# Reflections on the paper:
## “Design Pattern Implementation in Java and AspectJ ”
## by J. Hannemann and G. Kiczales

Benjamin Stopford
Email: Benjamin.Stopford@BarclaysCapital.com
07968 702589

Are there visible improvements in the AspectJ pattern implementations arising from the cross-cutting nature of design patterns?

### Cross-Cutting and Aspects

Aspect orientated programming involves the separation of concerns within an application so that they do not clutter or cut across one other. The Wikipedia definition [5] states:

> “In computer science, cross-cutting concerns are aspects of a program, that do not relate to the core concerns directly, but are needed for proper program execution.”

As an example let us consider a telecommunications application. This might have a core concern of “routing calls”, but code for other operations such as “timing” and “billing” those calls might be intertwined in the whole “routing” object hierarchy. Here the “billing concern” overlays the “routing concern” meaning that the code that implements each is crosscut (Fig. 1(a)). This intertwining of code for separate concerns, know as ***Cross-Cutting***, cannot be removed with regular programming methods.

Aspect Orientation allows the extraction of duplicated calls from multiple positions in the control flow into single Aspect commands that are woven back in at compile time. This leaves client code free of tertiary concerns that would otherwise cut across it (fig. 1(b)).

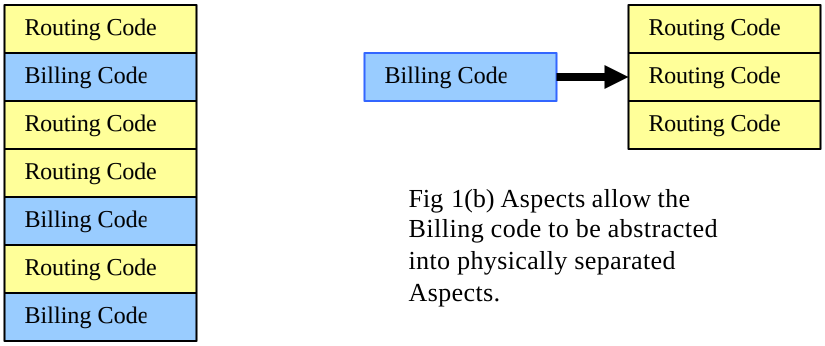

Fig 1(b) Aspects allow the Billing code to be abstracted into physically separated Aspects.

Fig 1(a)
Tangled code for Billing and Routing in the standard OO implementation.

### Patterns and Roles

Hannemann et al have observed that there are cross-cutting relationships between different roles within many of the GoF [2] patterns. They segregate these roles into two types. Firstly “Defining Roles” are those for which the code is completely encapsulated in the pattern itself. A good example is the Façade pattern which has no influence on clients other than to provide an interface with which they can interact. Such roles leverage little from an Aspect Orientated approach as their encapsulation means they rarely crosscut client code. The second role type observed by Hannemann are “Superimposed Roles”. These generally see improvements with the introduction of Aspects due to there being cross-cutting between the different roles which are superimposed as in the telecommunications example above.

### Cross-Cutting of between Pattern and Participant Classes

The first and probably most fundamental cause of cross-cutting within the standard OO (non-Aspect) patterns arise due to the superposition of the pattern role over the primary role of the participant class, which is likely to be something completely unrelated.

For example the class that is participating as a subject in the Observer Pattern is likely to have other responsibilities such as a regular programming task like “being a timer”. This class has a superimposition of roles, the role of being a Subject in the pattern and the role of “being a timer”. Hannemanns et al refer to this as the “Participants having their own responsibilities and justification outside the pattem context” [7]. The manifestation of this in the regular java implementation is a plethora of notifyObservers() methods within subject classes. Within the Aspect implementation however the code is modularised into a single class leaving the Subject with no knowledge of its participation in the pattern at all.

Such cross-cutting concerns are also demonstrated well by the Chain of Responsibility Pattern. This pattern allows an object to send a command without knowing what objects will receive it. The benefits due to cross-cutting are again revealed due to the concern of the pattern being separated from the concern of the participant classes i.e. each participant in the chain need not be aware of its role in the pattern.

