Pitfalls of AspectJ Implementations of Some of the Gang-of-Four Design Patterns

Miguel Pessoa Monteiro[1] and
João Miguel Fernandes[2]

[1] Escola Superior de Tecnologia de Castelo Branco,
Instituto Politécnico de Castelo Branco,
Av. do Empresário 6000-767 Castelo Branco, Portugal
mmonteiro@di.uminho.pt

[2] Departamento de Informática, Universidade do Minho
Campus de Gualtar, 4710-057 Braga, Portugal
jmf@di.uminho.pt

1. Introduction

For the previous edition of this workshop [8], the authors wrote a paper presenting the aims of the first author's Ph.D. project [13], which includes the development of a catalogue of refactorings [3] for the AspectJ programming language [1]. Case studies are be used for refactoring experiments, to gain the necessary insights. In this paper, we present several considerations on some code examples in AspectJ [10] from our second case study. They look like a critique, which may lead readers to assume we have a low opinion of them. That is not the case: we largely consider them a success, and proved to be a rich source of insights. Our intent is to simply point out some problems in using the code examples, thus contributing to their analysis. This is done in section 2.

We take the opportunity provided by this paper to present, in section 3, an update on a style rule proposed in the previous paper [13]. In section 4 we conclude the paper.

2. Analysis of the AspectJ Implementation of Design Patterns

AspectJ is a backwards-compatible aspect-oriented (AO) extension of Java and it seems natural to refactor existing Java code bases into AspectJ. Java legacy systems can thus benefit from the advantages brought by aspect-oriented programming (AOP). In the paper written for the previous edition of the workshop [13], we stated our intention of using a Java framework for workflow applications as the subject of our first refactoring experiment. Though that experiment yielded some results [12], they were not as rich and varied as we initially hoped. In addition, use of this framework as a case study was compromised by the decision, taken by its developers, to adopt a tool based on generation of source code. It is nonsensical to apply refactorings to generated code, and therefore our motivation to use that framework as a case study disappeared.

We used the dual implementations (version 1.1) [10] of the Gang-of-Four (GoF) patterns [9] in both Java and AspectJ as our second case study. These code examples became a de facto benchmark of good AO design and good code style. Study of these examples yielded many insights and provided a significant contribution to our work. Our approach was to explore the kind of refactorings that would enable the transformation of the Java implementations into AspectJ. Next, we tested and refined the resulting refactorings using other Java implementations of the same patterns, by different authors (e.g. [6] and [5]). We currently have several articles in preparation presenting the results of this work (e.g. [14]).

We detected problems and limitations in some of the AspectJ implementations during our study. Some of them were awkward to use, with limitations that may not be noticed until one attempts to use them in concrete cases. We are aware the implementations are, to a certain extent, speculative, but since these problems are not mentioned in [10], we believe our findings can usefully complement it. For instance, 12 of the aspects are called "reusable", without mentions to constraints or to some price that must be paid for such reusability – readers of that paper can be led to believe no such problems exist. Many of the problems relate to the complexity and inflexibility of interfaces. There is indeed a price to pay for making an aspect reusable: in some cases, we were left wondering whether the benefits were worth paying (e.g. in *Composite*). The intent of this paper is to call these problems to attention. Another issue is applicability. In some cases, the AspectJ implementation is not applicable to all possible instances of the pattern. We suspect that the applicability of some of the implementations is restricted (e.g. *Command*, *Memento* and *Template Method*). From our analysis, we conclude that attaining reusability is hard, even with AOP. Some of the causes for the difficulties stemmed from aspect-specific issues, including: (1) difficulties in obtaining the adequate joinpoints to weave the desired extra behaviour in the intended point of the program; (2) difficulties in obtaining joinpoints exposing the context required by the aspect at the appropriate moments; (3) difficulties for aspects to quantify over objects without violating encapsulation. We also detected difficulties in base code using proprietary components, which made it illegal to weave additional state and behaviour. Hannemann et al [10] also mention the lack of genericity, which leads to the widespread use of type casts, including type-unsafe downcasts – another obstacle to reusability.

Space constraints prevent us from providing a proper introduction to the patterns covered – interested readers can refer to [9]. We limit ourselves to an (necessarily brief) analysis of the corresponding AspectJ implementation from [10].

