

Design Patterns with Aspects: A Case Study

Marc Bartsch

*School of Systems Engineering, University of Reading, Reading, RG6 6AY, UK
m.bartsch@rdg.ac.uk*

Rachel Harrison

*Stratton Edge Consulting, Stratton Edge, School Hill, Cirencester, GL7 2LS, UK
rachel.harrison@strattonedge.com*

Abstract. *This paper reports on a case study in which four programs were implemented in both aspect-oriented and object-oriented versions using four different design patterns. We report on our experiences with the use of these design patterns and the trade-offs that needed to be considered. Furthermore, we present additional evidence for a reduction of coupling through the use of aspect-oriented versions of popular design patterns identifying those that may have a beneficial influence on software maintenance.*

1. INTRODUCTION

Research has been carried out recently which shows that design patterns [6] can lead to a reduction in coupling and to a better separation of concerns if used in an aspect-oriented manner [7]. Among the design patterns investigated were popular patterns such as *composite*, *visitor*, *decorator* and *observer*. The case study reported here complements this research by investigating these four design patterns in four programs from different domains. Each program, and thus each pattern, was implemented in two versions, an object-oriented and an aspect-oriented version.

Our aim was to gain general experience with the use of aspect-oriented implementations of the selected design patterns and the trade-offs that are involved in their use. *Decorator*, for example, has been reported to be problematic in the sense that the aspect-oriented implementation leads to a loss of dynamic properties such as dynamic reordering [8]. We wanted to gain more insight into the limitations of this design pattern and whether or not they could be remedied.

We were also interested in the extent to which coupling can be reduced by the use of an aspect-oriented implementation of design patterns in different contexts in order to present more evidence of the potential benefits that design patterns can offer. Among the four programs were two that use the *composite/visitor* pattern with different levels of abstraction. One program uses an abstract, role-based approach [8] whereas the other uses a much simpler implementation of the same patterns. We were interested in whether the level of abstraction had any general effect on the coupling we measured.

The four systems under consideration were all used in experimental research concerning design patterns and were originally written in C++ [9, 12]. All four programs were converted to Java and AspectJ and coupling was measured for each component (class, interface or aspect) of each program. Even though the four programs can be still be regarded as small scale systems ranging from 275 to 590 non-commented lines of code (NCLOC, [5]) they nonetheless resemble realistic small scale subsystems of the kind to be found as libraries or as components of larger systems.

Our work is directed at researchers and practitioners that are interested in additional evidence about limitations and the potentially beneficial use of design patterns in an aspect-oriented context. The remainder of this paper is structured as follows: Section 2 discusses related work; section 3 gives an overview of the methodology. Section 4 introduces each program in detail and section 5 presents the coupling measurement. Section 6 discusses the results and concludes with lessons learnt.

2. RELATED WORK

The two studies that are most related to our work are Hannemann et al. [8] and Garcia et al. [7]. Hannemann et al. report on the aspect-oriented implementations of popular design patterns using a role-based approach. In their work, design patterns were implemented in two layers. One abstract layer defined a pattern protocol, i.e. an aspect which localizes all code that belongs to an abstract view of a pattern. A certain instance of a pattern is then derived from this aspect. This approach leads to a reusability of the pattern protocol, since it offers a generic implementation of a pattern. The protocol can also define interfaces for each role that exists within a pattern. For example, the *composite* pattern protocol defines the roles *composite* and *leaf*. The assignment of classes to these roles will be done in the aspect that defines a specific pattern instance. In our case study, we used this role-based approach for the *composite/visitor* and the *observer* pattern.

Garcia et al. [7] report on a scalability study of aspect-oriented implementation of popular design patterns. In their work, the pattern implementations by Hannemann et al. were investigated with respect to different software properties. It was found that the patterns *composite/visitor* and *observer* can lead to a decrease in coupling whereas *decorator* can lead to an increase in coupling. It is our aim to corroborate their findings with results from additional coupling measurement to highlight a potential beneficial use of design patterns. In contrast to their study, we used a simpler coupling metric and the *composite/visitor* pattern in two different instantiations: one abstract, role-based instantiation and one straightforward instantiation.

3. METHODOLOGY

The approach that we applied in this case study consisted of several steps. First, we obtained the four C++ programs from their authors [9] and converted each C++ program to Java and AspectJ using either a role-based approach or a simpler solution. In order to guarantee equivalent implementations of the Java and AspectJ code, we were particularly concerned to achieve a similar level of abstraction. If a role-based approach has been used in the AspectJ version, a similar abstract implementation has also been used for the Java version. Consequently, if a simple pattern was used in the AspectJ version an equivalent simple version had to be found for the Java version, too. We used the Hannemann et al. pattern implementations as starting points.

Another area of concern was modularity. The aspect-oriented implementation usually achieved a high degree of localised code, i.e. all pattern related code was localised in a limited number of components and not scattered across each application. Even though the same degree of localised code could not be achieved in the Java version, we aimed to implement classes that exhibit similar concerns to those found in the aspect-oriented version. Inner classes and interfaces were found to be a useful technique in the object-oriented implementation to keep pattern related code in one place. For example, if the aspect-oriented pattern protocol defines interfaces for each role in an aspect, this was imitated by inner interfaces in a class that represented the pattern protocol.

Both the levels of abstraction and modularity have been adjusted frequently in each version of the programs until satisfactory implementations were found. This is of course a subjective assessment. Each program will be presented in detail in section 4 to show what levels of abstraction and modularity were chosen. Only if both implementations have been carefully designed with a similar level of abstraction and similar modularity can a fair comparison of the degree of coupling of the two systems be made.