The key point is that code modules for roles such as Subjects have no interest in containing code that facilitates their behaviour in the pattern. Hence the presence of any such code for this role is of no relevance and this crosscuts their primary role.

Hanneman et al classify this kind of cross-cutting further into two subsets :-

1. **Roles that Crosscut Participant Classes:** This defines the concept of different roles being intertwined as described above. Examples are the cross-cutting of the pattern roles, such as being a Subject in the Observer or Leaf in the Composite etc, with the base role of the participant class such as “being a timer”.
2. **Conceptual Operations Cross-Cutting Methods in one of more Classes:** This refers to the actual implementation of ‘one to many’ relationships across multiple classes which generally result from (1). This level of cross-cutting is often modularised into a single class in the Aspect implementations. In the Observer example this is typified by the modularisation of the scattered refresh() calls into a single subjectChange() pointcut.

**Cross-Cutting of Shared Participants between Multiple Pattern Instances**

The second level of cross-cutting pointed out by Hannemann et al lies in the improvement found when multiple patterns (or pattern instances) act upon the same subjects. For example there may be two instances of a pattern which each treat the *same class* or *method* with *different roles*. This “Pattern Composition” introduces additional cross-cutting issues.

For example considering multiple instances of the Observer pattern, certain participant classes may behave as Observer in one pattern instance and Subject in another. This cross-cutting over shared participants can be removed in the AspectJ implementation as each of the pattern instances access a single set of mapping points in the programs execution via Point cuts.

**Cross-Cutting and the Classification of Roles in Patterns**

Aspect orientated programming is driven by the desire to split code that applies to different concerns so that it does not crosscut the code of the core concern. This is typified by the superposition of the pattern role and participant role in the Observer pattern. The Observer example is successful because there is aclear argument for the roles imposed by the pattern being logically separate from the core concern of most Subject and Observer classes. Hanneman et al describe the various roles within the GoF patterns but there is no discussion of which concerns those roles really lie in.

The next question I will try to explore is: Are the roles observed in the patterns really in different areas of concern to the roles of the participant classes?

To explore this let us look at the Composite pattern. Put briefly the composite pattern supplies a mechanism through which you can compose a single object from a collection of other objects to produce a composition of behaviour. Hannemann et al present this pattern via a fictitious file system where a directory can be composed of files and other directories. The composite component allows each element to be treated in a similar way i.e. the pattern provides the mechanism that allows files and directories to be added to other directories etc.

The view put forward by Hannemann et al is that the pattern has two superimposed roles, the Composite and the Leaf. Cross-Cutting of roles can occur when the role of "being a Leaf" crosscuts the role that each class is designed to do, in this case "being a file or directory". The problem arises that whilst this is true in the case of the Observer pattern, in many pattern implementations the pattern is tied to the same concern as the implementing class.

In this file/directory example it could be said that the file should not need to know about the various methods that facilitate its behaviour and position in the file system. However another argument might be that the position of a file in a file system is likely to be considered pertinent to the core concern of being a file.

The Strategy pattern is another example with questionable disparity between roles simply due to the role of the pattern being intrinsically tied to the role of the participants. Hannemann et al use a sorter as an example which can be initialised with either a bubble or linear sort strategy. The aspect implementation removes the setting of the strategy to an aspect so that the core code for the sorter is not crosscut by it. However it is difficult to think of a situation where the sorter would lie in a separate concern to the sort type. Put in a more abstracted fashion the strategy for a class may not always exists in a separate concern to that of the class itself.

This is not to say that Hannemann et at are incorrect. They have defined roles within patterns and described methods through which the cross-cutting between those roles and the roles of implementing classes can be reduced.

However my opinion is that cross-cutting between patterns and participants should only be considered on an instance by instance basis where the core concern of the implementing class is known. Only then can you decide whether the pattern roles are truly distinct from the core concerns. Furthermore patterns, such as the strategy pattern and others, imply a usage that connects them intrinsically to their implementation. In such cases the modularisation of cross-cutting concerns in the AspectJ implementations would not really be separating truly disparate roles.

In conclusion there is clear benefit in cross-cutting behaviour in some of the patterns suggested. However for the removal of cross-cutting to be of real benefit there needs to be an orthogonal relationship between the roles of the intertwined code streams. Roles such as the Subject in the Observer pattern are clearly orthogonal to whatever that participant class might be. However most of the other patterns do not have the luxury of such clear distinctions in their roles.