All implementations (Java and AspectJ) presented in [10] include a Main class enclosing a static main method providing an illustrating use case, which acts as the driver to its corresponding example. We refer to some of these Main classes in our discussion.

2.1. Command

Command ([9], p.233) prescribes that some the objects represent **commands**, usually modelled by an abstract class (C++) or an interface (Java). Concrete commands are instances of concrete classes extending a Command type. The pattern also defines the **invoker** role for objects that place requests to the command objects. The AspectJ implementation is based on the reusable aspect CommandProtocol, which encapsulates the logic associated with the command role. CommandProtocol allows only one command to be associated to each invoker at a time, though it allows for alternative implementations resorting to composites to hold multiple commands (leaving open whether their traversals have a defined order or not). The authors remark (in a code comment within CommandProtocol) that a single command is usually sufficient, though we easily found an example [6] using *several* invokers. The advantage of the AspectJ implementation is to separate the pattern role from their primary logic. This assumes many commands include some other case-specific logic, in addition to the pattern-related logic. However, in many uses of Command the classes exist for the sole purpose of implementing command objects (usually through anonymous inner classes, which are by definition non reusable) and thus cannot benefit from this implementation.

CommandProtocol provides several variants for associating commands to invokers. The association can be *implicit*, through pointcuts, or *explicit*, through calls to the aspect method setCommand. The Main class uses the explicit mode (not necessarily a bad thing). We wondered how the implicit mode could be used and noticed that it presents a technical hurdle related to context capture. CommandProtocol models this functionality through the protected pointcut setCommandTrigger, which receives both the invoker and the command objects as parameters. We think that in practice, very few code bases can expose the context enabling the capture of both the invoker and the command in a single pointcut – for instance, this would be hard to do in the illustrating use case. In consequence, we suspect the implicit mode will not be feasible in many cases. This hurdle can be circumvented by refactoring the base code, but this is likely to be very invasive, and the resulting changes risk defeating the whole purpose of the refactoring – to make the base code oblivious [7] of the association between invokers and commands.

2.2. Composite

The AspectJ *Composite* (see also [9], p.163) achieves complete obliviousness from the pattern roles, but the authors aimed to make the aspect reusable as well. The CompositeProtocol aspect implements the composite functionality and defines a framework comprising (1) three marker interfaces representing a component, a composite component and a leaf component. The latter two extend the former. The remaining managing logic is done in terms of these interfaces; (2) a hash table, private to the aspect, responsible for mapping components to their children; (3) a protocol based on visitors, through which operations on the elements of the composite structure can be performed. The visitors are objects implementing one of two alternative (inner) interfaces, each defining an operation receiving a component as argument and differing in the return type – void for operations and Object for functions.

The requirement that all operations on elements go through the visitor interfaces – with all values passed as a single object – places a constraint on the client programmer. For instance, it is hard to implement operations in which an operation on one of the composite's elements use the results from the operation performed on the parent node or on previous children. The complexity is a result of the efforts to yield a reusable aspect. We managed to write an alternative aspect without visitors that preserves obliviousness and is simpler, though case-specific. We were left wondering whether the aspect's reusability sufficiently compensates for the awkwardness in using it. Similar problems plague in various degrees some of the other reusable aspects.

