Design Pattern Implementation in Java and AspectJ

Jan Hannemann
University of British Columbia
201-2366 Main Mall
Vancouver B.C. V6T 1Z4
jan [at] cs.ubc.ca

Gregor Kiczales
University of British Columbia
201-2366 Main Mall
Vancouver B.C. V6T 1Z4
gregor [at] cs.ubc.ca

ABSTRACT

AspectJ implementations of the GoF design patterns show modularity improvements in 17 of 23 cases. These improvements are manifested in terms of better code locality, reusability, composability, and (un)pluggability.

The degree of improvement in implementation modularity varies, with the greatest improvement coming when the pattern solution structure involves crosscutting of some form, including one object playing multiple roles, many objects playing one role, or an object playing roles in multiple pattern instances.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures – *patterns, information hiding, and languages;* D.3.3 [**Programming Languages**]: Language Constructs and Features – *patterns, classes and objects*

General Terms

Design, Languages.

Keywords

Design patterns, aspect-oriented programming.

1. INTRODUCTION

The Gang-of-Four (GoF) design patterns [9] offer flexible solutions to common software development problems. Each pattern is comprised of a number of parts, including purpose/intent, applicability, solution structure, and sample implementations.

A number of GoF patterns involve crosscutting structures in the relationship between roles in the pattern and classes in each instance of the pattern [6]. In the Observer pattern, an operation that changes any Subject must trigger notifications of its Observers – in other words the act of notification crosscuts one or more operations in each Subject in the pattern. In the Chain Of Responsibility pattern, all Handlers need to be able to accept requests or events and to either handle them or forward them to the successor in the chain. The event handling mechanism crosscuts the Handlers.

When the GoF patterns were first identified, the sample implementations were geared to the current state of the art in object-oriented languages. Other work [19, 22] has shown that implementation language affects pattern implementation, so it seems natural to explore the effect of aspect-oriented programming techniques [11] on the implementation of the GoF patterns.

As an initial experiment we chose to develop and compare Java [27] and AspectJ [25] implementations of the 23 GoF patterns. AspectJ is a seamless aspect-oriented extension to Java, which means that programming in AspectJ is effectively programming in Java plus aspects.

By focusing on the GoF patterns, we are keeping the purpose, intent, and applicability of 23 well-known patterns, and only allowing the solution structure and solution implementation to change. So we are not discovering new patterns, but simply working out how implementations of the GoF patterns can be handled using a new implementation tool.

Our results show that using AspectJ improves the implementation of many GoF patterns. In some cases this is reflected in a new solution structure with fewer or different participants, in other cases, the structure remains the same, only the implementation changes.

Patterns assign *roles* to their participants, for example Subject and Observer for the Observer pattern. These roles define the functionality of the participants in the pattern context. We found that patterns with crosscutting structure between roles and participant classes see the most improvement.

The improvement comes primarily from modularizing the implementation of the pattern. This is directly reflected in the implementation being textually localized. An integral part of achieving this is to remove code-level dependencies from the participant classes to the implementation of the pattern.

The implementation of 17 of the patterns is modularized in this way. For 12 of the patterns, the modularity enables a core part of the implementation to be abstracted into reusable code. For 14, it enables transparent composition of pattern instances, so that multiple patterns can have shared participants. For the 17 modularized patterns, all pattern code from some or all participants is moved into the pattern aspect, allowing those participants to be (un)pluggable with respect to the pattern.

These results – 74% of GoF patterns implemented in a more modular way, and 52% reusable – suggest it would be worthwhile to undertake the experiments of applying AspectJ to more patterns

OOPSLA '02, November 4-8, 2002, Seattle, Washington, USA.

and/or applying other aspect-oriented techniques to pattern implementations.

The rest of the paper is organized as follows. Section 2 surveys previously identified problems in design pattern implementation. Section 3 introduces the study format. In section 4, we present our AspectJ implementations and categorize the improvements we observed. Section 5 shows an analysis of our findings and observations. Related work is discussed in section 6, and Section 7 summarizes our work.

2. ESTABLISHED CHALLENGES

Numerous authors have identified challenges that arise when patterns are concretized in a particular software system. The three most important challenges are related to implementation, documentation, and composition.

Design pattern implementation usually has a number of undesirable related effects. Because patterns influence the system structure and their implementations are influenced by it [7], pattern implementations are often tailored to the instance of use. This can lead to them "disappearing into the code" [7] and losing their modularity [21]. This makes it hard to distinguish between the pattern, the concrete instance and the object model involved [15]. Adding or removing a pattern to/from a system is often an invasive, difficult to reverse change [4]. Consequently, while the design pattern is reusable, its implementations usually are not [21].

The invasive nature of pattern code, and its scattering and tangling with other code creates documentation problems [21]. If multiple patterns are used in a system, it can become difficult to trace particular instances of a design pattern, especially if classes are involved in more than one pattern (i.e. if there is pattern overlay/composition) [1].

Pattern composition causes more than just documentation problems. It is inherently difficult to reason about systems with multiple patterns involving the same classes, because the composition creates large clusters of mutually dependent classes [21]. This is an important topic because some design patterns explicitly use others patterns in their solution.

3. STUDY FORMAT

The findings presented in this paper are based on a comparative analysis of Java and AspectJ implementations of the GoF design patterns.

For each of the 23 GoF patterns we created a small example that makes use of the pattern, and implemented the example in both Java and AspectJ.[1] The Java implementations correspond to the sample C++ implementations in the GoF book, with minor adjustments to account for the differences between C++ and Java (lack of multiple inheritance, etc.). Most patterns have a number of implementation variants and alternatives. If a pattern offered more than one possible implementation, we picked the one that appeared to be the most generally applicable.

The AspectJ implementations were developed iteratively. The AspectJ constructs allowed a number of different implementations,

[1] The code is available for download at:
http://www.cs.ubc.ca/labs/spl/projects/aodps.html

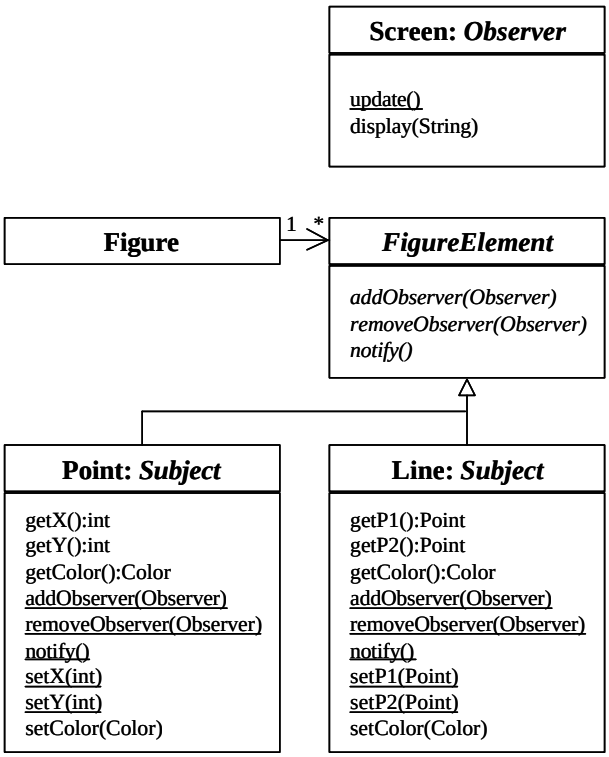

Figure 1. A simple Graphical Figure Element System that uses the Observer pattern in Java. The underlined methods contain code necessary to implement this instance of the Observer pattern.

usually with varying tradeoffs. Our goal was to fully investigate the design space of clearly defined implementations of each pattern. We ended up creating a total of 57 different implementations, which ranged from 1 to 7 per pattern. Some of the tradeoffs and design decisions are discussed in Section 4.