Each program was then reduced to those components that were related to a design pattern to achieve a fair comparison of the measured coupling. The coupling metric was then calculated for each component of the reduced system (see section 5).

4. DESIGN CONSIDERATIONS

In this section we will introduce the four programs that we used to measure coupling. Before the results of the coupling measurement are presented, design considerations for each program version will be given containing the specific design of each object-oriented and aspect-oriented program.

4.1 Graphics Library: Composite/Visitor

The Graphics Library [9] has been designed for creating, manipulating, and drawing simple types of graphical objects (circles and lines) on different types of graphical devices (alpha-numerical output, pixel graphic device). Graphical objects can be rotated or shifted and several graphical objects can be grouped together and manipulated as if they represented a single graphical object. In contrast to the original source code which uses the *composite* and *abstract factory* pattern, we use the *composite* and *visitor* pattern in our adaptation. Unlike *composite* and *visitor*, *abstract factory* was shown to exhibit slightly more coupling in the aspect-oriented version than in the object-oriented solution [7], so the pattern remained in our adaptation, but was not converted to AspectJ. The tree of all graphical objects is represented by a *composite* pattern and manipulating graphical objects are expressed through a *visitor* for each kind of manipulation (shifting, rotating and also drawing). With the help of a *visitor*, a stricter separation of concerns can be achieved: member methods of a graphical object no longer implement any manipulation functionality, but each manipulation is localized in a *visitor*.

4.1.1 Design

The overall aim in the design of the Graphics Library was to achieve a highly abstract, role-based implementation of the design patterns involved. A role-based approach to design patterns as suggested in [8] defines abstract roles that exist in a pattern. For example, the *composite* pattern contains the roles *leaf* and *composite* and the *visitor* pattern contains the roles *leaf* and *node*. Classes that participate in these design patterns can assume the roles usually by implementing an interface that expresses the role. The interface can either be implemented explicitly by a class in the object-oriented version or it can be introduced to a class in an aspect.

The motivation behind the aspect-oriented, role-based approach is to separate code that implements business logic from code that implements patterns [8]. Existing implementations of the visitor pattern require the implementation of an *accept method* in every class that participates in a visitor leading to a scattered implementation of the visitor concern. An aspect-oriented implementation can help to achieve a stronger localisation of pattern related code. Information about the objects that participate in a visitor pattern is kept within aspects and the participating objects remain unaware. The major benefit of such an approach is what Hannemann and Kiczales termed call the ‘pluggability’ of a component [8]: participants can be plugged into the visitor pattern or removed from it without any code changes in these objects. This may lead to less maintenance effort since pattern changes can be carried out in a restricted locality.

An abstract pattern protocol defines the basic logic of a pattern and can be reused in different instances of the same pattern [8]. It modularizes all those features of a pattern that are common to all instances. Thus using a pattern protocol supports reusability of pattern code.

In the Graphics Library a role-based approach is only implemented for the *composite* pattern. The role *composite* was assumed by the *Objectgroup* class representing a group of graphical objects. All other graphical objects assumed the *leaf* role. For this pattern the distinction between a *composite* and a *leaf* was sufficient since the pattern implementation does not contain any code that needs to distinguish between different *leaf* classes. However, for the *visitor* pattern each *visitor* defines specific behaviour for each node that it visits: drawing a line is different from drawing a circle. For this pattern, the classification into *node* and *leaf* seems insufficient. There are at least three solutions to this problem. First, the distinction between *node* and *leaf* could be kept and each *visitor* determines the dynamic type of each object at runtime. Specific functionality is then executed according to each type. However, such an approach may lead to maintenance problems as lists of possible downcasts have to be maintained. In an object or aspect-oriented context, the use of polymorphism seems preferable. Second, a role could be defined for each class that needs specific treatment by all *visitors*. In our case this would lead to one role per graphical object and would lead to an unnecessary level of abstraction. Third, the role-based approach for the *visitor* pattern could be abandoned. Roles in a *visitor* pattern can offer a benefit if the nodes or groups of nodes in the underlying *composite* structure allow a uniform treatment by all *visitors*. Since this is not the case in our visitor pattern implementation, we removed the role level in that pattern completely.

4.1.1.1 Object-Oriented Implementation

The object-oriented implementation consists of 23 components (20 classes and 3 interfaces), 590 lines of code and constitutes the largest of all programs. Figure 1 shows an overview of the object-oriented version for those classes and interfaces that participate in either the *composite* or the *visitor* pattern. The class *CompositeProtocol* implements the *composite* pattern and offers the inner classes *Component*, *Leaf* and *Composite* as well as methods to add objects to or remove objects from a *composite*. With the help of inner classes, concern related code can be grouped together – in this case code that belongs to the *composite* pattern implementation. In Figure 1 inner classes are presented as a *composite* aggregation with the stereotype <<*inner class*>> or, in the case of inner interfaces, with <<*inner interface*>>. A class that defines inner classes or interfaces is marked by an underlying frame surrounding not only itself but also all inner classes and interfaces. Class *Line* derives from class *Leaf* denoting that it plays the *leaf* role in the *composite* pattern. Class *Objectgroup* is the only class that derives from class *Composite*, since it plays the *composite* role. The assignment of a role to a class in this object-oriented version is thus managed through inheritance. *Component* and *Composite* also implement functionality to add or remove a child object from the *composite* pattern

The interface *GraphicalObjectVisitor* declares visit methods that all concrete *visitors* need to implement and also defines the interface *Visitable* which declares an *accept* method all nodes need to implement. Combining both interfaces in one file stresses the fact that both interfaces belong to the same concern, i.e. to the same pattern implementation. Each concrete *visitor* performs an implementation, for example *GraphicalObjectShift*, implements *GraphicalObjectVisitor*.