2.3. Decorator

Decorator ([9], p.175) is a way to emulate mixins [3] in languages not supporting the concept. The AspectJ implementation is based on advice – Hannemann et al [10] remark on the inherent limitations of this approach, namely the dynamic reordering of decorators. Advices are less flexible due to their static (i.e. compile time) nature. We believe this can be a serious drawback, due to various reasons. Besides the issue of different orderings in the composition of decorators – which in some cases can result in different behaviour – there are also the issues of enabling different combinations of decorators, and how to decorate just a subset of the instances of a class, or doing so only during specific phases. Plain advice cannot deal with these situations. Fortunately, it is possible to implement dynamic and flexible decorators using other techniques, such as context-aware pointcut designators (e.g. cflow, cflowbelow, within and withincode). This can lead to overly complex pointcuts, but we can complement the pointcuts with further techniques, such as registering the decorated objects and making the advice check the target object (see an example in Listing 1).

```
public class Component {
   public void sendMessage() {
      System.out.println("\tMESSAGE");
   }
}
public class Forwarder1 {
   public void forward(Component component) {
      System.out.println("Executing from Forwarder1: ");
      component.sendMessage();
   }
}
public class Forwarder2 {
   public void forward(Component component) {
      System.out.println("Executing from Forwarder2: ");
      component.sendMessage();
   }
}
public aspect Decorator {
   private Component _component;
   public void register(Component component) {
      _component = component;
   }
   public void unRegister() {
      _component = null;
   }
   pointcut client1calls(Component component):
      call(public void sendMessage(..))
      && cflow(execution(* Forwarder1.*(..)))
      && target(component);

   void around(Component component): client1calls(component) {
      if(component == _component) {
         System.out.println("BEFORE-----");
         proceed(component);
         System.out.println("------AFTER");
      }
      else proceed(component);
   }
}
public class Client {
   public static void main(String[] args) {
      Component component1 = new Component();
      Component component2 = new Component();
      Forwarder1 for1 = new Forwarder1();
      Forwarder2 for2 = new Forwarder2();

      Decorator.aspectOf().register(component1);
      for1.forward(component1);
      for2.forward(component1);
      for1.forward(component2);
      for2.forward(component2);

      for1.forward(component1);
      Decorator.aspectOf().unRegister();
      for1.forward(component1);
   }
}
```

Listing 1. Example of alternative decorator techniques.

2.4. Mediator

The *Mediator* ([9], p.273) is typically an object acting as the hub of communication for various other objects, named colleagues. The AspectJ implementation seems to regard *Mediator* as comprising a mediator role that can be attached to and detached from existing objects. The technique used, as in many other patterns, is based on a marker interface that a specific target class is made to implement through a declare parents clause. This approach brings the same benefits of obliviousness from pattern roles as with several other patterns. We nevertheless think this approach is not adequate for *Mediator*, because in our view the mediator role is defining, i.e. it only exists to perform this role. The AspectJ implementation marks one participant object with the marker interface, but this is misleading: the aspect holds the state needed to manage the various relationships and includes all the associated logic. In practice, it is the aspect, not one of the participant objects, which performs the role of mediator.

2.5. Memento

Memento ([9], p.283) defines the **originator** as the object whose state must be stored in a snapshot, the **memento** as the object storing a snapshot of the originator's internal state, and the **caretaker** as the object responsible for the memento's safekeeping. The memento needs privileged access to the originator's internals, something that is tricky to achieve in many languages (Cooper remarks in [5] that this is not directly possible in Smalltalk. The GoF book [9] suggests in that C++ implementations use the 'friend' construct). A common solution in Java [5] is to give the default (i.e. package protected) access to the originator's state, so that only classes within the same package have access to it, and place the memento class in the same package as the originator's.

The AspectJ implementation is based on an abstract aspect declaring a protected Originator marker interface, an abstract method (createMementoFor) receiving the originator as parameter and responsible for creating the memento, and another abstract method (setMemento) receiving both the originator and the memento and responsible for setting the memento with the originator's state. The caretaker is the Main class. Each specific case requires a concrete subaspect implementing the two methods and including a declare parents associating the class to the Originator interface. The memento role is represented by a standalone Memento interface declaring a setState method to set the memento's state and a getState method to return the memento's saved state.

The concrete aspect included in the example creates the memento through an anonymous class implementing the Memento type and providing a case-specific implementation of its setState and getState methods. The Memento interface is generally applicable and thus cannot refer to case-specific types – setState accepts an argument of type Object and getState returns a value of type Object. Clients of Memento must resort to downcasts. This interface strikes us as too constraining, because originators need to encapsulate their internals within a single object to be passed to setState. This entails creating an *additional* type just for the originator to pass its state to and from mementos. The use case in Main dodges this problem because the originator's state is a simple protected **int** field, wrapped within an Integer object and upcasted to Object when passed to the memento. Opposite downcasts occur when the memento returns its snapshot.

