The Perfection of Informality: Tools, Templates, and Patterns

Dirk Riehle, www.riehle.org, dirk@riehle.org

Stanford University

Abstract

Tool support for using design patterns in software development has been a failure so far. There are at least two reasons: Either, the tool and its pattern notation have remained incomprehensible for all but the most well-trained developers, or, the tool and notation have been so limiting that they have been of little use at all. The underlying reason for these two problems is a mismatch between two competing needs. Design patterns were introduced to support informal communication in human design activities and have therefore been defined only informally. Tool support, however, requires precise specification to allow for automated application and conformance checking. In this article, by distinguishing design patterns from design templates, I show how to reconcile the need for informal and effective communication with the need for precise specification.

1 Introduction

Design patterns are abstractions from recurring problem solutions in design, given a specific context. We abstract from our experiences and document them as design patterns so that we can better use them in the future. The declared use of design patterns, for a long time, has been expert use, usually on a whiteboard, as a means for supporting informal design activities [1].

The groundbreaking and popular Design Patterns book is geared towards the human reader. Each pattern is first illustrated with a motivating example. The example is followed by an illustration of the most common form that the pattern takes (the Structure Diagram, Participants, and Collaborations sections). Only at the end of the pattern description are other less common forms of the pattern discussed.

As helpful as the Structure Diagram may have been to the initial reader, as harmful has it been for the tools and notation community. In their efforts to formally specify a design pattern, this community has focussed on the most common form of the pattern, as captured by the Structure Diagram, ignoring other less common forms. The consequences were endless and pointless debates whether a particular design is an instance of, for example, the Observer or the Bridge pattern.

The use of design patterns to support human design activities is its original and most important purpose. Most notably in whiteboard discussions, design patterns are used to discuss a design without getting bogged down by the details. This way, design patterns make us more expressive and effective in dealing with key design issues. However, when trying to get into the details, tools for applying design patterns are helpful because they make developers faster and they are important, because they help developers avoid mistakes.

To get the best of both worlds then, I show in this article how we can support design patterns with design templates. A design pattern remains a prose-based description of a common design idea that triggers the recall of the proper experience in an expert designer. A design template is a precise specification of one common form that a design pattern can take. For a given design pattern, there is an unlimited set of design templates that fulfill the pattern idea.

This approach reconciles the original design pattern idea of helping expert communication with the need of the tools community for precise specification.

Section 2 reviews the original pattern definition of the patterns community and introduces an example that illustrates the problem. Section 3 reviews the tools' community approach and the resulting problems. Section 4 reconciles design patterns with design templates and describes its effects on tool support. Section 5 summarizes the article.

2 The Patterns Community

Alexander et al. and Gamma et al. define a pattern to be "the abstraction from a solution to a recurring problem in a specific context" [1, 2]. Consider the Observer pattern, which has the intent to "define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."

What are typical design structures that exhibit such a property? Figure 1 shows the Structure Diagram from the Design Patterns book. It illustrates how one class is statically coupled with its dependent classes using a narrow observer interface that provides just the methods needed to receive update notifications.

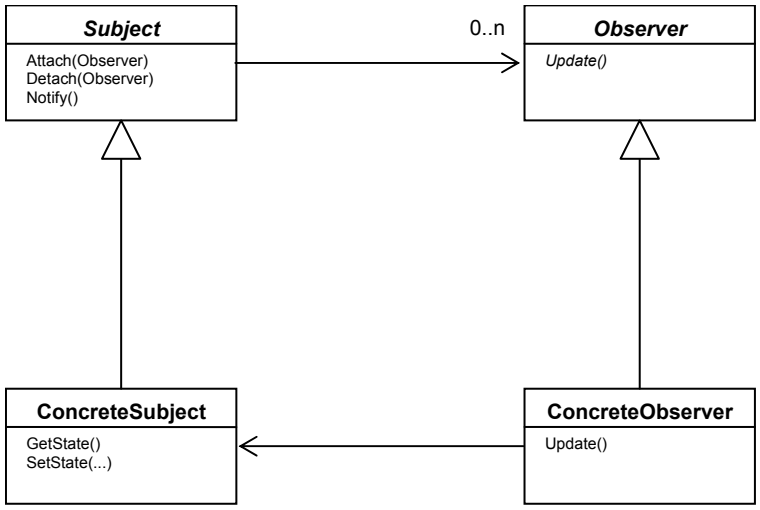

Figure 1: The Structure Diagram of the Observer Pattern from the Design Patterns book

But what about a generalized design structure as shown in Figure 2, where instances of a class describe their possible state transitions using state change objects and where dependent objects can register selectively for specific state transitions? One advantage of this design structure is that dependent objects are only informed about state changes if they previously registered for this type of change. Is this the Observer pattern as well?

What about so-called publish/subscribe systems, also known as information bus or event channel systems, where both the subject and the observer object are decoupled from each other using a transparent piece of middleware? Shaw and Garlan have called this an architectural style, but it fits the Observer pattern's intent description well [3].