4. RESULTS

This section presents a comparison of the AspectJ and Java implementations of concrete instances of the GoF design patterns. Section 4.1 is a detailed discussion of the Observer pattern. We use this discussion to present properties common to most of the AspectJ solutions. The remaining patterns are presented by building on the concepts developed in Section 4.1.

4.1 Example: the Observer pattern

The intent of the Observer pattern is to "define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically"[9]. Object-oriented implementations of the Observer pattern, such as the sample code in the GoF book (p. 300-303), usually add a field to all potential Subjects that stores a list of Observers interested in that particular Subject. When a Subject wants to report a state change to its Observers, it calls its own `notify` method, which in turn calls an `update` method on all Observers in the list.

Consider a concrete example of the Observer pattern in the context of a simple figure package, as shown in Figure 1. In such a system the Observer pattern would be used to cause mutating operations to

figure elements to update the screen. As shown in the figure, code for implementing this pattern is spread across the classes.

All participants (i.e. `Point` and `Line`) have to know about their role in the pattern and consequently have pattern code in them. Adding or removing a role from a class requires changes in that class. Changing the notification mechanism (such as switching between push and pull models [9]) requires changes in all participating classes.

4.1.1 The abstracted Observer pattern

In the structure of the Observer pattern, some parts are common to all potential instantiations of the pattern, and other parts are specific to each instantiation. The parts common to all instantiations are:

1. The existence of Subject and Observer roles (i.e. the fact that some classes act as Observer and some as Subject).
2. Maintenance of a mapping from Subjects to Observers.
3. The general update logic: Subject changes trigger Observer updates.

The parts specific to each instantiation of the pattern are:

4. Which classes can be Subjects and which can be Observers.
5. A set of changes of interest on the Subjects that trigger updates on the Observers
6. The specific means of updating each kind of Observer when the update logic requires it.

We developed AspectJ code that reflects this separation of reusable and instance-specific parts. An abstract aspect encapsulates the generalizable parts (1-3), while one concrete extension of the aspect for each instance of the pattern fills in the specific parts (4-6). The reusable `ObserverProtocol` aspect is shown in Figure 2.

4.1.1.1 The roles of Subject and Observer

The roles are realized as protected inner interfaces named `Subject` and `Observer` (Figure 2, line 3-4). Their main purpose is to allow for correct typing of Subjects and Observers in the context of the pattern implementation, such as in methods like `addObserver`. Concrete extensions of the `ObserverProtocol` aspect assign the roles to particular classes (see below).

These interfaces are protected because they will only be used by `ObserverProtocol` and its concrete extensions. No code outside the aspect and extensions needs to handle objects in terms of these roles.

These interfaces are empty because the pattern defines no methods on the Subject or Observer roles. The methods that would typically be defined on the Subject and Observer are instead defined on the aspect itself (see below).

For patterns that were abstractable we had to decide where to put the role interfaces. Two locations are possible: Either as a private interface inside the abstract aspect or as a separate public interface. We made this decision based on whether the role interface introduces client-accessed functionality, i.e. exposes functionality to clients (as for Strategy, Iterator, etc.) or not (as in the Observer case). If the role has no client-accessible functionality, it will only be referenced from within pattern aspects. For that reason, we placed it in the abstract aspect. In the other case, we moved the interface into a separate file to make it easier to reference.

```
01 public abstract aspect ObserverProtocol {
02
03   protected interface Subject  { }
04   protected interface Observer { }
05
06   private WeakHashMap perSubjectObservers;07
08   protected List getObservers(Subject s) {
09     if (perSubjectObservers == null) {
10       perSubjectObservers = new WeakHashMap();
11     }
12     List observers =
13       (List)perSubjectObservers.get(s);
14     if ( observers == null ) {
15       observers = new LinkedList();
16       perSubjectObservers.put(s, observers);
17     }
18     return observers;
19   }
20
21   public void addObserver(Subject s,Observer o){
22     getObservers(s).add(o);
23   }
24   public void removeObserver(Subject s,Observer o){
25     getObservers(s).remove(o);
26   }
27
28   abstract protected pointcut
29     subjectChange(Subject s);
30
31   abstract protected void
32     updateObserver(Subject s, Observer o);
33
34   after(Subject s): subjectChange(s) {
35     Iterator iter = getObservers(s).iterator();
36     while ( iter.hasNext() ) {
37       updateObserver(s, ((Observer)iter.next()));
38     }
39   }
40 }
```

Figure 2: The generalized ObserverProtocol aspect

4.1.1.2 The Subject-Observer mapping

Implementation of the mapping in the AspectJ code is localized to the `ObserverProtocol` aspect. It is realized using a weak hash map of linked lists to store the Observers for each Subject (line 6). As each pattern instance is represented by a concrete subaspect[2] of `ObserverProtocol`, each instance will have its own mapping.

Changes to the Subject-Observer mappings can be realized via the public `addObserver` and `removeObserver` methods (line 21-26) that concrete subaspects inherit. To have a `Screen` object S become the Observer of a `Point` Subject P, clients call these methods on the appropriate subaspect (e.g. `ColorObserver`):

```
ColorObserving.aspectOf().addObserver(P, S);
```

The private `getObservers` method is only used internally. It creates the proper secondary data structures (linked lists) on demand (line 8-19). Note that in this implementation the Subject-Observer mapping data structure is centralized in each concrete extension. All concrete aspects that subclass the abstract pattern

[2] A subaspect is the concrete extension of an abstract aspect, the concept being similar to subclasses in OO languages

```
01 public aspect ColorObserver extends ObserverProtocol {
02
03   declare parents: Point  implements Subject;
04   declare parents: Line   implements Subject;
05   declare parents: Screen implements Observer;
06
07   protected pointcut subjectChange(Subject s):
08     (call(void Point.setColor(Color)) ||
09      call(void  Line.setColor(Color)) ) && target(s);
10
11   protected void updateObserver(Subject s,
12                                 Observer o) {
13     ((Screen)o).display("Color change.");
14   }
15 }
```

```
16 public aspect CoordinateObserver extends
17   ObserverProtocol {
18
19   declare parents: Point  implements Subject;
20   declare parents: Line   implements Subject;
21   declare parents: Screen implements Observer;
22
23   protected pointcut subjectChange(Subject s):
24      (call(void Point.setX(int))
25       || call(void Point.setY(int))
26       || call(void Line.setP1(Point))
27       || call(void Line.setP2(Point)) ) && target(s);
28
29   protected void updateObserver(Subject s,
30                                 Observer o) {
31     ((Screen)o).display("Coordinate change.");
32   }
33 }
```

Figure 3. Two different Observer instances.

aspect will automatically have an individual copy of the field. This follows the structure presented in [9]. This can cause a bottleneck in some situations. These can be fixed, on a per pattern-instance basis, by overriding `getObservers` with a method that uses a more decentralized data structure.