The AspectJ design is awkward due to the attempt make it reusable. We believe that in this case the authors tried to do too much. A case-specific aspect would not be reusable, but it would be considerably more flexible and easy to use. One possible solution to the problem of providing the memento with privileged access to the originator's state would be to use a *privileged* aspect, but this is generally regarded as risky and bad style.

The Java design presented by Cooper ([5], p.169) implements the memento as a peer class placed within the originator's source file. The fields of the originator have package-protected access. The memento gets the snapshot by receiving the originator as an argument to his constructor. The memento holds a reference to its associated originator and is able to restore the originator's saved state through a restore method. The originator class itself does not contain any code associated with the pattern.

The Java implementation presented by Hannemann et al places the responsibility of generating the memento to the originator, in addition to its primary responsibilities. In our view, this Java design does not compare favourably with Cooper's, which achieves the same main advantage as the AspectJ design – the originator is oblivious of its role in the pattern (at the price of having the memento class in its source file). We think Cooper's design provides a more suitable Java example to be compared with the AspectJ design – comparisons should use good designs in *both* languages.

2.6. The "Multiple Inheritance" Patterns

Hanneman and Kiczales classify 5 patterns (*Abstract Factory, Bridge*, *Builder*, *Factory Method* and *Template Method*) in a group having in common (1) structural similarities, (2) use of inheritance to distinguish different but related implementations, (3) inability of AspectJ to provide more reusable implementations. Related code can still benefit from case-specific aspects, enabling abstract classes to be replaced with interfaces without loosing the ability to attach default state and behaviour.

This places an interesting question: should abstract classes be considered bad style in the context of AspectJ?

The aspects resort to inter-type declarations of concrete members targeting interfaces to achieve this. Classes implementing these interfaces inherit the introduced state and behaviour, in addition to the one acquired through single inheritance. In summary, this capability comprises mixin inheritance [3] (Filman et al [7] remark that mixin inheritance is the earliest form of oblivious quantification and that mixins with multiple inheritance comprise a full AOP technology). AspectJ can emulate mixins through various ways. For instance, we can place an inner static aspect within an interface, defining concrete state and behaviour to its enclosing interface, which is inherited by any implementing class.

The use of mixin-aspects instead of abstract classes has the disadvantage that interfaces cannot have constructors, even when augmented by aspects. Initialisation code cannot be placed in constructors and initialisation values must be placed in setter methods. This has impact on client code, which must remember to call the setters after (and in addition to) instantiating the implementing object.

Traditional implementations of *Template Method* ([9], p.325) comprise a method – not usually accessible in subclasses – calling various (occasionally abstract) methods that are left to be defined by client programmers, in subclasses. This pattern is one of the key concepts in the design of frameworks, in which subclasses are called by the framework and not the other way round. The AspectJ *Template Method* resorts to the above mixin technique, separating definitions from declarations by placing concrete members in the aspect and replacing the abstract class with an interface and a mixin-aspect.

Implementing *Template Method* this way strikes us a bit contrived, and seems to correspond to a narrow view of this pattern. *Template Method* was developed with inheritance and abstract classes in mind, and does not seem to lend itself so well to this kind of separation. Often the variation points, or "hooks", defined by the template method directly depend on the implementation used in the abstract class, but the AspectJ implementation assumes that several alternative implementations can be used interchangeably. The AspectJ implementation does not consider that the class with the template method may be concrete (e.g. java.lang.Thread). We therefore suspect this implementation is of limited applicability.

3. Aspects Can Be Everywhere

In [13] we proposed the idea of segregating aspects from classes and interfaces, by taking advantage of the Java rules governing packages and the *package-protected* (a.k.a. default) access. It is possible to associate different directories to the same logical Java package, for instance by registering various directories as entry points in the CLASSPATH environment variable. Java source files placed in equivalent points of the hierarchy belong to the same package. This becomes especially relevant in cases involving members with package-protected access. In [13] we proposed that aspects be segregated from the remaining code by placing them in separate directories associated with the same logical package. When proposing this guideline, we were drawing a parallel between aspects and elements such as unit tests and generated files.

