An Architecturally-Evident Coding Style: Making Your Design Visible in Your Code

George H. Fairbanks
Rhino Research
george.fairbanks@rhinoresearch.com

Abstract

Because of Eric Evans' Domain Driven Design, software developers are already familiar with embedding their domain models in their code. But the architecture and design is usually hard to see from the code. How can you improve that? This tutorial describes an architecturally-evident coding style that lets you drop hints to code readers so that they can correctly infer the design. You will learn why some design intent (the intensional part) is always lost between your design/architecture and your code. It builds upon ideas like Kent Beck's Intention Revealing Method Name pattern and provides a set of lightweight coding patterns and idioms that let you express your design intent in the code.

Categories and Subject Descriptors D.2.11 [*Software Engineering*]: Software Architectures

General Terms Design, Documentation

Keywords software architecture, software design, design intent, coding style, pattern

1. Presenter Bio

George Fairbanks is the president of Rhino Research, a software architecture training and consulting company. He holds a Ph.D. in Software Engineering from Carnegie Mellon University, where he was advised by David Garlan and Bill Scherlis. His dissertation introduced design fragments, a new way to specify and assure the correct use of frameworks through static analysis. He has publications on frameworks and software architecture in selective academic conferences, including OOPSLA and ICSE. He has written production code for telephone switches, plugins for the Eclipse IDE, Android phone applications, and everything for his own web dot-com startup. He maintains a network of Linux servers in his spare time. This tutorial is based on a chapter from his book *Just Enough Software Architecture: A Risk-Driven Approach* [4].

2. Target Audience

This is an intermediate-level tutorial aimed at software developers and architects. Ideal participants are active software developers who are fluent with software architecture concepts. However, background ideas (e.g., essentials of software architecture) will be covered sufficiently so that any software developer can participate. This tutorial is primarily slide-based lecture and some code examples will be shown.

3. Tutorial Objectives

Participants will learn some of the basics of software architecture just in case they do not already know them, things like the standard set of abstractions (modules, components, connectors, ports, styles/patterns) and relationships (designation, refinement, dependencies, partitions).

They will then learn what parts of the design or architecture can be straightforwardly translated into code and what parts cannot — the extensional parts (enumerated) go easily but the intensional parts (universally quantified) do not. So general rules such as "Never do validation in the UI" cannot easily be expressed in Java.

Given that, they learn the specific kinds of design intent that are valuable to express in the code and corresponding hard and soft mechanisms to make it visible.

Finally, participants learn about software frameworks as one particular difficulty, because frameworks usually impose their own idea of architecture on the code.

4. Summary of Contents

Kent Beck [1]and Eric Evans [3] have written about how you can express your design in the code. This session is based on a chapter from my book (which is going to the publisher in a few weeks). If others have written about expressing architecture in code then please let me know so I can add those references to the book.

A few years ago, Amnon Eden and Rick Kazman published a paper that prescribed a distinction between architecture and design [2]. They said that architecture included the intensional elements, the ones that were universally quantified (like "no client can circumvent the cache"), and that these cannot be expressed in mainstream programming languages. The code can respect intensional constraints but you cannot, for example, write that constraint as a Java expression (though you could in a rules-based language like Prolog). This is one large category of design intent that the developer knows when writing the code but that cannot be expressed in the programming language directly, so that design intent is lost.

The inspiration for this material comes from David Garlan who commented to me that source code differs in its ability to convey its architecture to readers. I have subsequently elaborated on his insight to develop a set of patterns that make the architecture more visible.

4.1 Inventory of hard and soft mechanisms to convey intent.

Hard mechanisms are machine checkable (e.g., the type system) while soft mechanisms rely on humans (e.g., method naming patterns).

4.2 Architectural design intent from the module viewtype

Source code is itself in the module viewtype so code expresses most elements from the module viewtype rather well. Most languages, however, lack a full-featured module system. Some express modules only via the directory structure where source files are saved, assuming that one directory means one module. They cannot express the dependencies between modules that are important parts of the architecture model. Programming languages commonly have relatively simple module visibility restrictions that can force you to break encapsulation.