Generally, whenever a pattern solution requires a mapping between participants (i.e. the successor field of handlers in Chain Of Responsibility) and the pattern implementation is abstractable, we can either define a field on the participant, or keep the mapping in a central data structure in the abstract aspect (as in this example). Whichever approach is chosen, the point of access to the data structure is the instance-specific aspect, so that different instances of the pattern involving the same participants are possible and will not become confused.

4.1.1.3 The update logic

In the reusable aspect, the update logic implements the general concept that Subjects can change in ways that require all their observers to be updated. This implementation does not define exactly what constitutes a change, or how Observers should be updated. The general update logic consists of three parts:

The changes of interest depict *conceptual operations*, a set of points in program execution, at which a Subject should update its Observers (to notify them of changes to its state). In AspectJ, sets of such points are identified with pointcut constructs. In the reusable aspect, we only know there are modifications of interest, but we do not know what they are. Therefore, we define an abstract pointcut named `subjectChange` that is to be concretized by instance-specific subaspects (line 28-29).

In the reusable part we only know that the Observers will have to be updated in the context of the pattern, but cannot predict how that is best achieved. We define an abstract update method `updateObserver` that will be concretized for each pattern instance (line 31-32). That way, each instance of the Observer pattern can choose its own update mechanism.

Finally, the reusable aspect implements the update logic in terms of the generalizable implementation parts mentioned above. This logic is contained in the after advice (line 34-39). This after advice says: whenever execution reaches a join point matched by the `subjectChange` pointcut, update all Observers of the appropriate Subject afterwards.

4.1.2 Pattern-instance-specific concrete aspects

Each concrete subaspect of `ObserverProtocol` defines one particular kind of observing relationship, in other words a single pattern instance. Within that kind of relationship, there can be any number of Subjects, each with any number of Observers. The subaspect defines three things:

- The classes that play the roles of Subjects and Observers. This is done using the `declare parents` construct, which adds superclasses or super-interfaces to a class, to assign the roles defined in the abstract aspect.
- The conceptual operations on the subject that require updating the Observers. This is done by concretizing the `subjectChange` pointcut.
- How to update the observers. This is done by concretizing `updateObserver`. The choice between push or pull model for updates is no longer necessary as we have access to both the Subject and the Observer at this point and can customize the updates.

The `declare parents` construct is part of the AspectJ open class mechanism that allows aspects to modify existing classes without changing their code. This open class mechanism can attach fields, methods, or – as in this case – interfaces to existing classes.

Figure 3 shows two different instances of the Observer pattern involving the classes `Point`, `Line`, and `Screen`. In both instances, `Point` and `Line` play the role of Subject, and `Screen` plays the role of Observer. The first observes color changes, and the second observes coordinate changes.

Note that the type casts in line 13 and 31 are expected disappear with the planned AspectJ support for generics. It will then be possible to create parameterized subaspects that incorporate the role assignment and are type safe.

Particular classes can play one or both of the Subject and Observer roles, either in the same pattern instance or separate pattern instances. Figure 4 shows a third pattern instance in which `Screen` acts as Subject and Observer at the same time.

```
01 public aspect ScreenObserver
02               extends ObserverProtocol {
03
04  declare parents: Screen implements Subject;
05  declare parents: Screen implements Observer;
06
07  protected pointcut subjectChange(Subject s):
08    call(void Screen.display(String)) && target(s);
09
10  protected void updateObserver(
11    Subject s, Observer o) {
12      ((Screen)o).display("Screen updated.");
13  }
14 }
```

Figure 4. The same class can be Subject and Observer

In the AspectJ version all code pertaining to the relationship between Observers and Subjects is moved into an aspect, which changes the dependencies between the modules. Figure 5 shows the structure for this case.

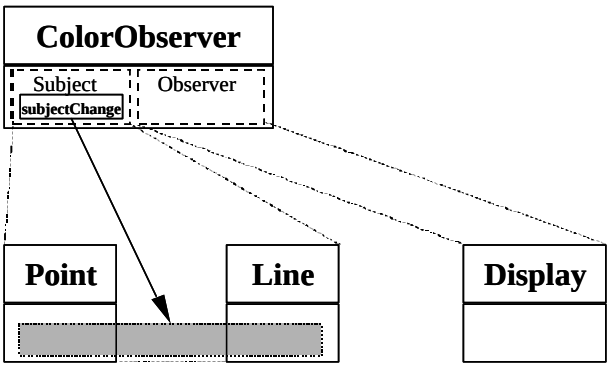

Figure 5: The structure of an instance of the Observer pattern in AspectJ. Subject and Observer roles crosscut classes, and the changes of interest (the subjectChange pointcut) crosscuts methods in various classes.

4.1.3 Properties of this implementation

This implementation of the Observer pattern has the following closely related modularity properties:

- *Locality* – All the code that implements the Observer pattern is in the abstract and concrete observer aspects, none of it is in the participant classes. The participant classes are entirely free of the pattern context, and as a consequence there is no coupling between the participants. Potential changes to each Observer pattern instance are confined to one place.
- *Reusability* – The core pattern code is abstracted and reusable. The implementation of ObserverProtocol is generalizing the overall pattern behavior. The abstract aspect can be reused and shared across multiple Observer pattern instances. For each pattern instance, we only need to define one concrete aspect.
- *Composition transparency* – Because a pattern participant's implementation is not coupled to the pattern, if a Subject or Observer takes part in multiple observing relationships their code does not become more complicated and the pattern instances are not confused. Each instance of the pattern can be reasoned about independently.
- *(Un)pluggability* – Because Subjects and Observers need not be aware of their role in any pattern instance, it is possible to switch between using a pattern and not using it in the system.

4.2 Other patterns

In the following we describe the remaining 22 GoF patterns and how the AspectJ implementation is different from a pure Java version. The patterns are grouped by common features, either of the pattern structures or their AspectJ implementations.

4.2.1 Composite, Command, Mediator, Chain of Responsibility: roles only used within pattern aspect

Similar to the Observer pattern, these patterns introduce roles that need no client-accessible interface and are only used within the pattern. In AspectJ such roles are realized with empty (protected) interfaces. The types they introduce are used within the pattern protocol. One abstract aspect for each pattern defines the roles and attaches default implementations where possible (see Figure 6 for parts of the abstract Composition aspect).

For patterns involving particular conceptual operations, the abstract pattern aspect introduces an abstract pointcut (to be concretized for each instance of the pattern), which captures the join points that should trigger important events (such as the execution of a Command in the Command pattern). As in the Observer example, advice (after, before, or around) is responsible for calling the appropriate methods.

In the Composite case, to allow walking the tree structure inherent to the patterns, we define facilities to have a visitor traverse and/or change the structure. These visitors are defined in the concrete aspect. See Figure 7 for an example of how statistics can be collected from the Composition structure. In this example we show an instance of the Composite pattern modeling a file system. Directories are Composites, and files are Leafs. The example shows how to calculate the disk space needed for the file system, assuming that `File` objects have a `size` field. Again, clients use a public method on the aspect to access the new functionality. Appropriate methods on the participants are introduced privately and are visible only by the aspect.[3]

4.2.2 Singleton, Prototype, Memento, Iterator, Flyweight: aspects as object factories

These patterns administrate access to specific object instances. All of them offer factory methods to clients and share a create-on-demand strategy. The patterns are abstracted (reusable) in AspectJ, with code for the factory in the aspect.

