. The question of integration of frameworks is a difficult one, but a design space which is thorough should document also the environment in which it is expected

the framework will be used. Finally, all these forms of documentation are useful in forming concepts to be used in re-engineering and in tracking evolution, though there are open questions as to how these descriptions can be integrated in a methodology.

4   Discussion and Future Work

We have argued for the need for an architectural description of frameworks and have discussed why design patterns are insufficient for such a description. We have proposed that more traditional artifacts of domain analysis and object-oriented design are better vehicles for giving a global view of a framework's architecture. This holds, of course, not only for frameworks, but also for other software applications.

Our interest in finding good ways to describe software architecture stems from our work on re-engineering and program understanding[1]. Documentation generated through domain analysis and preliminary design helps engineers in forming concepts about the software architecture and facilitates the task of more detailed understanding of the code. Though design patterns can also help, just detecting the presence of certain design patterns, without having an understanding of architectural structures is not of much help.

When architectural documentation is missing, any aid in regenerating it is of enormous help to engineers confronted with legacy software. We are therefore investigating notations and tools which help engineers to understand the structure of a software system and to extract different views of its architecture. We believe that a good tool for program understanding should allow an engineer to formulate hypotheses about the structure of the software and to confirm or reject hypotheses based on evidence from the source code. One such tool is described in [MN97] - it allows an engineer to compute a 'reflexion model' of a software system showing the divergence of the hypothesis formulated from the actual structures found in the source code.

We are currently investigating how dynamic information from program execution can help engineers in understanding an application's architecture. To this end we are developing a tool, DynamEx [RDW], to help engineers in generating multiple architectural views of an application by querying information about static structures of the source code and about events from executions.

5   Acknowledgments

References

[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. Stad. *Pattern-Oriented Software Architecture – A System of Patterns*. Wiley, 1996.

[Boo91]   G. Booch. *Object Oriented Analysis and Design with Applications*. The Benjamin Cummings Publishing Co. Inc., 1991.

[DMNS97] S. Demeyer, T. D. Meijler, O. Nierstrasz, and P. Steyaert. Design guidelines for tailorable frameworks. *Communications of the ACM*, 40(10):60–64, Oct. 1997.

---

[1]see FAMOOS project http://www.iam.unibe.ch/~famoos/

[GAO95]   D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, Nov. 1995.

[GHJV95]   E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, Mass., 1995.

[Joh92]   R. E. Johnson. Documenting frameworks using patterns. In *Proceedings OOPSLA '92 ACM SIGPLAN Notices*, pp. 63–76, Oct. 1992.

[Kru95]   P. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, Nov. 1995.

[MB97]   M. Mattsson and J. Bosch. Framework composition: Problems, causes and solutions. In *Proceedings of TOOLS USA '97*, July 1997.

[MN97]   G. Murphy and D. Notkin. Reengineering with reflexion models: A case study. *IEEE Computer*, 17(2):29–36, Aug. 1997.

[MRB98]   R. Martin, D. Riehle, and F. Buschmann, editors. *Pattern Languages of Program Design 3*. Addison Wesley, 1998.

[OQC97]   G. Odenthal and K. Quibeldey-Cirkel. Using patterns for design and documentation. In *Proceedings of ECOOP'97*, LNCS 1241, pp. 511–529, 1997.

[RDW]   T. Richner, S. Ducasse, and R. Wuyts. Understanding object-oriented programs through declarative event analysis. ECOOP '98 Workshop on Object-Oriented Reengineering.

[SG96]   M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.