



International Journal of Emerging Technologies in Computational and Applied Sciences(IJETCAS)

www.iasir.net

Aspect-Oriented Design Diversity

Daniela Gotseva*, Mario Pavlov**

Computer Systems Department

Technical University of Sofia

Sofia, PostCode 1000, boulevard KlimentOhriski 8

Bulgaria

Abstract: This article discusses the possibilities offered by the aspect-oriented design paradigm for different design styles. Some of the design styles are illustrated through aspect-oriented design patterns. The design patterns in this article are associated with common design problems and comprehensive example solutions. To understand the cost-effectiveness and address the fields of application of design patterns, each pattern is described objectively and indifferently.

Keywords: aspect-oriented design; aspect-oriented design patterns; aspect-oriented design diversity; patch patter; wormhole pattern; participant patter; clean design; static and dynamic crosscutting

I. Introduction

A design pattern is a generally accepted solution to a problem or a domain of problems within a particular field of expertise, usually in engineering branches such as architecture and software. Design patterns can also be viewed as solution templates. They do not provide a concrete solution to a problem, but define guidelines how to carry out a good, simple and solid solution. Design patterns are used extensively in software engineering because they can provide benefits, such as:

- re-usability
- clean design
- interoperability
- complexity reduction

Software design patterns use fundamental features of programming paradigms to compose solutions. For example, a good number of design patterns exist (and most likely even more will be discovered) for the object-oriented programming paradigm because it has many powerful features. There are also a number of design patterns for the aspect-oriented programming paradigm. Some are very interesting and powerful due to the feature diversity of the aspect-oriented programming paradigm. As an extension to the object-oriented design paradigm, the aspect-oriented design pattern can redefine many of the object-oriented design patterns to make them simpler and cheaper to implement. The aspect-oriented design pattern can also provide completely different design patterns for the same problems. As already mentioned, the features that the aspect-oriented paradigm provides are very diverse. This allows for different design styles and is the main reason why aspect-oriented programming is so interesting and has so much to offer. This article aims to illustrate aspect-oriented design diversity through design patterns. All patterns listed here do not have object-oriented equivalents and use special features of the aspect-oriented programming paradigm, such as static and dynamic crosscutting.

II. Patch

The field of software engineering has a great number of library software packages. As a result, a software system is normally composed of a number of “third party” libraries plus the code base of the system, which is usually much smaller, compared to the library code base it depends on. When a system heavily re-uses third party code or libraries, the following common problems may arise:

- discovering bugs in the library code
- introducing changes to the library code that break the client code
- creating limitations in the library code that can cause problems in the client code

Of course, the list of potential problems caused by library code we offer is far from exhaustive, but the issues listed here are often the most difficult to deal with. These problems are hard to address because library code by nature cannot be changed directly. So when changes to library code are required, the only way to really add the changes is to somehow push these changes upstream. This is not easy when a library is freely available, but is even harder when it is not. As a result most of the time library code problems are handled in the client code. This practice is problematic because the code that addresses the library problems (the patch code) inevitably ends up in the wrong place. Let's look at a couple of examples:

- Example 1: There is a bug or a performance issue in the log4j's Logger.log() method.

A possible solution would be to subclass `Logger` and try to work around the problem. As a result, the client code must ensure that each call to the `log()` method will be forwarded to the right class, for example `CustomLogger`. While this solution should work in a lot of similar cases, it adds to the system's complexity, comes with a certain extent of code duplication, and is altogether ugly.

- Example 2: A conditional log method is missing in `log4j` and will be added in a future version.

A possible solution would be to add a utility code to provide this function. As a result, at least one class has to be referenced from all instances where this function is required. This adds a kind of temporary code that is potentially referenced in many places. Furthermore, this solution decreases code readability because method calls to utility code are harder to understand than the same method calls on the object concerned, for example `LoggingTools.assertLog()` versus `Logger.assertLog()`. Such client utility code is confusing and in the wrong place, and even worse - it has to be removed (along with all references) after the conditional log method is added to the library code.

Fortunately aspect-oriented systems such as `AspectJ` can simplify dealing with library problems to a great extent. Using static crosscutting features allows for easy manipulation of library code. Through a feature known as intertype declaration (ITD), the static structure of a system can be altered, that is members can be added or redefined. Code listing 1 shows a better version of the patch code from Example 2 using an aspect.

Code listing 1

```
public aspect LoggerPatch {
    public void Logger.assertLog(boolean assertion, Level level, String msg) {
        if(assertion) {
            log(level, msg);
        }
    }
}
```

The `LoggerPatch` uses intertype declaration to weave the `assertLog()` method into the `Logger` class. In this way clients of `log4j` can directly refer to the `assertLog()` method as shown in Code listing 2.