In the AspectJ implementations, the factory methods are either parameterized methods on the abstract aspect or methods attached to the participants. If the former approach is used, multiple instances of the pattern compose transparently, even if all factory methods have the same names. The Singleton case is special in that we can

[3] Due to a bug in AspectJ release 1.0.6 the private abstract introduction of `Component.sizeOnDisk()` does not work. This is scheduled to be fixed in the next release.

```
public abstract aspect CompositionProtocol {

  protected interface Component {}
  protected interface Composite extends Component {}
  protected interface Leaf    extends Component {}

  private WeakHashMap perComponentChildren =
    new WeakHashMap();

  private Vector getChildren(Component s) {
    Vector children;
    children = (Vector)perComponentChildren.get(s);
    if ( children == null ) {
      children = new Vector();
      perComponentChildren.put(s, children);
    }
    return children;
  }

  public void addChild(Composite composite,
                       Component component) {
    getChildren(composite).add(component);
  }
  public void removeChild(Composite composite,
                          Component component) {
    getChildren(composite).remove(component);
  }

  public Enumeration getAllChildren(Component c) {
    return getChildren(c).elements();
  }

  protected interface FunctionVisitor {
    public Object doIt(Component c);
  }

  protected static Enumeration
   recurseFunction(Component c,
                   FunctionVisitor fv) {
    Vector results = new Vector();
    for (Enumeration enum = getAllChildren(c);
      enum.hasMoreElements(); ) {
        Component child;
        child = (Component)enum.nextElement();
        results.add(fv.doIt(child));
    }
    return results.elements();
  }
}
```

Figure 6. Part of the abstract Composite pattern implementation

turn the original constructor into the factory method using `around` advice and returning the unique object on all constructor calls.

Parameterized factory methods can alternatively be implemented according to Nordberg's factory example [18]: the factory method is empty (returns `null` or a default object). Other return values are provided by around advice on that method. If the arguments are appropriate, the advice creates a new matching object; otherwise it just proceeds with the regular execution. This allows us extend the factory (in terms of new products) without changing its code. Participants no longer need to have pattern code in them; the otherwise close coupling between an original object and its representation or accessor (Memento, Iterator) is removed from the participants.

```
public aspect FileSystemComposite extends
              CompositeProtocol {

  declare parents: Directory implements Composite;
  declare parents: File      implements Leaf;

  public int sizeOnDisk(Component c) {
    return c.sizeOnDisk();
  }

  private abstract int Component.sizeOnDisk();

  private int Directory.sizeOnDisk() {
    int diskSize = 0;
    java.util.Enumeration enum;
    for (enum =
     SampleComposite.aspectOf().getAllChildren(this);
     enum.hasMoreElements(); ) {
       diskSize +=
        ((Component)enum.nextElement()).sizeOnDisk();
    }
    return diskSize;
  }

  private int File.sizeOnDisk()      {
    return size;
  }
}        }
```

Figure 7. Part of a Composition pattern instance aspect

4.2.3 Adapter, Decorator, Strategy, Visitor, Proxy: language constructs

Using AspectJ, the implementation of some patterns completely disappears, because AspectJ language constructs implement them directly. This applies to these patterns in varying degrees.

The Adapter and Visitor pattern can be realized by extending the interface of the Adaptee (via AspectJ's open class mechanism). Decorator, Strategy and Proxy have alternate implementations based on attaching advice (mentioned for Decorator in [18]).

While simpler and more modular, the approaches have inherent limitations. The advice-based implementation of Decorator loses its dynamic manipulation properties (dynamic reordering of Decorators) and is thus less flexible. The interface augmentation for Adapter cannot be realized in this manner when we want to *replace* an existing method with another one that has the same name and arguments but a different return type.

Protection or delegation proxies can be implemented to be reusable using the above approach, but some applications of the Proxy pattern require the Proxy and the Subject to be two distinct objects (such as remote and virtual proxy). In these cases the Java and AspectJ implementations are identical.

4.2.4 Abstract Factory, Factory Method, Template Method, Builder, Bridge: multiple inheritance

These patterns are structurally similar: Inheritance is used to distinguish different but related implementations. As this is already nicely realized in OO, these patterns could not be given more reusable implementations. However, with AspectJ it is possible to replace the abstract classes mentioned in the GoF solution by interfaces without losing the ability to attach (default) implementations to their methods. With Java, we cannot use interfaces if we want to define a default implementation for methods that are part of the pattern code. In that respect, AspectJ's

open class mechanism effectively provides a limited form of multiple inheritance.

Besides that, Builder and Bridge have the following additional implementation considerations. For Builder, an aspect can intercept calls to the creation methods and replace them with alternate implementations using `around` advice (see Strategy above). For Bridge, a decoupling of Abstraction and Implementor can be achieved by using polymorphic advice as suggested by Nordberg [24]. While this approach reduces the coupling between the participants, it is less flexible when it comes to dynamically changing Implementors.

4.2.5 State, Interpreter: scattered code modularized

These patterns introduce tight coupling between their participants. In the AspectJ implementations, parts of the scattered code can be modularized.

In the State pattern, the crosscutting code for state transitions can be modularized in an aspect using (mainly) `after` advice. For Interpreter, it is still possible to augment or change the behavior of the system without changing all participant classes. This can be accomplished by attaching methods to the participants using the open class mechanism.

4.2.6 Façade: no benefit from AspectJ implementation

For this pattern, the AspectJ approach is not structurally different from the Java implementation. Façade provides a unified interface to a set of interfaces to a subsystem, to make the subsystem easier to use. This example mainly requires namespace management and good coding style.

5. ANALYSIS

In this section, we present an analysis of the previously observed benefits of implementing patterns with AspectJ. The analysis is broken into three parts:

- The general improvements observed in many pattern re-implementations.
- The specific improvements associated with particular patterns.
- The origins of crosscutting structure in patterns, and a demonstration that observed improvements correlate with the presence of crosscutting structure in the pattern.

5.1 General Improvements

For a number of patterns, the AspectJ implementations manifest several closely related modularity benefits: locality, reusability, dependency inversion, transparent composability, and (un)pluggability. Attempting to say which of these is primary is difficult, instead we simply describe them and discuss some of their interrelationships.

The AspectJ implementations of 17 of the 23 GoF patterns were localized. For 12 of these, the locality enables a core part of the implementation to be abstracted into reusable code. In 14 of the 17 we observed transparent composability of pattern instances, so that multiple patterns can have shared participants (see Table 1).

The improvements in the AspectJ implementations are primarily due to inverting dependencies, so that pattern code depends on participants, not the other way around. This is directly related to locality – all dependencies between patterns and participants are localized in the pattern code.