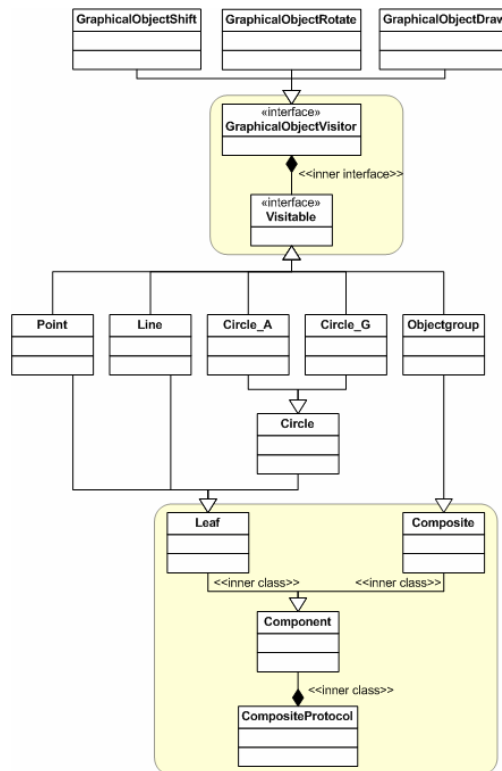


Figure 1. UML diagram of the object-oriented Graphics library

Even though *visitor* and *composite* have been implemented with the help of two different class or interface hierarchies, they are not completely separated: each *visitor* needs to route calls to a *composite* down to its children. For example, if all graphical objects of an *Objectgroup* are supposed to be rotated, the rotate *visitor* needs to visit all child graphical objects of a given *Objectgroup* object. Also, Figure 1 does not show the *uses* relationship between all three *visitors* and all graphical objects. These relationships have been removed from this diagram to improve readability.

4.1.1.2 Aspect-Oriented Implementation

The aspect-oriented implementation consists of 24 components (2 aspects, 5 interfaces and 17 classes) and 579 lines of code. Figure 2 shows an overview of the aspect-oriented version for those components that participate in either the *composite* or the *visitor* pattern. The *composite* pattern is implemented by the use of an abstract aspect *CompositeProtocol*. Similarly to the object-oriented version it defines a *leaf* and a *composite* role derived from *Component* and additional functionality to add objects to or to remove objects from the *composite*. The boxes represent one modular unit in the implementation, i.e. an aspect that defines inner interfaces. Only in the UML representation have the inner interfaces been separated from their defining aspect to achieve a cleaner representation.