Though this guideline may be useful in some specific cases, it is based on the hypothesis that the segregation of aspects from classes brings benefits in the general case – one we feel is still shared by many people. From the experience gained since we wrote the previous paper, we now think this is not the right approach. There are several reasons for this. One is that some aspects – particularly abstract ones – tend to be small frameworks with auxiliary classes (e.g. exceptions) and interfaces [10]. It would not make sense to segregate these. In addition, the scope of applicability of aspects can vary widely. Aspects can affect an entire system comprising multiple packages, in which case it makes sense to place them in their own packages. The scope of aspects can also be made to restrict to a single package, in which case we should place them within that package. This leads to the problem of setting aspects apart from other kinds of elements, for instance when a programmer looks at dozens of source files placed in the same package. We wanted to distinguish aspects from classes (and from interfaces) but this was not possible, as all the files used the same extension – * .java. The more recent versions of AJDT [2] use the *.aj extension as the default, which seems to suggest a solution. Unfortunately, we think this will not prove to be so, because aspects can also span a single class. Such aspects should be placed within the class's source file, either as inner aspects or peer aspects. Interfaces can likewise enclose inner aspects (cf. cap.8 of [11]), and classes can enclose AspectJ-specific constructs (e.g. the *Participant* pattern [11]). Such code is not legal Java code and their source files should bear the *.aj extension as well. We conclude that aspects and aspect constructs can be placed in many places and it is pointless to segregate them – developers must resort to the views provided by IDEs and on their personal knowledge of the systems in order to distinguish one kind of source file from the others.

4. Conclusion

We present some considerations on the AspectJ implementations [10] of the GoF patterns [9]. Though aspects bring improvements in most cases, even with AOP it is hard to achieve reusability. In some cases, a reusable aspect comes at the price of awkward and inflexible interfaces that risk defeating one of the purposes of reusability – to lighten the burden of client programmers. In other cases, the applicability of a reusable aspect is restricted to an arguably small set of cases. In the case of the *Memento*, we believe the AspectJ design does not greatly improve on some of the best Java designs possible (e.g. [5], p.169). More flexible AspectJ designs for *Decorator* can be found than that presented in [10].

We backtrack on a style rule we proposed in the previous edition of this workshop [13]: we no longer propose that aspects be segregated from the base code. Aspects and aspect constructs should sometimes be placed with interface and class source files. For this reason we think it is pointless to try to segregate them.

Acknowledgements

Miguel Pessoa Monteiro is partially supported by *PRODEP III (Medida 5 – Acção 5.3 – Eixo 3 – Formação Avançada de Docentes do Ensino Superior)* and by *PPC-VM (POSI/CHS/47158/2002)*.

References

[1] AspectJ home page http://www.eclipse.org/aspectj/
[2] AJDT home page. http://www.eclipse.org/ajdt
[3] Refactoring Home Page, http://www.refactoring.com/
[4] G. Bracha and W. Cook Mixin-Based Inheritance, ECOOP/OOPSLA 1990.
[5] J. Cooper. Java Design Patterns: A Tutorial. Addison-Wesley 2000. Availabe at http://www.patterndepot.com/put/8/DesignJava.PDF.
[6] B. Eckel. Thinking in Patterns, revision 0.9. Book on progress, May 20, 2003. Available at http://64.78.49.204/ IPatterns-0.9.zip.
[7] R. E. Filman and D. P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness, workshop on Advanced Separation of Concerns, OOPSLA 2000, Minneapolis, October 2000.
[8] L. Fuentes, J. Hernández and A. Moreira (eds.), proceedings of the Desarrollo de Software Orientado a Aspectos workshop at JISBD, Alicante, November 2003.
[9] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.
[10] J. Hannemann and G. Kiczales, Design Pattern Implementation in Java and AspectJ, OOPSLA 2002, November 2002.
[11] R. Laddad, AspectJ in Action – Practical Aspect-Oriented Programming, Manning 2003.
[12] M. P. Monteiro, J. M. Fernandes, Object-to-Aspect Refactorings for Feature Extraction (industry paper), AOSD'2004, UK, Lancaster, March 2004. Available at http://aosd.net/ 2004/archive/Monteiro.pdf.
[13] M. P. Monteiro, J. M. Fernandes, Some Thoughts On Refactoring Objects to Aspects, in [8].
[14] M. Monteiro, J. Fernandes, Towards a Catalog of Aspect-Oriented Refactorings, technical paper, to be published.