Design Pattern Builder

A Concept for Refinable Reusable Design Pattern Libraries

Tobias Dürschmid

Hasso Plattner Institute, University of Potsdam, Germany

tobias.duerschmid@student.hpi.de

Abstract

Reuse is one of the core principles in professional software engineering. Design patterns provide a reusable solution for common design problems. But their implementations are generally not reusable as they are often well tailored to a specific context. We introduce a concept, that facilitates the reuse of their implementations by defining an abstract design pattern definition that can be instantiated with specialized design decisions. This approach is a meta-level Builder constructing design patterns as first-class citizens. It simplifies the application of design patterns by providing a pattern library and still being able to adjust it to the concrete context.

Categories and Subject Descriptors D.2.13 [*Reusable Software*]: Reuse models; D.2.11 [*Software Architectures*]: Patterns

Keywords Design patterns, modularity, reusability, AOP

1. Introduction

The implementation of a design pattern often includes substantial boiler-plate code that is scattered over multiple classes. Furthermore the developer can make mistakes while implementing a design pattern (e.g. forgotten synchronization in a multithreaded Singleton implementation). Reusable implementations of design patterns speedup development by avoiding faults, focusing on the design decisions instead of implementation details and better standardization of the vocabulary. But the reuse of their implementations can cause some challenges motivated by the following example: A developer wants to apply the Observer design pattern [1] to a virtual monopoly game. The `Player` object has the role of the Subject and the `Display` object the role of the Observer interested in the current balance of the corresponding `Player` object.

Crosscutting concerns. In order to archive reuseability, modularizing design patterns is an essential step. But the Observer code is scattered over the domain classes `Player` and `Display`. Hence the introduced superimposed roles Observer and Subject can crosscut the behavior of the involved classes [3]. This tends to lead to poor modularity and less reusability [3].

Context tailoring. Design patterns have to be tailored to their concrete context by making individual design decisions like using the push or pull model, defining the collection type for managing the observers for a subject or using publisher-subscriber-middleware.This makes each design pattern implementation unique and makes it unlikely that one configurable generic implementation of a design pattern can fit all variations of the context.

We argue that context tailoring can be archived by separating the design decisions from the structural design pattern definition and the option of refining these variation points. In order to modularize the design patterns, it is desirable to have languages constructs which treat design patterns as first-class citizens of the programming languagelike classes, methods or aspects in AOP.

2. Concept

If design patterns are first-class citizens on the same level of abstraction as classes, it is possible to apply design patterns to design pattern implementations. Our concept uses the Bridge to separate the design pattern definition from the design pattern instantiation and the Builder to simplify the construction of a concrete design pattern instance. Listing 1 shows one possible application of the concept to the Observer example.

Design pattern definition. The generic part of a design pattern consists of the definition of roles (e.g. Observer and Subject), their responsibilities (e.g. managing a list of Observers, calling `notify` on all Observers when the Subject has changed) and the collaboration between the roles. The

SPLASH Companion'16, October 30 – November 4, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4437-1/16/10...$15.00
http://dx.doi.org/10.1145/2984043.2998537

roles can be played by classes, objects, methods, attributes or values according to the type of concern. In our concept, the design pattern definition may delegate responsibilities to variation points similar to object-based inheritance (delegation). This variation points are design decisions to be made in order to configure the design pattern. Refining a design pattern implementation to a new context can be done by implementing one of these variation points or by overriding some parts of the base-design-pattern, similar to the Bridge design pattern [1] with the design pattern definition as Abstraction and the design decisions as Implementations.

Design pattern instantiation. In order to simplify the construction of one concrete design pattern instance, we apply the Builder design pattern [1] on a meta-level to design pattern implementations. The developer configures the Pattern Builder with a specification that chooses one implementation for each design decision, e.g. use a linked list as data structure for managing the registered observers and the push model without publisher-subscriber-middleware. In order to decouple the client from the concrete implementations, the specification can consist of functional and non-functional requirements the implementations should have.

Furthermore, the design pattern instatiation assigns the roles of the pattern definition (e.g. Observer and Subject) to the concrete entities in the domain code (e.g. `Display` objects and `Player` objects). The events, the observer should watch for (e.g. setting the balance) can be defined using a pointcut that is passed to the Pattern Builder.

Through design pattern instantiation, the participating domain classes are linked together. The domain code should have no dependencies to the instantiation code in order to allow easy changes of the architecture. The dynamic weaving of the pattern code together with the domain code can be done using AOP.The instantiation code could look like this:

```
PatternBuilder<DPObserver> builder = new PatternBuilder;

builder.setRole(Subject, player); //methods for configuring
      the Subject role are injected into player
player.setCollectionType(ArrayList); //class as parameter
PushModel model = new PushModel(builder); //instantiate the
      design decision
player.setModel(model);
model.setChanged(set(player.balance)); //pointcut as parameter

builder.setRole(Observer, display);
display.setNotify(display.showText(newValue)); //method as
      parameter
```

Listing 1. Design pattern instantiation of the Observer

3. Related Work

Since the publication of design patterns [1], many researchers tried to find a way to reuse their implementations.

One approach to modularize design patterns is AOP.

Hannemann and Kiczales [3] describe design pattern implementations in AspectJ.Some of their implementations (e.g. Observer, Composite, Iterator) could be reused, but they do not provide the ability of specialization or configuration of the design pattern implementations.

Automatic code generation for design patterns simplifies the implementation and allows refining, but produces more code that has to be maintained. This can lead to the round-trip problem:modified code will be overriden by regeneration. In contrast, our concept uses the weaving of aspect-oriented programmingin order to keep the pattern code out of the domain code.

The concept of *Object Teams* [4] facilitates refinable reusable implementations of multi-object collaborations. It modularizes crosscutting collaborations by introducing a ”team” as new refinable first-class language construct. We apply a similiar concept focusing on design pattern implementations. The application of Builder and Bridge on a meta-level to design patterns has not been done in this field.

4. Discussion

A quantitative study of *Hannemanns* implementation [3] measuring separation of concerns, coupling, cohesion, and size, shows improvements in separation of concerns but an increase in complexity [2]. This also applies to our solution because of the similarities to AOP solutions.

Our concept allows to fit the concrete context by refining. Unlike code generation, these refinements can be reused in similar contexts because of the separation of the implementation of the design decisions and their composition. Furthermore the refactoring between patterns is easier because most of the changes only happen in the design pattern instantiation that can the adapted quickly. Furthermore, software product line developmentcan benefit from this concept by easily changing quality attributes of the system by choosing appropriate design decisions.

The downside is that there is more coding effort for the design pattern definition because of the more complex architecture. But usually there are many use cases for one reusable implementation. Hence the additional effort charges off quite fast if the library is used in many projects.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[2] A. Garcia, C. Sant’Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa. Modularizing design patterns with aspects: a quantitative study. In *Proceedings of the Conference on Aspect-oriented software development, AOSD 2005*, pages 3–14. ACM, 2005.

[3] J. Hannemann and G. Kiczales. Design Pattern Implementation in Java and AspectJ. In *Proceedings of the ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA 2002*, pages 161–173. ACM, 2002.

[4] S. Herrmann. Object teams: Improving modularity for crosscutting collaborations. In *Proceedings of the Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World, Net. ObjectDays 2002*, pages 248–264. Springer, 2002.