



How Aspect-Oriented Programming Can Enforce Clean Design

Daniela Gotseva*, Mario Pavlov**

Computer Systems Department

Technical University of Sofia

Sofia, PostCode 1000, boulevard KlimentOhridski 8
Bulgaria

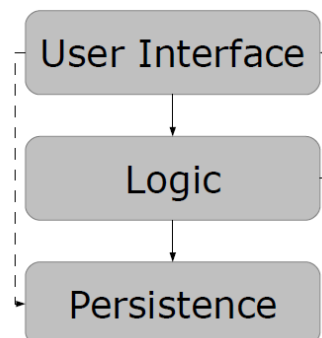
Abstract: This article discusses the idea of software design policies. It explains the problem of software design decay and presents common ways to address this problem via software design policy enforcement. The main part of this article is dedicated to design policy enforcement via aspects. A general scenario is illustrated using working examples. Also a complete framework is created for reference and full demonstration of the solution presented in this article.

Keywords: Aspect-oriented Programming; Java Programming; Code Duplication; Software Design Policy; Code Scattering

I. Software Design Policy

A software design policy is a rule within a particular software design. Such rules must be followed closely to keep designs clean and concise during the life cycle of a software system. Design policies work on different scales: from large scales like system layers to small scales like function's invocations. For example, design policies can be used to define boundaries between system layers or define forbidden functions. Figure 1 shows a large scale policy that defines the layer dependencies of a software system.

Figure1 Large Scale Policy.



The continuous arrows show the dependencies that are allowed and the dashed arrows show the forbidden dependencies. For example, the User Interface (UI) layer can only depend on the Logic layer, but the UI layer cannot depend on the Persistence layer. In a lower scale policy example, there is a rule to use a specific API to approach certain tasks, for instance to use non-blocking I/O operations as opposed to blocking operations.

Policies are normally set when designs are developed. Their main purpose is to protect the designs from decay. However, design policies may be set after the design has been implemented. Usually code refactoring's result in new design policies. In the real world software design continues to change and evolve after it is implemented. New requirements often lead to design changes. Whenever the design changes, the design policies should be also revised. If design policies change, it is best to refactor the system to comply with the new policies. Otherwise the system becomes expensive to maintain. When some parts of a system comply with a certain set of rules and other parts do not, changes to the system lead to code that is confusing and ugly. In other words, it is easier and cheaper to refactor the system to comply with the new policies rather than adhere to old and new rules for an indefinite amount of time and then eventually refactor the system [7-9].

One of the main issues with design policies is that they can be difficult to follow. All people who work on a system should be well aware of all design policies and how they have to be followed. Also, if for some reason a policy is broken, this should be somehow made obvious. Deviating from design policies for any reason leads to design decay.

II. Software Design Decay

Software design decay is the process of gradually crossing the boundaries of the original design rules. When work on a software system is in progress, we say that the system is evolving. This evolution (similar to evolution in the real world) may result in realizations that are suboptimal according to the original design or directly deviate

from the design rules. Unfortunately, in software engineering there is no such thing as natural selection and the only cure for inferior realizations is refactoring. Software design decay also works on different scales. Although design decay occurs differently on different levels, it invariably occurs for the same reason, because of changes during the system evolution that for any reason do not follow the original design. Such changes are often grounded in common excuses such as:

- Dead-lines pressure: when deadlines are pressing, people tend to circumvent rules and just make the required change to set the task to finished as soon as possible. The result is often bad and ugly code.
- Quality has a lower priority: sometimes engineers are not allowed to spend enough (or even any) effort on code quality, which can make the system unstable and very difficult to maintain.
- People are unaware of the rules: this becomes really problematic when engineering teams grow in size. When new people are added to the team, they cannot learn everything about the system before they start working on it, so they tend to make mistakes caused by unawareness.
- People do not care about the rules: some people simply do not think that design rules are worth following and they just do not follow them. It could be also that they do not follow certain rules, because they do not like them.
- The system has low test coverage and that makes refactoring too expensive or impossible: this is probably the worst possible state for a system. If the system does not have enough tests, it becomes a matter of not breaking the system rather than following policy rules. Once a system is low on tests, it is increasingly more difficult to maintain that system.

When a design realization starts decaying, the level of difficulty and the amount of work required to make changes to the system increases. The process of decay may even lead to violating general design principles and cause:

- Code duplication
- Code tangling [1]
- Code scattering [1].