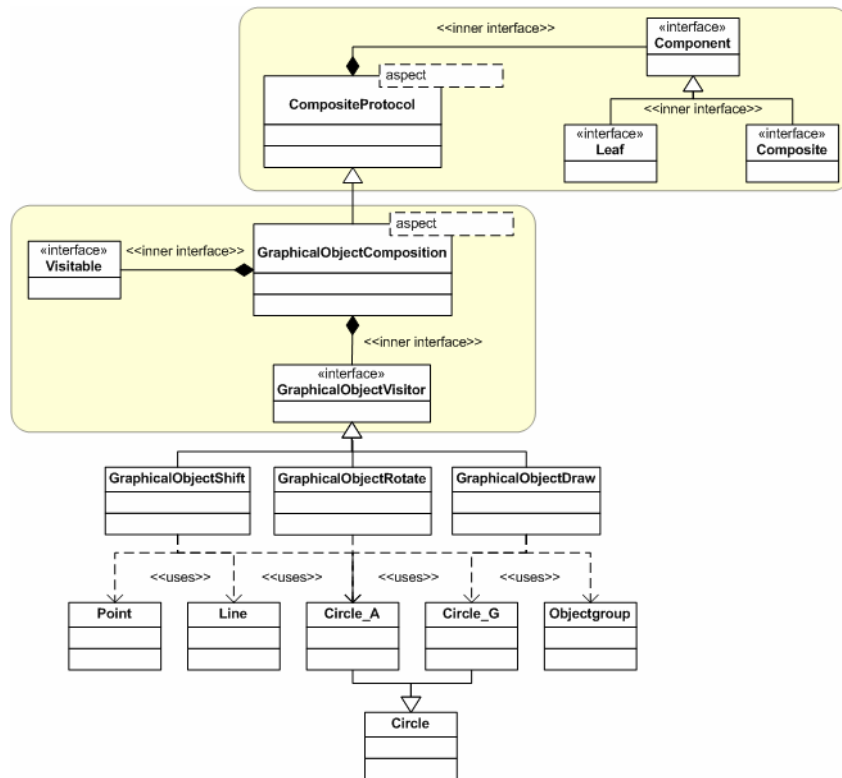


Figure 2. UML diagram of the aspect-oriented Graphics library

Figure 3 shows the object and the aspect-oriented implementation of the *composite* protocol. The comparison shows that they both achieve similar modularity and level of abstraction. They both define two roles and the same additional functionality to add children to and to remove children from the pattern. The actual assignment of a class to a role in the aspect-oriented system is not managed through inheritance but through an inter-type declaration in an additional aspect *GraphicalObjectComposition* which is derived from *CompositeProtocol* and which implements the concrete instance of the *composite* pattern. *GraphicalObjectComposition* also defines the interface *Visitable* for those nodes that can be visited. In contrast to the object-oriented implementation, nodes like *Point* or *Line* do not have to implement this interface, but *GraphicalObjectComposition* uses *declare parents* statements to create this inheritance relationship within the aspect in a non-intrusive manner. The nodes contain no code that links them to the *visitor* or the *composite* pattern. The main features of this approach are that the abstract *composite* pattern logic is modularized in an abstract aspect to achieve reusability, the specific pattern instance is managed in *GraphicalObjectComposition* and classes do not need to implement pattern specific code to participate in the *composite* pattern. The aspect-oriented implementation is similar to the object-oriented version in the sense that both the *composite* and the *visitor* pattern achieve a similar modularity. The *Component*, *Leaf* and *Composite* hierarchy is similarly modularised in both systems as well as the *Visitable* interface and the base class/aspect for all *visitors*. The main differences between the two implementations are the decoupled nodes of the *composite* pattern (*Point*, *Line*, *Circle* and *Objectgroup*). Such decoupling has another effect on *Objectgroup*: the semantics of this class are determined by the role that it plays in the *composite* pattern. It has no meaning outside the pattern which leads to an artificial, empty class in the aspect-oriented implementation, since all *composite* related code can be found in *CompositeProtocol* or *GraphicalObjectComposition*.

The original C++ version of the Graphics Library was implemented without a *visitor*. Member methods implemented functionality such as drawing, rotating or shifting in every single graphical object. The result was a system in which manipulation code was highly scattered across the application. Also, each manipulation concern crosscut the *composite* concern in the sense that for each kind of manipulation, the *composite* class, in this case class *Objectgroup*, needed to route calls to its children: in order to shift an object group, each single member of the group needed to be shifted.

<pre> public class CompositeProtocol { protected static class Component { // Vector to hold all child components. private Vector children = new Vector(); protected Vector getChildren(){ if (children == null){ children = new Vector(); } return children; } // Defines empty method on COMPONENT. public void addChild(Component component){} // Defines empty method on COMPONENT. public void removeChild(Component component){} public Enumeration getAllChildren(){ return getChildren().elements(); } } public static class Composite extends Component { // Defines concrete method on COMPOSITE. public void addChild(Component component) { getChildren().add(component); } // Defines concrete method on COMPOSITE. public void removeChild(Component component) { getChildren().remove(component); } } public static class Leaf extends Component {} } </pre>	<pre> public abstract aspect CompositeProtocol { public interface Component {} // Vector to hold all child components. private Vector Component.children = new Vector(); private Vector Component.getChildren() { if (children == null) { children = new Vector(); } return children; } // Defines empty method on COMPONENT. public void Component.addChild(Component component) {} // Defines empty method on COMPONENT. public void Component.removeChild(Component component) {} public Enumeration Component.getAllChildren() { return getChildren().elements(); } protected interface Composite extends Component {} // Defines concrete method on COMPOSITE. public void Composite.addChild(Component component) { getChildren().add(component); } // Defines concrete method on COMPOSITE. public void Composite.removeChild(Component component) { getChildren().remove(component); } protected interface Leaf extends Component {} } </pre>
---	--

Figure 3. Object and aspect-oriented implementation of the *composite* protocol

The composition pattern will always crosscut the *visitor* pattern or, if no *visitor* is used in an implementation, it will crosscut the manipulation concern. In an aspect-oriented version of the software without a *visitor*, in which the *composite* pattern is fully localized and separated from the rest of the application, code that manipulates a graphical object needs to be introduced to the aspect that defines the *composite*: shifting code for a leaf object can be found at the corresponding leaf object itself, but shifting code for a *composite* object can be found where the *composite* is defined. In a system that uses a *visitor* to modularize each manipulation concern, such as drawing, shifting or rotating, these concerns do not crosscut the *composite* concern anymore, but the *visitor* pattern now does: the *visitor* needs to be routed to the children of a *composite* to apply an object manipulation.

4.2 Boolean Formulas Library: Composite/Visitor

The Boolean Formulas library [9] is a library for representing boolean terms (AND, OR, XOR, NOT and variables), for printing the formulas in two different styles, in infix notation on a single line or prefix notation on multiple lines with indentations and for evaluating the formulas. Like the Graphics library the Boolean Formulas library also uses the *composite* and *visitor* pattern, but implements simpler versions.

4.2.1 Design Considerations

The aspect-oriented version does not use an abstract *composite* protocol that defines the roles that a class can play in the pattern. Instead, only one aspect directly implements the *composite* logic without any role assignment, leading to a less abstract solution. Also, in contrast to the Graphics Library, we removed all empty classes as a result of moving all *composite* logic into an aspect. The object-oriented version exhibits a comparable level of modularity and abstraction.

The motivation behind the aspect-oriented implementation is to preserve a separation of business logic and pattern code. Similarly to the more abstract, role-based approach for the visitor and composite pattern, the participating classes of the visitor pattern in this simpler version do not contain any pattern related code. Understandability may be increased. However, this solution does not offer the same degree of reusability as the more abstract version with a pattern protocol: each instance of the visitor pattern has to be re-implemented. There is no common protocol that defines the logic of a pattern.