An object or class that is oblivious of its role in a pattern can be used in different contexts (such as outside the pattern) without modifications or redundant code, thereby increasing the reusability of participants. If participants do not need to have pattern-specific code, they can be readily removed from or added to a particular pattern instance, making the participants (un)pluggable. To benefit from this, the participants must have a meaning outside the pattern implementation. For example, the participants in a Chain Of Responsibility pattern often have other responsibilities in the application they are in (as widgets in the GUI example in GoF), while Strategy objects usually just encapsulate an algorithm.

The locality also means that existing classes can be incorporated into a pattern instance without the need to adapt them; all the changes are made in the pattern instance. This makes the pattern implementations themselves relatively (un)pluggable.

Pattern locality should also allow a developer to easily impose global policies related to the design patterns, such as adding thread safety, logging facilities or performance optimizations.

In essence, we observe typical advantages generally associated with localized concerns with regards to future changes and program evolution. In particular, the problematic case of pattern composition/overlay [1, 7, 15, 21] becomes better structured (and easier to reason about) when pattern instances are defined in separate modular units.

In addition to code-level benefits, the modularity of the design pattern implementation also results in an inherent documentation benefit. As mentioned in [1, 21], the mere existence of classes that exclusively contain pattern code serve as records of what patterns are being used. In the AspectJ cases, we observe two additional improvements. First, all code related to a particular pattern instance is contained in a single module (which defines participants, assigns roles, etc.). This means that the entire description of a pattern instance is localized and does not "get lost" [21] or "degenerate" [7] in the system. Secondly, with the current AspectJ IDE support, all references, advised methods etc. are hyperlinks that allow a developer an overview of the assignment of roles and where the conceptual operations of interest are.

In 12 cases we were able to develop reusable pattern implementations. This happened by generalizing the roles, pattern code, communication protocols, and relevant conceptual operations in an abstract reusable aspect. For any concrete instance of the pattern, the developer defines the participants (assigns roles) and fills in instance-specific code. Changes to communication protocols or methods that are part of the abstract classes or interfaces involved do not require adjusting all participants.

If we can reuse generalized pattern code and localize the code for a particular pattern instance, multiple instances of the same pattern in one application are not easily confused (composition transparency). The same participating object or class can even assume different roles in different instances of the same pattern (see the Observer example above). This solves a common problem with having multiple instances of a design pattern in one application.

Table 1. Design pattern, roles, and desirable properties of their AspectJ implementations

	Modularity Properties				Kinds of Roles	
Pattern Name	Locality(**)	Reusability	Composition Transparency	(Un)pluggability	Defining(*)	Superimposed
Façade	Same implementation for Java and AspectJ				Façade	-
Abstract Factory	no	no	no	no	Factory, Product	-
Bridge	no	no	no	no	Abstraction, Implementor	-
Builder	no	no	no	no	Builder, (Director)	-
Factory Method	no	no	no	no	Product, Creator	-
Interpreter	no	no	n/a	no	Context, Expression	-
Template Method	(yes)	no	no	(yes)	(AbstractClass), (ConcreteClass)	(AbstractClass), (ConcreteClass)
Adapter	yes	no	yes	yes	Target, Adapter	Adaptee
State	(yes)	no	n/a	(yes)	State	Context
Decorator	yes	no	yes	yes	Component, Decorator	ConcreteComponent
Proxy	(yes)	no	(yes)	(yes)	(Proxy)	(Proxy)
Visitor	(yes)	yes	yes	(yes)	Visitor	Element
Command	(yes)	yes	yes	yes	Command	Commanding, Receiver
Composite	yes	yes	yes	(yes)	(Component)	(Composite, Leaf)
Iterator	yes	yes	yes	yes	(Iterator)	Aggregate
Flyweight	yes	yes	yes	yes	FlyweightFactory	Flyweight
Memento	yes	yes	yes	yes	Memento	Originator
Strategy	yes	yes	yes	yes	Strategy	Context
Mediator	yes	yes	yes	yes	-	(Mediator), Colleague
Chain of Responsibility	yes	yes	yes	yes	-	Handler
Prototype	yes	yes	(yes)	yes	-	Prototype
Singleton	yes	yes	n/a	yes	-	Singleton
Observer	yes	yes	yes	yes	-	Subject, Observer

(*) The distinctions between defining and superimposed roles for the different patterns were not always easy to make. In some cases, roles are clearly superimposed (e.g. the Subject role in Observer), or defining (e.g. State in the State pattern). If the distinction was not totally clear, the role names are shown in parentheses in either or both categories.

(**) Locality: "(yes)" means that the pattern is localized in terms of its superimposed roles but the implementation of the remaining defining role is still done using multiple classes (e.g. State classes for the State pattern). In general, (yes) for a desirable property means that some restrictions apply

5.2 Specific improvements

5.2.1 The Singleton case

The AspectJ version of the pattern implementation opened up two design options that are not possible in Java: First, is Singleton an inherited property, or do we have an inheritance anomaly? Second, do we want a devoted factory method to provide the Singleton instance, or do we want the constructor to return it whenever it is called?

We decided to implement the Singleton property as inherited, but provided facilities to exclude specific subclasses from the Singleton protection if desired.

For the second, we decided that using the constructor instead of a devoted factory method was beneficial. The factory, if desired, can then be implemented either directly in the class, or as a transparently composed aspect.

5.2.2 Multiple inheritance and Java

As originally presented, some of the GoF patterns make use of multiple-inheritance in their implementation, for example the class

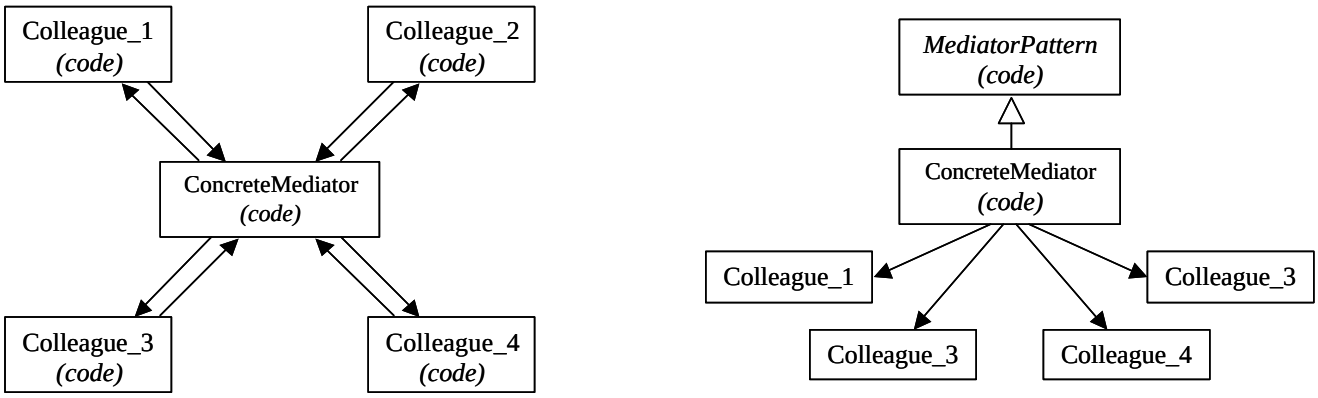

Figure 8: Dependencies and (pattern) code distribution in a typical instance of the Mediator pattern for Java (left) and AspectJ (right) implementations. The AspectJ implementation removes cyclic dependencies and localizes the pattern code.

version of the Adapter pattern. For many patterns, the roles that participants play within the patterns are realized as abstract classes in Java. Participant classes inherit interfaces and default implementations from these abstract classes. But if the participant classes have functionality outside the pattern context (such as GUI widgets as Subjects or Observers in the Observer pattern), they are usually already part of an inheritance hierarchy. Since Java lacks multiple inheritance, implementation in these cases can be somewhat awkward: In Java, if a participant has to inherit both its role and its other functionality, then one of the supertypes has to be realized as an interface. Unfortunately, interfaces in Java cannot contain code, making it impossible to attach default implementations of methods, for example.