Programming languages let you declare data structures and classes but not the larger architectural elements like component, connector, and port types. It is difficult to see what set of classes makes up a component or connector type. Classes and interfaces can express what services are provided, but not what services are required. While you can talk about the dependencies code has, it is usually awkward or impossible to express those dependencies in the code itself.

Protocols for interaction are an obvious concern and visible in architecture models but have no first-class representation in source code, though code comments often discuss legal calling sequences. Protocols can be expressed using annotations, which are increasingly common in object-oriented languages. Annotations are also being used to express other architectural properties.

4.3 Architectural design intent from the runtime viewtype

The entire runtime viewtype is hard to envision from looking at source code because you must read through the code and mentally animate the runtime instances. This mental animation is made harder with branching, looping, and input parameters. When relevant code is not co-located it is easy to overlook places where new components are instantiated or where connections are made.

A runtime view of the system can look like a sea of objects. Boundaries between components are hard to discern because the code does not let you declare anything larger than a class. Connectors are hard to see too because identical communication mechanisms, such as method calls or the observer pattern, are used both within and between components. Connectors may have no runtime representation at all. Communication between components does not happen just at ports, but often from any number of objects inside the components.

Runtime constraints and styles are exceptionally difficult to see from the source code. Constraints and styles usually refer to components and connectors rather than objects, so inferring them is doubly hard. First, components and connectors must be identified from the sea of objects and second the rules governing their runtime arrangement must be inferred.

4.4 Architectural design intent from the allocation viewtype

The runtime viewtype is merely difficult to infer from source code, but it is usually impossible to infer the allocation viewtype. Natural language is used to describe how code should be deployed, if it is written down at all. Most code is deployed in one large chunk on a single machine, but not always. The kind of machine and the network properties will impact the system's performance, and in cases it may be possible to express these properties in the code.

4.5 Patterns for expressing this design intent

Not everything can be expressed but we can express quite a bit. Method names can embed architectural intent, such as properties of connectors (compare "read()" with "readAsynchronous()"). The "reification" pattern is heavily used to make explicit what was implicit: The class hierarchy (or "tag" interfaces) can express component and connector types so that they stand out from other classes. Classes can also represent ports and can check protocol state and compliance.

5. Structure of Contents

The example that runs throughout is for a natural-language email processing system. Processing proceeds using a pipe and filter network of linguistic classifiers. Each filter in the network shares a common superclass and there is just one pipe class. Because all components are subclasses of the Component class, you can easily find them all by using your IDE's "browse hierarchy" view. Creation of these filters is localized in one method, as is the creation of the network, so you can envision the runtime structure from looking at the code.

Section 1: What is software architecture? This section provides background information on software architecture, including basic concepts (quality attributes, architecture drivers, architecture as a skeleton) and a conceptual model (views and viewtypes, domain-design-code).

Section 2: Design intent and how it is lost. It shows how design and implementations are the same and how they differ. Specifically, it describes how intensional [2] design elements (such as "No update may circumvent the cache") are hard to convey in the code, while extensional elements can be expressed.

Section 3: Techniques for expressing architecture in code. It teaches the Model-In-Code Principle: Expressing a model in the system's code helps comprehension and evolution. This principle applies equally to the domain (well-accepted in OO/agile circles) as well as to the design / architecture (accepted in architecture circles). However, design concepts like Layers and Components are not automatically visible, but you can follow a set of intention revealing patterns and idioms to do so. This tutorial describes patterns for package organization, subclassing, naming (classes, variables), first-class connectors, and initialization.

Section 4: Architecture and frameworks. More often than not, developers build code within an application framework (like Spring, Eclipse, OSGi, or Enterprise Java Beans). In some ways this helps express architecture, and in other ways it obfuscates it.

References

[1] K. Beck. *Smalltalk Best Practice Patterns.* Prentice Hall PTR, 1996.

[2] A. H. Eden and R. Kazman. Architecture, design, implementation. *International Conference on Software Engineering (ICSE)*, pages 149–159, 2003.

[3] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley Professional, 2003.

[4] G. Fairbanks. *Just Enough Software Architecture: A Risk-Driven Approach.* Marshall & Brainerd, 2010.