All of these processes have a strong negative impact on a system. Unfortunately, design decay inevitably leads to system evolution slowdown or a complete shutdown. Even more unfortunate is that when this happens, the common response is to put more people to work on that system which (more often than not) leads to bad results and even project failure.

The right way to avoid design decay is to strictly follow the design and never break its rules. For the reasons stated above this is not an easy task. It is a lot of effort to keep a good level of awareness about the design within the team working on the system, motivate the people to follow the rules willingly and prevent any violations. Of course, fending off design decay successfully is very rewarding. Clean design allows systems to grow and evolve, which is crucial for success. Although it is expensive to maintain clean design, it is cheaper in the long run because proactive system maintenance leads to higher quality as opposed to reactive system maintenance. In other words, it is cheaper to actively keep a system clean rather than cleaning up a system after problems have already occurred.

III. Software Design Policy Enforcement

A possible approach to minimize system maintenance cost would be to enforce design rules in order to eliminate the possibility of errors caused by unawareness or negligence. Generally, there are two ways to enforce design rules or policies: manual and automatic enforcement. The most common way of policy enforcement is through regular code reviews. Automatic enforcement is achieved with the help of various static code analysis tools. Table 1 shows the pros and cons of both approaches.

Table 1 Shows the pros and cons of Software Design Policy.

Policy Enforcement Type	Pros	Cons
Manual	Resolves non-trivial violations Aligns code to a single coding style Raises overall code familiarity within the team	Generally very slow More expensive for bigger changes Violations can be overlooked
Automatic	Immediate indication when violations are introduced Never misses statically determinable issues Generally very fast Independent of the change size Policy sets can be very large	Cannot detect non-trivial violations May require complex configuration

There is no definite answer as to which approach is better. However, it is safe to state that both approaches can be used in conjunction. For the purpose of this article, we are going to focus on the automatic approach.

There are a number of tools that provide static code analysis, such as FindBugs [2] and PMD [3]. While these tools are great, they are in a way limited to the predefined set of rules they support and extending them with domain specific policies is either difficult or not possible. However, aspect-oriented programming can be used to define policies that can be enforced at compile time. Since aspect-oriented systems support a wide variety of join points, it is quite easy to create aspects that define join points that serve to identify design violations. Moreover, because aspect-oriented systems such as AspectJ are very powerful, they allow for very powerful design policies. To illustrate how to achieve this in practice, I have created a small framework for design policy enforcement [4].

The framework is composed of a set of abstract aspects. Each aspect defines a design policy category and can be extended to provide concrete and/or domain specific policy enforcement for this category. The categories can include but are not limited to:

- API access
 - i. Reflection access
 - ii. System class access
 - iii. Direct field access
 - iv. Library API access
- Exception and Error handling
 - i. RuntimeException handling
- Object instantiation
- Layer/module access
- Custom/domain-specific enforcements

Each aspect from the framework defines generic join points that identify the entire policy category and abstract join points that can be used from concrete aspects to augment, filter or add to the policy, so that the resulting concrete aspect serves as concrete design policy enforcement. For example, the JDK access policy enforcement abstract aspect from the framework looks like Code listing 1.

Code listing 1

```
public abstract aspect JdkAccessEnforcement {
    pointcut jdk16() :
    call(* java.applet.*.*(..)) || call(java.applet.*.new(..)) || get(* java.applet.*) ||
    call(* java.awt.*.*(..)) || call(java.awt.*.new(..)) || get(* java.awt.*) ||
    // code omitted for brevity
    public abstract pointcut allowed();
    public abstract pointcut enabled();
    declare error : enabled() && (!allowed() && jdk16()) : "Illegal JDK access!";
}
```

The JdkAccessEnforcement aspect has one concrete pointcut that defines all calls to code from Java Development Kit 1.6 (JDK6) and two abstract pointcuts for clients to implement. It also has a declare error construct that triggers if code selected from the enable() pointcut has calls to code selected from the jdk6() pointcut and at the same time is not selected from the allowed() pointcut. So a concrete aspect looks like Code listing 2.

Code listing 2

```
public aspect JdkAccessEnforcementImpl extends JdkAccessEnforcement {

    publicpointcut allowed() : call(* java.lang.*.*(..)) || call(java.lang.*.new(..));
    publicpointcut enabled() : within(org.project.database.*);
}
```

JdkAccessEnforcementImpl restricts the access to all JDK code except for code in the java.lang package and this rule applies to all code within the org.project.database package. In other words, this aspect ensures that the database module can access only the java.lang part of JDK. This approach of creating policies is very flexible because policies can be created to apply only to specific parts of the system and even have different definitions for these parts.