Code listing 2

```
public class Account {
    private static final Logger logger = Logger.getLogger(Account.class);
    // code omitted for brevity
    public void withdraw(int amount) {
        // code omitted for brevity
        balance -= amount;
        logger.assertLog(balance < LOW_BALANCE, Level.WARN, "Account balance below " + LOW_BALANCE);
    }
}
```

This technique enables the client code to call the `assertLog()` method directly on the logger object. This way from the client's point of view the `assertLog()` method is in the right place. When this method is officially added in the third party library, the only change that would be required to use the official `assertLog()` is to just remove the `LoggerPatch` aspect. No other changes to the client code would be necessary. This approach is known as the patch pattern [1]. This pattern is both interesting and controversial. While it is useful for the following reasons:

- library code can be modified or extended indirectly in order to fix known bugs or add required features
- client code remains unaffected by patch aspects
- the pattern eliminates some legal restrictions on library modification and redistribution because the actual changes are not physically done to the library but they reside in the client code

The patch pattern also has the following downsides:

- the patch code is rather fragile and can be broken by changes to the library code
- the pattern somewhat decreases the cohesion and increases the coupling of the client code

Certainly the patch pattern can help in many situations, but its usage should be kept under control because of its possible negative impact.

III. Wormhole

One of the properties of clean code is the lack of method parameters. Code is considered clean and readable when methods have none or at most one parameter. Good practice when writing methods is to aim for zero or one parameter. Stated like this, it sounds easy and straightforward, but when it comes to actually laying out the code, things look different. It is not rare for method bodies to require context information, and a lot of it at that. When the context is available within the same object, context information is easy to obtain. However, when the context is only available to the caller of the method, there is no other way but to pass the necessary information to the method (callee) as a parameter or a collection of parameters. So the only good way for a method to get required

context information that is not available in the method's host class is through method parameters. Of course, there are other possible solutions, such as global variables and ThreadLocal, but they might do more harm than good and they are better left aside. This does not mean that adding method parameters whenever a piece of context information is required is good, on the contrary, adding method parameters are almost as bad. At least two reasons exist why it is undesirable to have a lot of method parameters:

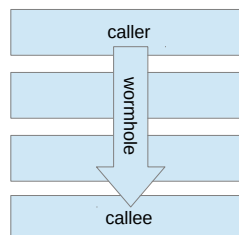
- Method parameters make reading and understanding code increasingly more difficult. Method signatures become too long and verbose. Calling methods with a high number of parameters is also error prone.
- When you need to modify method parameters, the entire call chain should reflect the changes. Figure 1 shows the impact of such modification. When call chains that carry context information are very big (as is often the case) changing context parameters could become a nightmare.

Figure 1 The impact of method modification.

```
loadImage(String src,
findUserStorage(String src,
obtainStorageConnection(String src,
authorize(String user)
```

We can apply a few techniques to decrease the impact of the high number of method parameters, such as “Preserve Whole Object” [2] and “Introduce Parameter Object” [2]. These are refactorings that are designed to encapsulate what varies, namely to encapsulate changes to method parameters. Such refactorings are very helpful, but they introduce dependencies, for example the parameter object. As useful as these refactorings may be, they do not completely eliminate the problem with method parameters. Aspect-oriented programming features can be used to engineer a general and complete solution to this problem. Aspect-oriented systems such as AspectJ can have pointcuts [3] that select join points [3] within the control flow of other pointcuts and also collect the runtime context information of all join points. This means that we can create a pointcut that effectively perforates the call stack, making context information available from caller to callee, see Figure 2.

Figure 2 How to make context information available from caller to callee.



This is referred to as the wormhole pattern [4-6]. Using the wormhole pattern with aspect-oriented programming makes passing any context information as method parameters redundant. Aspects can just perforate the calls stack, collect the necessary context information, and perform the necessary crosscutting operations, such as authorization and policy-based routing. A full working example of the wormhole pattern is shown in Code listing 3.

Code listing 3

```
public class Client {

    private Gateway gateway = null;
    private boolean important = false;
    public Client(Gateway gateway, boolean important) {
        this.gateway = gateway;
        this.important = important;
    }

    public void readNews() {
        String contents = gateway.request("news.com");
        System.out.println(contents);
    }

    public void readEmail() {
        String contents = gateway.request("webmail.com");
    }
}
```

```

        System.out.println(contents);
    }

    public boolean isImportant() {
        return important;
    }
}

public class Gateway {
    public String request(String address) {
        return "direct contents of " + address;
    }
}

public class Proxy {
    public String retrieve(String address) {
        return "proxy contents of " + address;
    }
}

public aspect PolicyBasedRouting {
    private Proxy proxy = new Proxy();
    pointcut callerExecution(Client client) : execution(* Client.*(..)) &&this(client);
    pointcut calleeExecution(String address) : execution(* Gateway.request(java.lang.String))
    &&args(address);
    public pointcut wormhole(Client client, String address) : cflow(callerExecution(client))
    &&calleeExecution(address);
    String around(Client client, String address) : wormhole(client, address) {
        if(client.isImportant()) {
            return proceed(client, address);
        }
        return proxy.retrieve(address);
    }
}

public class Launcher {
    public static void main(String[] args) {
        Gateway gateway = new Gateway();
        Client regularClient = new Client(gateway, false);
        regularClient.readNews();
        regularClient.readEmail();
        Client importantClient = new Client(gateway, true);
        importantClient.readNews();
        importantClient.readEmail();
    }
}