4.2.1.1 Object-Oriented Implementation

The object-oriented implementation consists of 355 lines of code and 13 components (11 classes and 2 interfaces). Figure 4 shows an overview of those classes and interfaces that participate in either of the two design patterns. The implementation of the *composite* and *visitor* pattern is straightforward. Class *NFoldTerm* implements the *composite* pattern with classes *OrTerm* and *AndTerm* deriving from this class. The interface *Formula* defines an *accept* method for the *visitor* pattern which all boolean formula classes implement. All *Visitors* implement the *FormulaVisitor* interface. As far as modularity is concerned, code that belongs to the *visitor* pattern is scattered across all classes except *NFoldTerm*. Code that belongs to the *composite* pattern can be found in *NFoldTerm* and in all *visitors* since each *visitor* needs to route calls to a *composite* to the children of that class. Hence, the object-oriented implementation does not achieve a high separation of pattern related code.

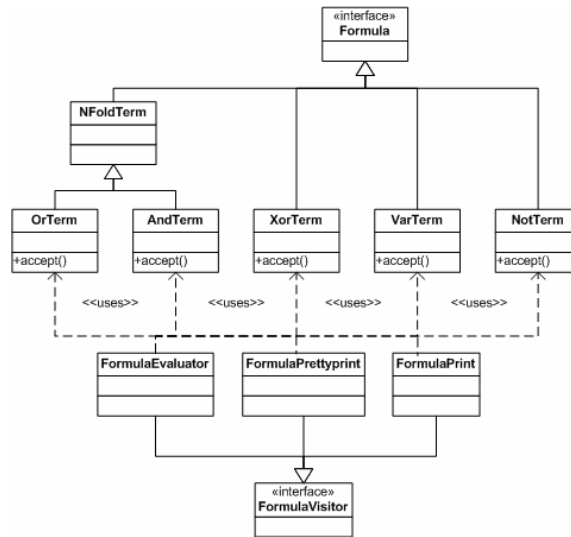


Figure 4. UML diagram of the object-oriented Boolean Formulas Library

4.2.1.2 Aspect-Oriented Implementation

The aspect-oriented version consists of 331 lines of code and 14 components (9 classes, 3 interfaces and 2 aspects). Figure 5 shows an overview of those 13 classes, interfaces and aspects that participate in either of the two design patterns. It exhibits a stronger modularisation of the two patterns than the object-oriented implementation. The aspect *FormulaCompositeProtocol* implements the *composite* pattern with help of an inner interface *NFoldTerm*. *OrTerm* and *AndTerm* implement this interface through inter-type declarations and *declare parents* statements in the aspect. The aspect *FormulaVisitorProtocol* defines an interface *FormulaVisitor* that all *visitors* need to implement and also introduces *accept* methods to all classes representing a boolean expression. The modularity is similar to the object-oriented version except that all code related to the *visitor* pattern is now modularised in one aspect and the *visitors* only. Code relating to the *composite* pattern can be found in one aspect and the *visitors* since they have to visit the children of *composite* classes.

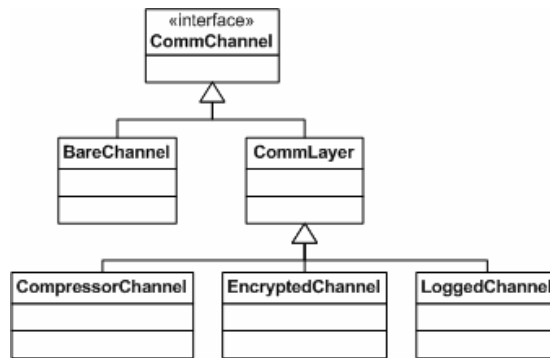


Figure 6. UML diagram of the object-oriented Communication Channel Library

4.3.1.2 Aspect-Oriented Implementation

The aspect-oriented version consists of 383 lines of code and 11 components (5 classes, 1 interface and 5 aspects). Figure 7 shows an overview of those classes, interfaces and aspects that participate in the *decorator* pattern. One of the main properties of the aspect-oriented implementation is the fact that the *decorator* pattern is directly supported by an aspect, i.e. by *around advice*. This kind of advice may be executed instead of a certain method execution or call. The AspectJ language supports the decision to continue with the original execution or call or to suppress it. The advice code can add functionality such as logging or encryption and then pass on the changed data to the original method and continue with the original control flow. This aspect-oriented core behaviour is exactly what a *decorator* offers: to provide added functionality in a transparent way.

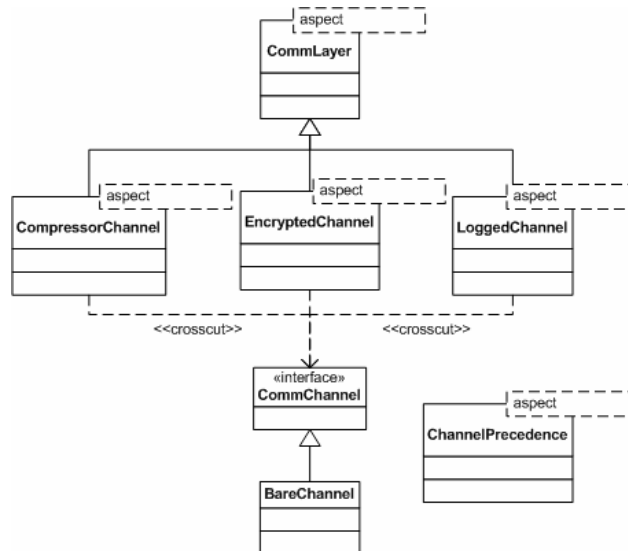


Figure 7. UML diagram of the aspect-oriented Communication Channel Library

Our implementation of the aspect-oriented *decorator* consequently consists of a base aspect *CommLayer* that describes all methods each *decorator* should ‘decorate’ through the use of pointcuts [8]. Each *decorator* derives from this base aspect and implements an around advice for each pointcut defined in the base aspect. This implementation exhibits similar qualities to the object-oriented implementation. The difference is that the *decorators* are no longer part of the base channel hierarchy. In order to guarantee the same order in which the *decorators* act on the base channel, a separate aspect *Channel Precedence* has been implemented that defines this order.