Hence I feel there is still an open question as to whether Hannemann et al can really claim a general cross-cutting improvement. The very premise for Aspect development is the physical segregation of roles or concerns within an application. Unfortunately whilst there is an almost unilateral improvement in modularity throughout the pattern suite true cross-cutting enhancement is only confirmed in a select few.

Are there visible improvements in modularity of the AspectJ pattern implementations when compared to the corresponding design patterns implemented in Java?

One of the overwhelming advantages of Object Orientated programming is its ability to increase modularization. However the Object Orientated framework itself imposes limits on the level that can be supported. Software engineers have put considerable effort into furthering this aim within the OO environment using the available tools [3]. This has lead to the introduction of patterns that encapsulate solutions to recurring problems in abstracted reusable forms. These patterns increase modularization by removing code from multiple implementations and into the pattern itself. But there is a limit to how far this approach can go.

Aspects extend this aim to allow modularization, not just at the level of program flow, but also across different areas of concern within the application. Code can be abstracted into common aspect modules that would otherwise cut across several application roles. We will explore the increase in modularity via the use of Aspects in the Observer pattern.

**The Observer Pattern, Implemented in Java**

The Observer pattern defines a one-to-many relationship between a Subject (or subjects) and any number of Observers. There are thus two roles; Subject and Observer. Should the subject object change, all Observers are notified automatically. This is analogous to receiving football result texts on your mobile phone. You are the Observer and your football team is the Subject. You wish to be notified whenever your team scores a goal.

To initiate the process the Observer must first register for notifications. In our example you would ask your service provider to send you updates. Whenever your team scores a goal their state changes and your network provider would send you, and everyone else that is registered, a text that notifies you of the change in score. This corresponds to the action of the Multicaster[1] whose responsibility is to maintain the one to many relationships between Subject and the many observers, and notify them of state changes (fig 2(a)).

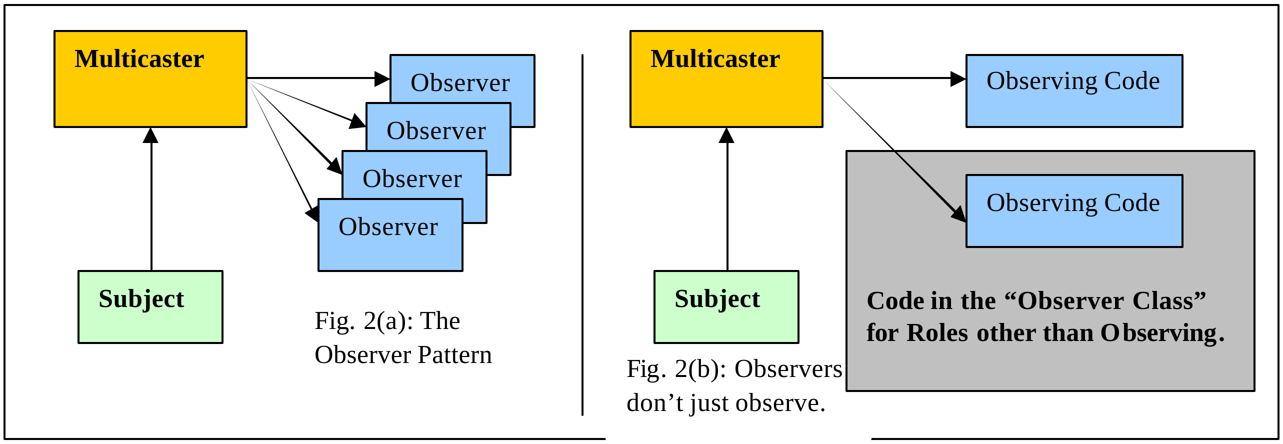

Fig. 2(a): The Observer Pattern

Fig. 2(b): Observers don’t just observe.

**Physical Differences in Observer when Implemented in Aspects**

We note that it is possible to abstract the responsibilities within the Observer pattern into two parts.