The open class mechanism in AspectJ provides us with a more flexible way of implementing these patterns, as it allows to attach both interfaces and implementations (code) to existing classes.

5.2.3 Breaking cyclic dependencies

Some design patterns regulate complex interactions between sets of objects. In object-oriented implementations these classes are tightly coupled and mutually dependent. One example of a design pattern that introduces cyclic dependencies is Mediator, a variation of the Observer pattern that is often used in UI programming. Here, changes to Colleagues (e.g. widgets) trigger updates in the Mediator object (e.g. director). The Mediator, on the other hand, might update some or all of the Colleagues as a reaction to this.

A typical structure for this pattern is shown in Figure 8 (left). Inheritance relationships (the Mediator and Colleague interface) are not shown. The pattern introduces cyclic dependencies between Mediator and Colleagues (denoted by arrows pointing in opposite direction). The pattern code (for updates etc.) is distributed both over Mediator and all Colleagues.

In the AspectJ implementation (Figure 8, right), the indirection introduced by the ConcreteMediator aspect removes the cyclic dependencies. The aspect defines the participants, assigns the roles and identifies which points in the execution trigger updates. Colleagues do not have to have any pattern-related code in them, they are "freed" of the pattern. Changes to the pattern (for example, the notification interface) are limited to a single module (the aspect). Again, an abstract aspect (here: MediatorProtocol) implements generalizable parts of the pattern.

5.3 Crosscutting structure of design patterns

This section presents the origins of crosscutting structure in the patterns and shows that the observed benefits of using AspectJ in pattern implementation correlate with crosscutting in the pattern.

Roles define the behavior and functionality of participants in a pattern. Examples of such roles are Component, Leaf and Composite for the Composite pattern, Subject and Observer for the Observer pattern, or Abstract- and ConcreteFactory for the Abstract Factory pattern. Crosscutting in pattern structure is caused by different kinds of roles and their interaction with participant classes.

In some patterns, the roles are *defining:* the participants have no functionality outside the pattern. That is, the roles define the participants completely. Objects that play the Façade role, for example, provide a unified interface to a subsystem and (usually) have no other behavior of their own. Defining roles often include a client-accessible interface.

In other patterns, the roles are *superimposed:* they are assigned to classes that have functionality and responsibility outside the pattern. In the Observer pattern for example, the classes that play Subject and Observer do more than just fulfilling the pattern requirements. In a GUI context, Subjects could be widgets, for example. In other words, classes that have behavior outside the Observer pattern context. The Subject role is thus only an augmentation of the already existing class. Superimposed roles usually do not have a client-accessible interface.

In object-oriented programming, defining roles are often realized by subclassing an abstract superclass to achieve different but related behaviors; superimposed roles are often interfaces that define behavior and responsibilities.[4]

[4] There is a misalignment in Java in that methods on a superimposed role may only be intended for use by the pattern, but they have to be defined on an interface, which require they be public.

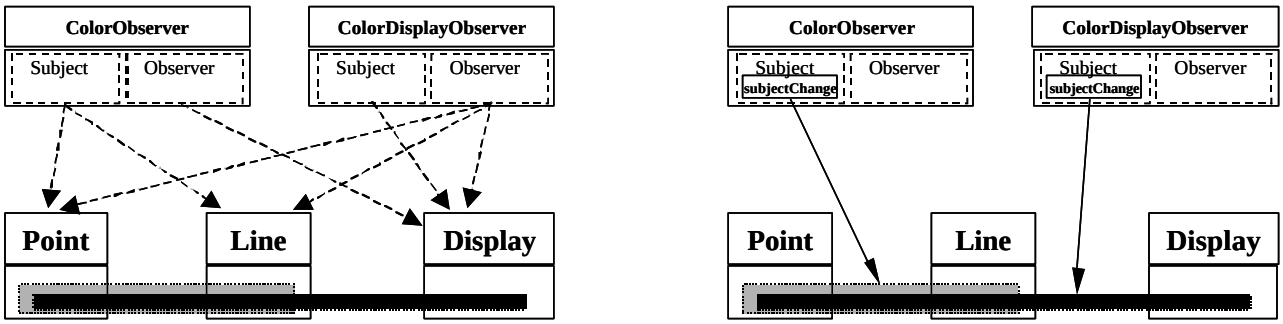

Figure 9: Crosscutting caused by pattern composition. In particular, this figure shows how pattern composition introduces additional crosscutting by extending Figure 5 with a second pattern instance. The left illustrates how a class can play multiple roles, while the right shows how mapping points in program execution onto the code crosscuts the participant's methods.

5.3.1 Roles and crosscutting

Superimposed roles lead to three different kinds of crosscutting among patterns and participants:

- Roles can crosscut participant classes. That is, for 1 role, there can be n classes, and 1 class can have n roles; i.e. the Subject role as shown in Figure 5.
- Conceptual operations of interest can crosscut methods in one or more classes. That is, for one conceptual operation there can be n methods, and 1 method can be in n conceptual operations; i.e. the `subjectChange` operations triggering an Observer update as shown in Figure 5.
- Roles from multiple patterns can crosscut each other with respect to classes and/or methods. That is, 2 classes that pattern A sees as part of 1 role, pattern B may see as in more than 1 role, and vice versa. The same is true for conceptual operations; i.e. Subject role and `subjectChange` operations as shown in Figure 9.

Table 1 shows that the types of roles a pattern introduces and the observed benefits of an AspectJ implementation correlate. The design patterns can be divided into three groups: those with only defining roles, those with both kinds of roles and those with only superimposed roles. The table shows that while the AspectJ implementations of the patterns in first group show no improvements, patterns from the last group show improvements in all modularity benefit categories we identified. For patterns that have both kinds of roles, the results are dependent on the particular pattern.

Given that AspectJ is intended to modularize crosscutting structure, this result should not be surprising. It says that patterns that involve primarily crosscutting structure are well modularized in an AspectJ implementation. (Note that AspectJ does not remove the crosscutting of the pattern, but rather provides mechanisms to modularize that structure.)

5.3.2 A predictive model?

The tight correlation between pattern roles, the crosscutting a pattern introduces, and the observed benefits of an AspectJ implementation suggest a predictive model of the benefit from AspectJ implementation of a given design pattern.

With defining roles, each unit of abstraction (class) represents a single concept, i.e. the functionality of a class corresponds to its role in the pattern. Inheritance is used to distinguish between related but different implementations. In such a case, transparency and pluggability are not useful properties, as each participant is inherently useful only within one particular pattern instance.