What about a Linda-style tuple-space, where threads publish data objects into the tuple-space and where other threads fish them out [4]? Originally designed for parallel computation, Linda fits the intent of the Observer pattern well. What about all the variants of the discussed designs that use a pull-mode rather than a push-mode to receive event notifications?

All of these design structures are instances of the Observer pattern. The Structure Diagram in a design pattern description is the illustration of one common form only [5]. This implies there can be many different generalized forms of a pattern that satisfy the pattern idea.

Therefore: *A design pattern is an abstract idea and there can be several different variants of the same pattern, all representing the same idea in a different generalized form.*

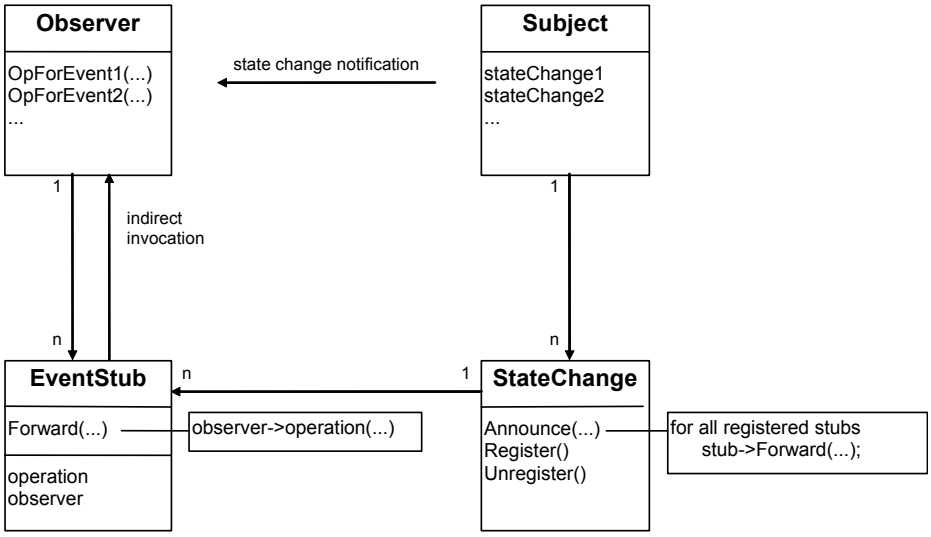

Figure 2: A variant of the Observer pattern with explicit state transitions

Following this definition, a design pattern is primarily a label for a common design experience that expert developers share because they've seen the same idea being applied over and over again.

The key phrase here is "common design experience". It provides us with the starting point for resolving the apparent conflict between the needs of informal human communication and the need for tool support. Common design experience relates to a developer's experience as grown in practice. As such it serves the purpose of communicating about designs with other expert developers.

Communicating about a design on a whiteboard does not require that the patterns and concepts employed have a formal basis. Effective human communication is inherently ambiguous and unprecise and frequently takes place on a level that contains lots of detail mistakes. In fact, I've often found such ambiguity to be an enabler for creative thought and would not want to miss it.

This is what design patterns were originally introduced for and where they are most useful: *The purpose of a design pattern is to make communication between expert developers more effective.*

3 The Tools and UML Community

When design patterns became popular, it was only natural to develop tool support.

The first generation of tools was limited. Usually, a tool showed a developer the design structure of the Structure Diagram and allowed for renaming and parameterization of the design before the design was applied or code was generated. Effectively, the design pattern was captured using a simple code generation template.

There are two things problematic with this approach:

- It does not capture the wide range of forms a design pattern can take.
- The pattern was expressed using an adhoc and usually inconsistent language.

I'll address the second problem first.

A language for expressing a design pattern needs to provide modeling concepts that are different from the traditional object-oriented modeling concepts of class and association, to name the most common ones. Already in the Structure Diagram, a participant "class" is not a class but rather a placeholder for a class. The placeholder becomes a class only when it gets instantiated.

This problem becomes more apparent when we go back to the original Structure Diagram of the Observer pattern. In Figure 1, we can see exactly one ConcreteObserver class. However, every experienced developer knows that this class is not only just a placeholder for one class, but for an unspecified number of classes. There could be and should be classes ConcreteObserver1, ConcreteObserver2, etc.

Traditional object-oriented modeling has no capability of expressing such a cardinality constraint. For example, in UML it is possible to express a 1..n relationship between objects. However, it is not possible to express such a relationship between classes [6].

In addition, UML does not provide support for modeling patterns. While the UML 1.x specifications point towards using collaboration specifications, they don't show how to do so. Moreover, collaboration specifications are too limiting and don't capture many structural design patterns that are more about class structure than about collaboration behavior.

In reaction to these shortcomings, a second generation of pattern modeling languages was introduced [7, 8]. These languages made a clear distinction between the concepts used for modeling a pattern and those used for modeling an instantiated pattern. While clean in their approach, these pattern modeling languages use first-order logic or related languages that remain incomprehensible to most software developers.

However, even the improved modeling languages cannot capture all the wealth of a design pattern in one form. The authors of the Design Patterns book allowed for different generalized forms, and so must tool support.