Hannemann [8] points out that the aspect-oriented implementation of the *decorator* pattern has some limitations. As stated previously, the aspect-oriented *decorator* loses its dynamic properties such as dynamic reordering. In order to guarantee a similar implementation between the object and the aspect-oriented version, the following list gives an overview of observed limitations and how we tried to remedy them:

1. *Order*. The order of advice that is applied to methods can be determined by a *declare precedence* statement in an aspect at compile time. During execution, this order cannot be changed and there is only one order that

can be defined. Hence it is not possible to apply the same *decorators* in a different order to different base objects. This problem could not be resolved.

2. *Multiplicity*. Defining the number of instances of a *decorator* working on a certain number of base channel instances is problematic. The following cases need to be considered:
 - a) One *decorator* instance per base channel instance (1:1), for example, one instance of the logging *decorator* acts on one instance of the base channel. Such a scenario could be achieved with a *perTarget* statement in the AspectJ language. However, since in such a case the aspect is not a singleton it seems difficult to initialize a certain aspect instance from outside the aspect as might be needed for the logging *decorator*.
 - b) One *decorator* instance for multiple base channel instances (1:n), e.g. one logging *decorator* instance decorates several instances of a base channel. This problem can be overcome by implementing a list that keeps track of all active *decorator* aspects. Every time *decorator* advice is executed, this list needs to be checked to find whether the *decorator* is turned on or off for a specific base channel and depending on this decision to continue execution or not. The problem with this solution is that every time a decorated method is being called, each *decorator* aspect has to find out which *decorators* are active even if none have been assigned. This is certainly an unnecessary overhead.
 - c) Several *decorator* instances for one base channel instance (n:1), e.g. several logging *decorator* instances decorate a single base channel instance. In order to satisfy this requirement, the instantiation of aspect *decorators* needs to be controlled. However, since the flexibility of creating aspect instances is limited in AspectJ, such behaviour can hardly be implemented without redefining the aspect *decorator's* behaviour: the aspect would merely act as a selection component for the correct *decorator* instance. For example, the logging *decorator* aspect would not provide logging functionality, but would select the instance of a logging *decorator* class that is needed for a certain base channel instance.

Overall, the aspect-oriented implementation presented a lot of problems that needed to be overcome to assure a similar functionality to the object-oriented version leading to an administrative overhead. Issues like the order of decorators inhibit a functionally equivalent implementation of both systems. The observed overhead will be likely to have an impact on performance and on maintainability. One effect is immediately apparent: our AO solution is much larger than the OO one (383 NCLOC versus 275 NCLOC).

4.4 StockTicker Library: Observer

The StockTicker program is used for directing a continuous stream of stock trades (title, number, unit price) from a stock market to one or more displays. The displays advertise the information or part of it. The data (of type *StockTrade*) come from a *FileTradeStream* or a *WebTradeStream* which simulates the trade data stream by reading the data repeatedly from a file or from a (fake) web service. The types of the displays are *SimpleStockTicker*, *VolumeStockTicker* or *ConsoleStockTicker*.

4.4.1 Design Considerations

For the implementation of the StockTicker library, the *subject/observer* pattern has been used. The displays are the *observer* while the two different trade streams are the subjects being observed.

The motivation behind the aspect-oriented observer pattern is again a stronger separation of business and pattern code. The subjects and observers do not contain any pattern related code which increases their reusability. All pattern code is kept within aspects and the abstract, role-based implementation enhances the reusability of the pattern code [8].

4.4.1.1 Object-Oriented Implementation

The object-oriented implementation consists of 317 lines of code and 11 components (8 classes and 3 interfaces). Figure 8 shows an overview of those classes and interfaces that participate in the *subject/observer* pattern. In the object-oriented implementation, the roles *subject* and *observer* are apparent as interfaces that participating classes need to implement. The *observer* interface declares an update method that takes a subject object. Each *observer* needs to downcast a subject to its dynamic type in order to perform the update logic. *ConsoleStockTicker*, *SimpleStockTicker* and *VolumeStockTicker* implement the *Observer* interface and *FileTradeStream* and *WebTradeStream* implement the *Subject* interface.