With superimposed behavior, the situation is different. Participants have their own responsibilities and justification outside the pattern context. If we force one such class into the pattern context, we have – at the very least – two concerns represented by one module of abstraction (class): The original functionality and the pattern-specific behavior. The resulting tangling and oftentimes code duplication can cause problems as the modularity is compromised. For these patterns and their implementations, a clean modularization of the pattern functionality and the original functionalities of the participants is desirable. In an AspectJ implementation it is usually possible to modularize the abstracted pattern behavior and have one aspect per pattern instance assign roles, conceptual operations, and fill in instance-specific code. Since the participants do have a meaning outside the pattern context, they are not inherently restricted to a single role or even a single pattern instance.

This model appears to be accurate for those GoF patterns that have only defining or only superimposed roles. For others, the expected benefits seem to depend on the number of participants implementing a particular kind of role. Superimposed roles that map to multiple participants (e.g. Element in Visitor, Composite or Leaf in Composite) indicate potential for modularization, even if the pattern also includes defining roles.

6. RELATED WORK

There is a lot of related work focusing either on patterns beyond the GoF patterns, or on issues beyond those in this paper. Note that since our work focuses on the implementation of existing design patterns, we do not mention publications dealing with finding new patterns. In particular, related work has been done to:

1. Investigate pattern applicability in other language paradigms
2. Automate code generation for patterns, to create a design patterns code library, or to develop tool support for program design with patterns

3. Classify existing patterns in order to reduce the number of distinct patterns or to pinpoint inherent relationships between them
4. Address the problem of design pattern composition
5. Enhance the representation of design patterns

6.1 Design patterns and language paradigms

Work in this area is directly related to this paper: We investigate design pattern implementations in AspectJ (AOP) and compare it to implementations in Java (OO).

Norvig's work on design patterns in dynamic programming [19] explores impacts on the GoF design patterns when implemented in Lisp and/or Dylan. This work is another indicator that patterns depend on the language paradigm. Of the 23 patterns, he found that 16 either become either invisible or simpler due to first-class types, first-class functions, macros, method combination, multimethods, or modules.

Sullivan investigated the impact of a dynamic, higher-order OO language (Scheme with a library of functions and macros to provide OO facilities) on design pattern implementations [22]. In-line with Norvig's work, he observed that some design pattern implementations disappear (if language constructs capture them), some stay virtually unchanged and some become simpler or have different focus.

Nordberg describes how AOP and component-based development can help in software module dependency management [17]. In a different work, he views design pattern improvements from the point of view of indirections and shows how replacing or augmenting OO indirection with AOP indications can lead to better designs [18].

Kühne showed the benefits of combining programming paradigms via design patterns [12]. In his work, he introduces design patterns to integrate high-level concepts from functional programming in OOP.

DemeterJ is an adaptive aspect-oriented extension to Java and another example of how new language constructs can make design patterns (as described in GoF) disappear. The Visitor design pattern is directly supported in DemeterJ [26].

A few aspect-oriented design patterns have been suggested. For example, Lorenz's work describing Visitor Beans, an AOP pattern using JavaBeans [14], or AOP versions of particular design patterns as the Command pattern [20].

6.2 Pattern libraries, parameterized patterns, and tool support

Since design pattern descriptions contain information about how the participants interact with each other, what interfaces and variables they have to have, it is only natural to investigate how much of the design and code generation process can be automated. In many cases, the design patterns "essence" can be encapsulated in an abstract aspect and reused. These aspects can be thought of as a library for patterns, or as library of building blocks for systems using design patterns.

Budinsky et al. [4] propose a tool for automated code generations from design pattern descriptions. Their tool integrates pattern code into existing systems using multiple inheritance. An interesting property of their tool is that it allows for different versions of each design pattern, according to the pattern descriptions in GoF. Such design choices are dynamically reflected in updated UML diagrams and changed code, so that developers can see the effects of their choices.

In a paper by Florijn et al. [7] a different tool is presented that uses a pattern representation based on so-called fragments (see section 6.5) that allows detecting whether a pattern does not conform to a particular design pattern "contract" and that can suggest improvements.

A paper by Mapelsden et. al. [15] shows a CASE tool that uses their design pattern modeling language DPML (see section 6.5). The tool provides an explicit separation between design patterns, their instances, and object models, which a user study found effective in managing the use of design patterns.

Alexandrescu's [2] generic components offer a different approach to make design pattern more flexible and reusable. These components are reusable C++ templates that are used to create new pattern implementations with little recoding. In [21], Soukup describes a C++ library of reusable pattern implementations, which uses an approach quite similar to ours. To avoid invasive changes to existing classes, "pattern classes" are introduced, which are encapsulations of the pattern role implementations. These classes include pattern code and a description of the pattern and participants in a parameterized form describing the roles and which code to inject where. Concrete instance of a pattern are created using these descriptions and a special code generator. In our work, the functionality of the pattern classes are replaced by abstract aspects that encapsulate the roles and pattern behaviors. Instead of weaving a role-class mapping and the description to create code, a concrete aspect is used to assign the roles and to fit in appropriate code.

6.3 Pattern Classification

Based on our comparison, we classify design patterns according to their usage of roles, as this is what we found to affect their potential to benefit from an aspect-oriented implementation.

Various works have addressed the growing number of design patterns and tried to classify existing patterns according to various characteristics. Agerbo [1] distinguishes between fundamental design patterns (FDPs), and language-dependent design patterns (LDDPs). While FDPs are not covered by any language construct (in any language), LDDDs have different implementations (or disappear completely) depending on the language used.

Gil [10] proposes a similar classification based on the closeness of patterns to actual language constructs. He identifies three different types of patterns: clichés, idioms, and cadet patterns. Clichés are "common uses of prevalent mechanisms" of a particular programming language, idioms are language mechanisms of non-mainstream languages, and cadet patterns are "abstraction mechanisms not yet incorporated in any programming language". We used the reasoning that Façade is more a generally accepted mechanism for information hiding (a Cliché in Gil's terminology) than a fully-fledged pattern to explain why it does not profit from an AspectJ implementation.

Zimmer [23] investigated the relationship between patterns in pattern compositions. He introduces a three-layer classification of the GoF design pattern based on their potential role in pattern

compositions. The different categories are "basic design patterns and techniques" for rudimentary patterns used in others; "design patterns for typical software problems" for higher-level patterns for more specific problems. Finally, "design patterns specific to an application domain" is for domain specific patterns. Compared to our work it appears that patterns that use other patterns in their solution (i.e. are higher up in the hierarchy) should introduce more crosscutting than others and profit more form an AspectJ implementation. It turns out, however, that the usage of roles is much more relevant for determining how crosscutting a pattern is.

6.4 Roles and pattern composition

Pattern composition has been shown as a challenge to applying design patterns. In our work, we show how coding design patterns as aspects can solve the modularity problems associated with pattern composition.

The Role Object Pattern [3] has been introduced to deal with different requirements imposed on objects in different contexts. This approach is an OO attempt to deal with superimposed roles[5]. The separation of core functionality and role is realized by introducing role object fields into the core classes, which themselves share a high-level interface with the role classes. This creates cyclic references: `ComponentCore` stores a list of roles, and each `ComponentRole` has a reference to the core object they are attached to. While introducing tight coupling between core and role, this approach enables dynamically adding and removing roles from an object. Fowler [8] presents guidelines on different variations of the pattern and when to use them.