Another interesting example is creating policies for enforcing factory patterns. The framework aspect is shown in Code listing 3.

Code listing 3

```
public abstract aspect FactoryEnforcement {
    public abstract pointcut instantiation();
    public abstract pointcut factory();
    declare error : instantiation() && !factory() : "Direct instantiation not allowed! Use factory!";
}
```

This aspect does not have concrete pointcuts, instead it only defines the abstract pointcuts and the relation between them. An implementation of this aspect is shown in Code listing 4.

Code listing 4

```
public aspect FactoryEnforcementImpl extends FactoryEnforcement {
    publicpointcut instantiation() : call(org.project.garage.Car.new(..));
    publicpointcut factory() : within(org.project.garage.factory.CarFactory);
}
```

Here `FactoryEnforcementImpl` defines which class instantiations are to be restricted and the location where they are permitted, namely the factory. This basically restricts all calls to the constructor of the `Car` class, except when they are made from within the `CarFactory` class.

We can also create policies for error handling. The framework contains an aspect that can restrict the handling of Errors. Generally, it is a good idea not to handle Errors, which are normally thrown by the JVM (Java Virtual Machine). Code listing 5 shows the abstract aspect.

Code listing 5

```
public abstract aspect ErrorHandlingEnforcement {
    pointcuterrorHandling() : handler(java.lang.Error+);
    public abstract pointcut enabled();
    declare error : enabled() &&errorHandling() : "Errors must not be handled!";
}
```

Similar to the other aspects, this aspect is very simple. It has one concrete pointcut that selects code that has the catch construct with a parameter `Error` or any of the `Error` sub-classes. The aspect has one abstract pointcut for clients to define which code this rule applies to. An implementation looks like Code listing 6.

Code listing 6

```
public aspect ErrorHandlingEnforcementImpl extends ErrorHandlingEnforcement {
    publicpointcut enabled() : within(org.project.database.*);
}
```

The concrete aspect only needs to define where this rule is to be enforced. In this case it is all code within the database module. Usually when an Error such as “out of memory” and “stack overflow” occurs, there is not much that could be done so it should not be caught.

To illustrate a larger scale policy, the framework has an aspect that can restrict interactions between different layers/modules in a system. It is shown in Code listing 7.

Code listing 7

```
public abstract aspect LayerAccessEnforcement {
    public abstract pointcutsourceLayers();
    public abstract pointcuttargetLayers();
    declare error : sourceLayers() &&targetLayers() : "Illegal layer access!";
}
```

Because the `LayerAccessEnforcement` aspect represents a larger scale policy, its definition is very high level to provide the necessary flexibility. It has only two abstract pointcuts whose purpose is to select join points that represent unidirectional high level dependencies. An example implementation is shown in Code listing 8.

Code listing 8

```
public aspect UiToBusinessLayerAccessEnforcement extends LayerAccessEnforcement {
    publicpointcutsourceLayers() : within(org.project.ui.*);
    publicpointcuttargetLayers() : getFieldAccess() || setFieldAccess() || instantiations() ||
methodCalls();
    pointcutgetFieldAccess() : get(* org.project.*) && !get(* org.project.business.*);
    pointcutsetFieldAccess() : set(* org.project.*) && !set(* org.project.business.*);
    pointcut instantiations() : call(org.project.*.new(..)) && !call(org.project.business.*.new(..));
    pointcutmethodCalls() : call(* org.project.*.*()) && !call(* org.project.business.*.*());
}
```

In this example the source layer is defined as all code within the UI module and the target layer is defined as all calls to code in the entire system that is not in the business module. So this policy will trigger if code from the UI module tries to access code that is not in the business module, for example the database module. This is a powerful technique to keep the design boundaries strict.

These policies will be most useful and effective if they are enforced as early as possible. This means that they should be enforced at build time. This is also a simple task. The AspectJ compiler can be used as a drop-in replacement for the Java compiler, allowing for seamless integration with any build system. However, AspectJ has “out of the box” integration [5] with Apache Ant [6] which (when used) is usually more elegant. There are several ways to add policy enforcement to an existing build system based on Apache Ant. One way is to compile the classes and aspects separately and then use binary weaving [5] to enforce the policies as in Code listing 9.