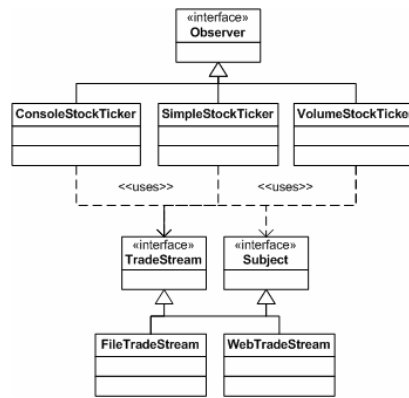


Figure 8. UML diagram of the object-oriented StockTicker Library

4.4.1.2 Aspect-Oriented Implementation

The aspect-oriented version consists of 317 lines of code and 14 components (8 classes, 4 interface and 2 aspects). Figure 9 shows an overview of those classes, interfaces and aspects that participate in the *subject/observer* pattern. The aspect-oriented implementation consists of two aspects. *ObserverProtocol* represents the *observer* pattern by defining the *Observer* and the *Subject* interface. *StockTickerObserver* derives from *ObserverProtocol* and implements the concrete *observer* pattern instance for stock ticker displays. In particular, it crosscuts all stock ticker display classes by declaring a new inheritance relationship between the display classes and the *Observer* interface as well as between the trade stream classes and the *Subject* interface. Thus, a localized implementation of the *observer* pattern has been achieved. It mirrors the implementation by Hannemann et al. [8].

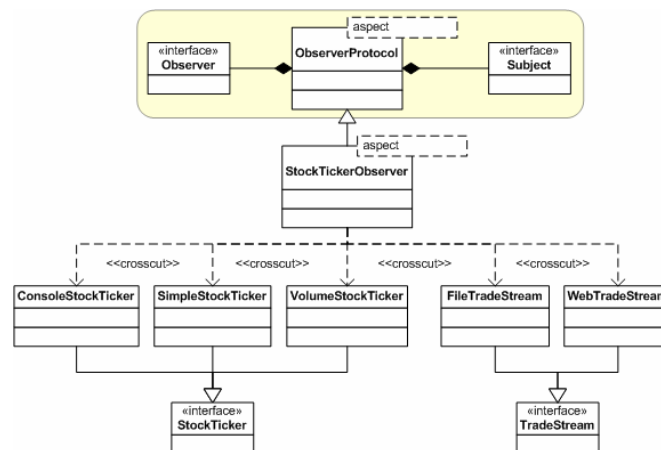


Figure 9. UML diagram of the aspect-oriented StockTicker Library

One of the strengths of the aspect-oriented *observer* implementation is the fact that the participants in this pattern, i.e. those classes that play the *observers* and those that play the *subjects* role, do not need to be aware of their role. The entire pattern logic is managed by the abstract *ObserverProtocol* aspect and concretised in the *StockTickerObserver* aspect. However, such an approach might lead to casting problems: the *observer* protocol declares an abstract, generic update method with a *Subject* and an *Observer* class as formal parameters and which needs to be implemented in *StockTickerObserver*. The semantics is such that the state of that particular subject has changed and so the *observer* needs to be notified. However, since this method is designed in a generic way, downcasts or reflection have to be used to find out about the dynamic type of each subject and *observer* pair. Depending on their type, the correct update method on the *observer* also needs to be called. Such an approach might lead to long switch statements if useful base classes are not available. In order to avoid this problem, we kept the *StockTicker* interface which defines an update method that all stock ticker displays need to implement. It is almost identical to the *Observer* interface from the object-oriented implementation, which also defines an update method. However, the *StockTicker* interface expects a *TradeStream* object whereas the *Observer* protocol expects a *Subject* object. The downcast problem exists in the object-oriented version, too, but only in a weaker form. Since each display needs to downcast their own *Subject* objects, no downcast is necessary

for *Observer* objects. Overall, the aspect-oriented version offers the benefit of a separated implementation of the *observer* pattern, but may lead to long cast statements if suitable base classes are missing.

5. COUPLING MEASUREMENT

The concept of coupling for structured design was first introduced by Stevens et al. as “the measure of the strength of association established by a connection from one module to another”[10]. A connection was defined as a reference to some label or address defined elsewhere. In order to avoid "ripple effects", where a change in one component results in errors in another component, a common design aim is to achieve low coupling between components, i.e. a low number of connections [2, 10]. This early concept of coupling was applied to the object-oriented paradigm by Coad and Yourdon to account for inheritance [4]. Briand et. al [3] suggested considering different coupling types independently, i.e. coupling caused by the use of type identifiers versus coupling caused by the invocation of methods, for example. We follow this recommendation and only consider coupling through the use of type identifiers, i.e. coupling occurs if a component uses another component's type identifier as a return type in methods or as types of local variables. This type of coupling is only one of many as outlined in [1]. The nature of this coupling relationship has an impact on maintainability. Changes to the identifier of a type may need adjustment of all components that use the given component's type identifier. The use of a type identifier which has been defined as an inner class or an inner interface will be regarded as a coupling relationship with its outmost class [11]. This may lead to a reduction of coupling if a coupling connection is counted only once – in this case the coupling connection to the outmost class of an inner component.

Program	Coupling in Java	Coupling in AspectJ	% difference
Graphics Library (<i>composite/visitor</i>)	38	32	-16%
Boolean Formulas Library (<i>composite/visitor</i>)	39	35	-10%
Communication Channel Library (<i>decorator</i>)	24	29	+21 %
StockTicker Library (<i>observer</i>)	24	24	0%

Table 1. Coupling measurement for the StockTicker Library

Table 1 gives an overview of all four programs and the amount of coupling that was measured in the Java and the AspectJ version. The last column shows the reduction or increase of the measured coupling. The *composite/visitor* versions achieved a reduction of coupling for AspectJ of 16% and 10%, whereas our *decorator* implementation led to an increase of 21% in the aspect-oriented version. Only the *observer* implementation did not exhibit any change in coupling.