Other work describes different approaches to model roles and their relationship to the concrete classes playing those roles. Mikkonen [16] formalizes them as behavioral layers (object slices). Florijn et. al. [7] introduces a fragment model (see below) that represents participant roles as a particular kind of fragments. Mapelsden et. al [15] differentiate explicitly between patterns, their instances, and object models. Their graphical notation (DMPL) allows mapping roles to concrete classes. Design pattern libraries and code generators usually introduce a means to assign pattern roles to concrete classes. The most commonly used tools to weave role-related code into existing classes are multiple inheritance [1, 4, 16], or a dedicated weaver [21].

6.5 Alternative pattern representations

This area is remotely related in that it outlines new approaches to design pattern notation.

A number of papers address problems with the preciseness of the pattern description format presented in GoF. Lauder and Kent [13] introduce a hierarchical model (consisting of three layers based on UML notations) for describing pattern structures and dynamic behavior. The role model captures the "pure pattern", and is refined by a type-model (similar to the GoF UML diagrams), which is in turn refined by an instance-specific model that uses the concrete names a particular pattern instance. The authors claim that the three models complement each other and that a developer should have access to all three models of a particular pattern.

Florijn et. al. [7] suggest a fragment-based representation of design patterns. A fragment depicts a design element such as a class, method or association). Patterns themselves and all elements in a pattern instance (classes, relationships among them, code) are represented as (graphs of) fragments.

Mapelsden et. al. [15] introduce the design pattern modeling language DPML, built upon similar concepts as UML. This multi-level approach (design patterns, pattern instances, and object models) makes it possible to show objects and their roles within the pattern.

Mikkonen [16] addresses the problem that the temporal behavior of design patterns is difficult to reason about and proposes a formal notation for this purpose. This model formalizes patterns as behavioral layers, and realizes the interactions between objects as atomic actions. With this approach, pattern compositions can be modeled.

7. SUMMARY

Improvement from using AspectJ in pattern implementations is directly correlated to the presence of crosscutting structure in the patterns. This crosscutting structure arises in patterns that superimpose behavior on their participants. In such patterns the roles can crosscut participant classes, and conceptual operations can crosscut methods (and constructors). Multiple such patterns can also crosscut each other with respect to shared participants.

The improvements manifest themselves as a set of properties related to modularity. The pattern implementations are more localized, and in a number of cases are reusable. Because the AspectJ solutions better align dependencies in the code with dependencies in the solution structure, AspectJ implementations of the patterns are sometimes also composable.

Localizing pattern implementation provides inherent code comprehensibility benefits – the existence of a single named unit of pattern code makes the presence and structure of the pattern more explicit. In addition, it provides an anchor for improved documentation of the code.

Our results suggest several directions for further experimentation, including applying AspectJ to more patterns, attempting to make systematic use of our reusable pattern implementations, and attempting to use AspectJ in legacy code bases that are known to be influenced by design pattern thinking. Another avenue for future work is to compare these results with the use of other aspect-oriented techniques.

8. ACKNOWLEDGEMENTS

Our thanks go to Gail Murphy and the anonymous reviewers for their helpful comments on earlier versions of this paper.

9. REFERENCES

[1] Agerbo, E., Cornils, A. How to preserve the benefits of Design Patterns. Proceedings of OOPSLA 1998, pp. 134-143

[2] Alexandrescu, A. Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley, 2001

[3] Bäumer, D., Riehle, D., Siberski, W., and Wulf, M. Role Object Pattern. Proceedings of PLoP '97. Technical Report

[5] In that the core classes already have defined responsibility and the role introduces additional responsibilities.

WUCS-97-34. Washington University Dept. of Computer Science, 1997

[4] Budinsky, F., Finnie, M., Yu, P., Vlissides, J. Automatic code generation from Design Patterns. IBM Systems Journal 35(2): 151-171

[5] Coplien, J. O. Idioms and Patterns as Architectural Literature. IEEE Software Special Issue on Objects, Patterns, and Architectures, January 1997

[6] Coplien, J. O. Software Design Patterns: Common Questions and Answers. In: Rising L., (Ed.), The Patterns Handbook: Techniques, Strategies, and Applications. Cambridge University Press, NY, January 1998, pp. 311-320

[7] Florijn, G., Meijers, M., Winsen, P. van. Tool support for object-oriented patterns. Proceedings of ECOOP 1997

[8] Fowler M.: Dealing with roles. Proceedings of PLoP '97. Technical Report WUCS-97-34. Washington University Dept. of Computer Science, 1997

[9] Gamma, E. et al. Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994

[10] Gil, J., Lorenz, D. H. Design Patterns vs. Language Design. ECOOP 1997 Workshop paper

[11] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwing, J. Aspect-Oriented Programming. Proceedings of ECOOP '97, Springer Verlag, pages 220-242, 1997

[12] Kühne, T. A Functional Pattern System for Object-Oriented Design. Ph.D. Thesis, Darmstadt University of Technology, Verlag Dr. Kovac, ISBN 3-86064-770-9, July 1999

[13] Lauder, A., Kent, S. Precise Visual Specification of Design Patterns. Proceedings of ECOOP 1998

[14] Lorenz, David H. Visitor Beans: An Aspect-Oriented Pattern. ECOOP 1998 Workshops, pages 431-432, 1998

[15] Mapelsden, D., Hosking, J. and Grundy, J. Design Pattern Modelling and Instantiation using DPML. In Proceeding of TOOLS Pacific 2002, Sydney, Australia. Conferences in Research and Practice in Information Technology, 10. Noble, J. and Potter, J., Eds., ACS

[16] Mikkonen, T. Formalizing Design Patterns. Proceedings of ICSE 1998, pp. 115-124

[17] Nordberg, M. E. Aspect-Oriented Dependency Inversion. OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems, October 2001

[18] Nordberg, M. E. Aspect-Oriented Indirection – Beyond Object-Oriented Design Patterns. OOPSLA 2001 Workshop "Beyond Design: Patterns (mis)used", October 2001

[19] Norvig, P. Design Patterns in Dynamic Programming. In: Object World 96, Boston MA, May 1996

[20] Sletten, B. Beyond Actions – A Semantically Rich Command Pattern for the Java™ Foundation Classes (JFC/Swing) API. Presentation at JavaOne 2002

[21] Soukup, J. Implementing Patterns. In: Coplien J. O., Schmidt, D. C. (eds.) Pattern Languages of Program Design. Addison Wesley 1995, pp. 395-412

[22] Sullivan, G. T. Advanced Programming Language Features for Executable Design Patterns. Lab Memo, MIT Artificial Intelligence Laboratory, number AIM-2002-005, 2002

[23] Zimmer, W. Relationships Between Design Patterns. In: Coplien, J. O., Schmidt, D. C. (eds.) Pattern Languages of Program Design. Addison-Wesley, 1995, pp. 345-364

[24] The AspectJ user mailing list. http://aspectj.org/pipermail/users/

[25] The AspectJ web site. http://www.aspectj.org

[26] The DemeterJ web site. http://www.ccs.neu.edu/research/demeter/DemeterJava/

[27] The Java web site. http://www.java.sun.com