[1] Note that the concept of a Multicaster is common in implementations of the Observer pattern but is not necessarily required. Hannemann et al do not use one in their Java implementation. The Multicaster is useful for comparative purposes as it contrasts well with the Concrete Aspect in the AspectJ Implementation.

**(1) The Base Abstraction**: The base abstraction can be considered to consist of

- The framework of the pattern.
- The requirement to have a Subject and Observer.
- The mapping between them.
- The facility to update them when they change.

**(2) The Pattern Instance:** The second responsibility is the physical implementation of the pattern instance which designates those *specific* Subject and Observer classes involved.

Hannemann et al make use of this to split their implementation of the Observer pattern into two sections. The abstract aspect covers the contractual obligations (i.e. necessitating the addition of specific subject and observers) and the notification mechanism. I will denote this the "Abstract Aspect Pattern". This "Abstract Aspect Pattern" is then extended by a "Concrete Aspect" which defines the specific relationship between Subject and Observers of a specific type via the pattern i.e. the specific methods to be observed and those to be notified are defined here on a case by case basis (fig 3(b)).

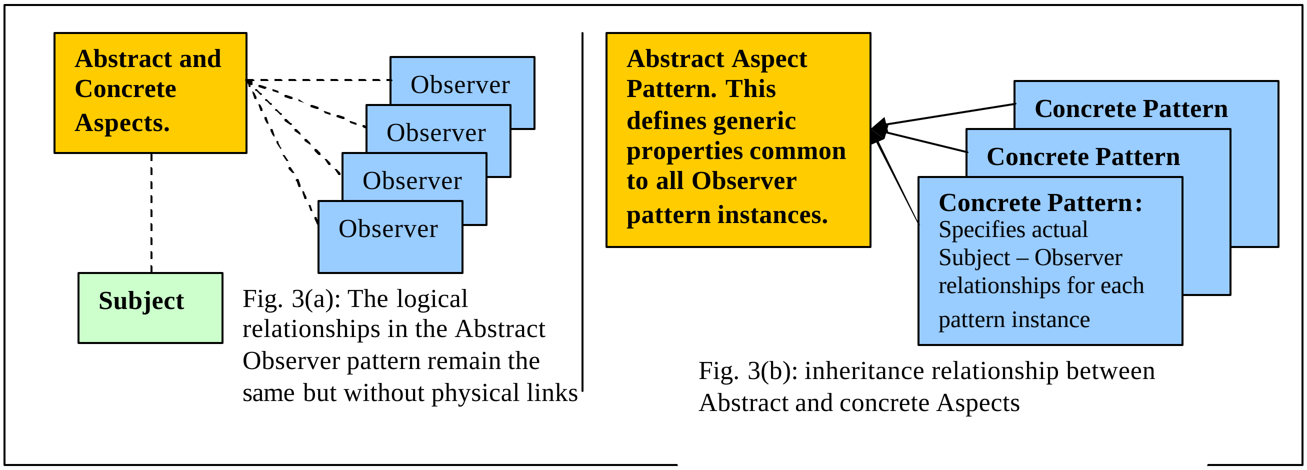
Abstract and Concrete Aspects.
Observer
Observer
Observer
Observer
Subject
Abstract Aspect Pattern. This defines generic properties common to all Observer pattern instances.
Concrete Pattern
Concrete Pattern
Concrete Pattern: Specifies actual Subject – Observer relationships for each pattern instance

Fig. 3(a): The logical relationships in the Abstract Observer pattern remain the same but without physical links

Fig. 3(b): inheritance relationship between Abstract and concrete Aspects

**Modularity Improvement within the Aspect Code.**

The first modularity improvement lies within the aspect code itself where responsibilities that are generic to the pattern can be abstracted to single, common place. This is the reason for the "Abstract Aspect Pattern". In the Hannemann et al implementation they abstract out the responsibility for maintaining the observers list, notifying observers and defining the classes that take on the observing and subject roles. Thus no matter how many Observer patterns are used between different Subject and Observers the same "Abstract Aspect Pattern" can be used, clearly increasing the modularity and reusability of the implementation.