Code listing 9

```
<target name="compileAndPackageClasses">
```

```

        <javacsrcdir="${project.classes.src.dir}" destdir="${project.build.dir}" />
        <jar destfile="${project.output.dir}/${project.jar}" basedir="${project.build.dir}" />
    </target>

    <target name="compileAndPackagePolicies" depends=" compileAndPackageClasses" >
        <iajcoutJar="${project.output.dir}/${policies.jar}" sourceRoots="${project.policies.src.dir}">
            <classpath>
                <pathelement path="${aspectj.lib.dir}/aspectjrt.jar" />
                <pathelement path="${project.output.dir}/${project.jar}" />
            </classpath>
            <aspectpath>
                <pathelement path="${framework.jar}" />
            </aspectpath>
        </iajcoutJar>
    </target>

    <target name="enforcePolicies" depends=" compileAndPackageEnforcements" >
        <iajcoutJar="${project.output.dir}/${woven.policies.jar}">
            <inpath>
                <pathelement path="${project.output.dir}/${policies.jar}" />
                <pathelement path="${project.output.dir}/${project.jar}" />
            </inpath>
            <classpath>
                <pathelement path="${aspectj.lib.dir}/aspectjrt.jar" />
            </classpath>
            <aspectpath>
                <pathelement path="${framework.jar}" />
            </aspectpath>
        </iajcoutJar>
    </target>

```

If enforcePolicies is called, it will first build the classes then build the policies (aspects) and finally via binary weaving enforce the policies. Another way of achieving the same would be to do everything in one go as in Code listing 10.

Code listing 10

```

<target name="enforcePolicies">
    <iajcdstDir="${project.output.dir}">
        <sourceroots>
            <pathelement path="${project.classes.src.dir}" />
            <pathelement path="${project.policies.src.dir}" />
        </sourceroots>
        <classpath>
            <pathelement path="${aspectj.lib.dir}/aspectjrt.jar" />
        </classpath>
        <aspectpath>
            <pathelement path="${framework.jar}" />
        </aspectpath>
    </iajcdstDir>
</target>

```

In this case enforcePolicies does everything at once that is it compiles classes and aspects, and weaves them. This is formally known as source weaving [5].

Whether source or binary weaving is used depends entirely on the build and development processes. Binary weaving is more flexible, whereas source weaving is simpler, but the final result is the same. If there are policy violations, the output of the AspectJ compiler looks like Code listing 11.

Code listing 11

```

/project/src/ErrorHandler.java:32 [error] Errors must not be handled!
} catch (NoClassDefFoundError e) {
~~~~~

// output omitted for brevity
/project/src/Garage.java:30 [error] Direct instantiation not allowed! Use factory!
new Car();
~~~~~

```

The output is very descriptive and just a brief look at it shows exactly where and what the problem is. It contains the file system path to the class with the violation, the line number where the violation occurs, a description of the violation and highlights the violation itself.

To sum up, design policy enforcement using aspect-oriented programming is relatively cheap to setup. The initial set of policies could be loose to ensure the smooth adding of policy enforcement. Then gradually the policies can grow in number and become tighter and stricter.

IV. Conclusion

The technical part of a project is essential for its success. Aspect-oriented programming can not only make a software system highly modular but also keep it clean, tidy, and in good order. It is only logical to use aspect-oriented programming for design policy enforcement when the former is already used in the system. We can also logically use aspect-oriented programming for design policy enforcement as a gateway to introduce aspect-oriented programming in the system. Either way, the result is a better system. Keeping a system clean may require some additional effort, but in the long run it can be very rewarding. Neglecting the proactive maintenance of a clean system could lead to its downfall. In other words, your code could be your personal heaven or hell.

References

- [1] D. Gotseva and M. Pavlov, Aspect orientation: The path to highly modular software design, J. of Communication and Computer Engineering, 2012.
- [2] <http://findbugs.sourceforge.net/>
- [3] <http://pmd.sourceforge.net/>
- [4] <http://sourceforge.net/projects/dpe/>
- [5] D. Gotseva and M. Pavlov, Aspect-oriented programming with AspectJ, International Journal of Computer Science Issues, Vol. 9, Issue 5, No 1, September 2012, pp. 212-218.
- [6] <http://ant.apache.org/>
- [7] I. Kiselev, Aspect-Oriented Programming with AspectJ, Sams, 2002.
- [8] R. Laddad, Aspect J in Action, Practical Aspect-Oriented Programming, Manning Publications, 2003.
- [9] R. Laddad, Aspect J in Action: Enterprise AOP with Spring Applications, Manning Publications, 2009.