The reduction of coupling in the aspect-oriented versions can be explained by the reduced coupling of the nodes of the *composite* pattern. We discovered this reduction of coupling in the aspect-oriented version even though we used inner classes and interfaces which helped to reduce coupling in the object-oriented implementation. The increase in coupling of the *decorator* pattern is mostly due to the aspect that controls *precedence*, since it uses the type identifiers of all aspects. The *observer* pattern does not show any change in coupling although there is a decrease of coupling among the participants of that pattern. This reduction is outweighed by the aspect that implements the concrete *observer* instance, as it defines the roles for each participant. However, a reduction of coupling could be achieved in the aspect-oriented system through the addition of more subject participants.

Overall, the aspect-oriented *composite* and *visitor* pattern seem to be good candidates for a beneficial use of design patterns, whereas the *decorator* pattern should probably be applied with caution. The coupling changes in the *observer* pattern seem neutral, but an advantage may be realized if the pattern is scaled up.

6. CONCLUSIONS

In this paper we investigated the object-oriented and aspect-oriented implementations of four programs that are all centred around design patterns. We measured coupling and found a reduction of coupling in two of the four programs. One did not show any change in coupling and one program exhibited more coupling in the aspect-oriented version than in the object-oriented version.

An interesting finding was the fact that inner classes and interfaces can also help object-oriented implementations to reduce coupling and improve localisation. The *CompositeProtocol* class in the object-

oriented Graphics Library defined inner classes for each role of the *composite* pattern. The use of inner classes helped to reduce coupling since the inheritance relationship between *Leaf*, *Composite* and *Component* was not counted as a coupling connection and kept pattern related code in one place.

As far as the *composite* and *visitor* patterns are concerned, both the more abstract and the simpler version led to reduced coupling. Again, the question of which implementation to prefer is a trade-off between a higher cognitive complexity and the reusability of the abstract pattern protocol. The aspect-oriented *observer* pattern might suffer from cast problems more than the AO version, since it needs to downcast *subject* and *observer* classes to its dynamic type if no suitable base classes are available.

Our measurement results show that for the AO implementation a reduction of coupling could be found for the *composite* and *visitor* pattern. Our *observer* pattern implementation currently shows no difference in coupling between the object and the aspect-oriented version, but we would anticipate less coupling in the aspect-oriented version when more subject classes have been added. *Decorator*, however, does show more coupling in the aspect-oriented version. These measurements confirm the findings by Garcia et al. [7], who also found a decrease of coupling for the aspect-oriented implementations of *composite*, *visitor* and *observer*.

The lessons learnt in this case study are directed at the *decorator* pattern. It seems that a lot of effort has to be invested in this pattern to achieve an aspect-oriented implementation that offers a similar flexibility to its object-oriented counterpart. For some dynamic properties this seems impossible. Again, a trade-off is necessary between the separation of concerns in the aspect-oriented solution, where the *decorators* are fully detached from the rest of the application and the flexibility gained in the object-oriented version.

For the other patterns, we found the separation between aspect and pattern code to be another valuable contribution. These patterns support reusability of their participants in contexts in which they don't interact with patterns. Localizing pattern related code might be a step towards increasing the maintainability of design patterns.

Our conclusion is that more research needs to be directed at those design patterns that exhibit potentially beneficial properties, i.e. *composite*, *visitor* and *observer*. Our future work will thus focus on experimental work to gather evidence about the maintainability of software which uses aspect-oriented implementations of such patterns. We also hope that this case study will encourage more empirical work into the beneficial applications of design patterns.

REFERENCES

- [1] M. Bartsch and R. Harrison, "A Coupling Framework for AspectJ - Extended Abstract," presented at Evaluation and Assessment in Software Engineering (EASE), Keele, UK, 2006.
- [2] V. R. Basili, L. C. Briand, and W. L. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators," *IEEE Transactions on Software Engineering*, vol. 22, pp. 751-761, 1996.
- [3] L. C. Briand, J. W. Daly, and J. K. Wust, "A Unified Framework for Coupling Measurement in Object-Oriented Systems," *IEEE Transactions on Software Engineering*, vol. 25, pp. 91-121, 1999.
- [4] P. Coad and E. Yourdon, *Object-Oriented Design*, 1st ed: Prentice Hall, 1991.
- [5] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, 2nd ed: International Thomson Computer Press, 1996.
- [6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading: Addison-Wesley, 1997.
- [7] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa, "Modularizing Design Patterns with Aspects: A Quantitative Study," presented at International Conference on Aspect-Oriented Software Development (AOSD), Chicago, USA, 2005.
- [8] J. Hannemann and G. Kiczales, "Design Pattern Implementations in Java and AspectJ," presented at ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 2002.
- [9] L. Prechelt, B. Unger, W. F. Tichy, P. Brössler, and L. G. Votta, "A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions," *IEEE Transactions on Software Engineering*, vol. 27, pp. 1134-1144, 2001.
- [10] W. Stevens, G. Myers, and L. Constantine, "Structured Design," *IBM Systems Journal*, vol. 13, pp. 115-139, 1974.
- [11] M. Stojanovic and K. El-Emam, "ES2: A Tool for Collecting Object-Oriented Design Metrics from C++ and Java Source Code," National Research Council Canada NRC/ERB-1088, NRC 44888, 2001.
- [12] M. Vokáč, W. Tichy, D. I. K. Sjøberg, E. Arisholm, and M. Aldrin, "A Controlled Experiment Comparing the Maintainability of Programs Designed with and without Design Patterns – A Replication in a Real Programming Environment," *Empirical Software Engineering*, vol. 9, pp. 149-195, 2004.