**Modularity Improvement between Aspect and Client Code Alone.**

**(i) Modularity through Abstraction**

The second modularity increase comes from the fact that the pattern utilises the ability of AspectJ to represent crosscut concerns in single program units. As the "Abstract Aspect Pattern" and "Concrete Aspect" are implemented as Aspects their source code exists separately from the physical implementations of the Observer and Subject classes. This allows modularisation of the cross-cutting concerns within the standard pattern so that all the distributed calls to methods like refresh() and notifyObservers() in the Java implementation can be modularized into single aspect classes, in this case the updateObserver() and subjectChange() calls in the Concrete Aspect. This modularization is a direct result of aspect based operators such as pointcuts.

**(ii) Modularity though Dependency Reduction**

The Hannemann implementation increases modularity by further reducing dependencies within the client code. This is done by *inverting the flow of control* within the pattern so that only a downward dependency from the pattern to the client remains. Put another way; the pattern has an implicit dependency on the client code (the Subject and the Observer) but the client code has no reverse dependency back on the pattern (see Fig 4). This has the additional benefit that it increases (un)plugability in the client code. The client has no awareness of it's inclusion in the pattern and hence it can be added to and removed from its role in the pattern at will, but the operation of the pattern is the same.

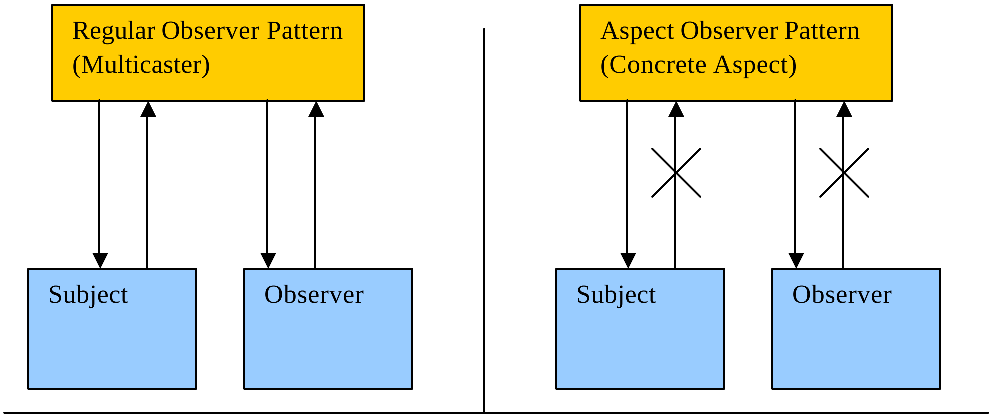
Regular Observer Pattern (Multicaster)
Aspect Observer Pattern (Concrete Aspect)
Subject
Observer
Subject
Observer

Fig 4. Demonstration of the one way (downwards) dependency between Pattern and Subject/Observer in the AspectJ implementation. The concept of a Multicaster is used for demonstrational purposes (not used in the java Hannemann et al example).

**Summary**

We have seen that there is a clear improvement in the modularity of the AspectJ implementation of the Observer pattern. This is represented over three levels reaping benefit through textual localisation, removal of code from participant classes and abstraction to common aspects. From an implementation point of view these factors allow the pattern to exist in a neutral manor, most notably, without any dependencies from the participant classes back onto the pattern itself. Considering that the Observer pattern is used frequently for communications around application frameworks such dependency removal is extremely useful. 'Framework' concerns can be completely separated from client code and this in turn facilitates a true separation of concerns.

**References**

[1] J. Hannemann and G. Kiczales, “Design Pattern Implementation in Java and AspectJ ”, in Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications OOPSLA’02), pages 161-173,November 2002.

[2] E. Gamma, R. Helm, R. Johnson and J. Vlissides, “Design Patterns. Elements of Reusable Object-Oriented Software”. Addison-Wesley, Reading, MA, 1995.

[3] G. Booch, "Object-oriented development," IEEE Trans. Software Eng., vol. SE-12, pp. 211-221, Feb. 1986.

[4] http://www.cs.wm.edu/~coppit/csci435-spring2004/lectures/17-cross-cutting-concerns.pdf

[5] http://wikipedia.org/

[6] - [1] - Section 4.1.1

[7] - [1] - Section 5.1

[8] - [1] - Section 4.1.3