4 Design Patterns and Design Templates

Returning to the first problem with the initial generation of pattern tools, to successfully provide tool support, design patterns need to take the form of templates that developers can readily apply. And, since there is no single template that can satisfy the wealth of possible applications that a design pattern can have, we need multiple design templates.

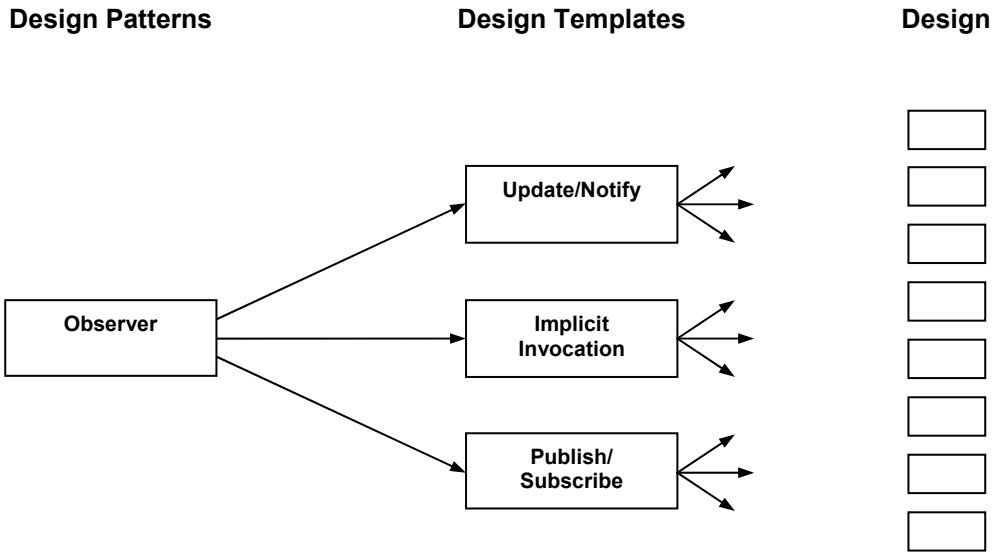

Figure 3: Levels of Abstraction in Design Pattern Use

This separation of the design pattern idea from a number of formally specified design templates resolves the example confusion discussed above. For one pattern, the Observer pattern, we can have three (and more) design templates, here called Update/Notify, Implicit Invocation, and Publish/Subscribe. Each of the templates can be

applied a thousand times in different system designs. Figure 3 illustrates the relationship between these different levels of abstraction.

Each level of abstraction serves a different purpose:

- The design pattern level provides expert developers with a name that hooks into their experience of how to solve a specific category of design problems.
- The design template level provides developers with formally specified design structures that can be applied efficiently in a given design using a tool.
- The design level is the resulting design in which it is clear where a specific design template was applied and to which design pattern this template is linked.

With appropriate tool support, this separation of concern allows for automated application and conformance checking of design patterns while leaving developers the freedom to work effectively on a whiteboard or any other informal communication medium.

This then is how design templates help design patterns and thereby become useful: *The purpose of a design template is to help automate the application of a pattern and to check for compliance in an implementation.*

5 Conclusions

In this article I propose to distinguish between design pattern and design templates in object-oriented design to make tool support for patterns possible and useful.

Design patterns and templates complement each other. A design pattern captures the idea of how to solve a specific category of problems in design, and a design template is a precisely specified (yet general) form that provides sufficient detail to allow for automated application and conformance checking through a modeling tool. For any given design pattern, there can be any number of design templates that fulfill the design pattern's idea.

Design patterns are the domain of human expert communication, for example, using a whiteboard. Design templates are the domain of modeling tools and are used to make the design process less error-prone. Separating the design pattern from supporting design templates allows everyone to do what they are best at: To creatively solve a design problem, and to efficiently determine the details of a design and implementation.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[2] Christopher Alexander et al. *A Pattern Language.* Oxford University Press, 1977.

[3] Mary Shaw and David Garlan. *Software Architecture. Prentice* Hall, 1996.

[4] Nicholas Carriero and David Gelernter. "Linda in Context." Communications of the ACM 32, 4. 444-458.

[5] John Vlissides. Personal Communication, 2003.

[6] Object Management Group. *UML 1.5 Specification.* OMG, 2003.

[7] Amnon Eden et al. "LePUS: A Declarative Pattern Specification Language." Tel Aviv University, 1998.

[8] Toufik Taibi et al: "Formal Specification of Design Patterns." Journal of Object Technology 2, 4: 127-140.

Biography

Dirk Riehle has worked in software research and development, both on the technical and the business side. He is interested in large-scale software systems development. Most of his technical work has focussed on object-oriented software architecture, frameworks, and patterns. He is also interested in metamodeling and was the first implementor of a UML virtual machine (OOPSLA ’01). Dirk holds a Ph.D. in computer science from ETH Zurich and is half-way through the MBA program at Stanford’s Graduate School of Business. For all technical materials, please see www.riehle.org. Dirk welcomes email at dirk@riehle.org.