```

This is a hypothetical policy-based routing example. In the example, a client sends requests for web resources to a gateway and the gateway should return the respective resource. A business policy specifies that important clients get direct access to the resources and regular clients are transparently forwarded to a proxy. In this example, no context information (about the importance of the client) is passed to the gateway/proxy side, yet regular clients are forwarded to the proxy and important clients get direct access. The policy rule is enforced by the PolicyBasedRouting aspect. The around() advice triggers when the gateway's request() method (the callee) is called from within the control flow of any of the client's methods (the caller). The wormhole() pointcut is responsible for selecting these join points. The wormhole works regardless of the number of layers between the callee and the caller. As long as the specified callee execution is triggered by execution of the specified caller, the wormhole will function. In this example there are no additional layers between the callee and the caller, but in a real-world scenario we are likely to have some. With the help of a fairly simple aspect, you do not have to pass the context information required by the crosscutting concern explicitly and that greatly simplifies the code.

IV. Participant

When using aspect-oriented programming, selecting the right join points is essential. It is the only way to specify where the crosscutting functionality is applied. Aspect-oriented systems provide very powerful mechanisms to select join points. Almost every static and dynamic point of a system's state and execution can be selected, except for some operations that take place on the stack. Although this enables almost every part of the system to be advised, in some cases it is not entirely clear exactly which join points should be selected. For

example, a case in question is when join points have a specific business characteristic that should be selected, but they do not have a common technical characteristic, for example a common super type. Of course, we can pre-define a specific list of such join points and use it in pointcuts, but the list will have to be maintained explicitly, that is whenever such join points appear or disappear, they have to be respectively added or removed from the list. Obviously, this is undesirable and should be avoided. The good news is that there is a way around this problem. Let us imagine that we have a text editor application that supports many operations on text. Some operations modify the text and some do not. We want a visual indicator to go on when the text is modified. Suppose that all operations are so diverse that they do not share types, packages or anything else that can allow reliable selection of all join points that modify the text. The naïve approach would be just to pre-define a list with these operations and use it in the pointcuts. As stated already, this is undesirable. Instead we could let the operations themselves contribute their join points to an abstract aspect that switches the indication on and off. An example of this approach is show in Code listing 4.

Code listing 4

```
public abstract aspect TextChangeIndicator {

    public abstract pointcuttextChanged();

    after() : textChanged() {
        // switch the indicator on
    }
}

public class EncodingOperation {

    public void encodeText(String encoding) {
        // code omitted for brevity
    }

    private static aspect EncodingChangeIndicator extends TextChangeIndicator {
        publicpointcuttextChanged() : execution(void EncodingOperation.encodeText(java.lang.String));
    }
}
```

The TextChangeIndicator abstract aspect has an abstract pointcut that is used by an after() advice to trigger the indication of changed text. The EncodingOperation class has a nested aspect that extends the TextChangeIndicator aspect and contributes the join point of its execution. So the aspect that does the indicator switching does not select any join points, but let's its subaspects do that. In a sense, every operation defines its join points if the operation should participate in the indicator functionality. This is known as the participant pattern [4-6]. This pattern is both simple and powerful. It can be used to greatly simplify join point selection. The usefulness of this pattern comes at a certain cost. Due to the nature of the pattern, the collaboration of aspects and classes gets reversed. With the participant pattern, classes know about aspects instead of the other way round. The consequence is that it is not easy to remove the abstract aspect because classes depend on it directly.

V. Conclusion

The aspect-oriented design paradigm is a celebration of design diversity. It offers the freedom to choose the right design style for the right problem. Unlike other programming paradigms, the aspect-oriented paradigm does not promote a single "right" design style. The examples in this article try to illustrate the viability of different design approaches through design patterns. Our analysis has shown that we should not discard different design styles simply because they are different. As seen in the examples, the proposed design patterns work at least as good as traditional design styles if not better. The research in the area of aspect-oriented design should be further continued and expanded.

References

- [1] J. Noble, A. Schmidmeier, D. J. Pearce, and A. P. Black, Patterns of Aspect-Oriented Design, January 18, 2008
- [2] M. Fowler, K. Beck, J. Brant, W. Opdyke, don Roberts, Refactoring: Improving the Design of Existing Code, Addison Wesley, 2002
- [3] D. Gotseva and M. Pavlov, Aspect orientation: The path to highly modular software design, J. of Communication and Computer Engineering, 2012.
- [4] I. Kiselev, Aspect-Oriented Programming with AspectJ, Sams, 2002.
- [5] R. Laddad, Aspect J in Action, Practical Aspect-Oriented Programming, Manning Publications, 2003.
- [6] R. Laddad, Aspect J in Action: Enterprise AOP with Spring Applications, Manning Publications, 2009.