

# Patterns of Aspect-Oriented Design

James Noble<sup>1,2</sup>, Arno Schmidmeier<sup>3</sup>, David J. Pearce<sup>2</sup>, Andrew P. Black<sup>4</sup>

(1) Imperial College London

(2) Permanent: Victoria University of Wellington, {kjj,djp}@mcs.vuw.ac.nz

(3) AspectSoft, arno@aspectsoft.de

(4) Portland State University, black@cs.pdx.edu

June 11, 2007

## Abstract

Aspect-oriented programming languages are becoming commonplace, and programmers are accumulating experience in building and maintaining aspect-oriented systems. This paper addresses how the use of these languages affects program *design*: how aspect-oriented languages change the design space, which designs should be emulated and which avoided, and the strengths and weaknesses of particular kinds of design. We identify five patterns of aspect-oriented design: Spectator, Regulator, Patch, Extension, and Heterarchical Design. For each pattern, we describe the problem it solves, show how aspect-oriented language features are used in the pattern, give characteristic examples of the pattern's use, and assess its benefits and liabilities.

Our patterns provide the beginnings of a taxonomy of aspect-oriented design. We believe that they should help programmers to understand and evaluate existing aspect-oriented designs, to improve new designs, to make better use of the aspect-oriented features of new programming languages, and also guide those who wish to implement these patterns in non aspect-oriented languages.

*Dear Workshop: this paper already has an interesting history, and we plan to do more with it after the workshop. For EuroPloP, we have expanded the patterns part of the paper, indeed, the first two patterns were one in the prior version. It still remains to significantly increase the depth of description and analysis in each pattern. We expect to remove much of the discussion material (into a different paper), but are including it here so you can have some idea where we're coming from. **Authors** ▶ We've also included our comments so you can see some of our questions: this really is a work in progress. ◀ Probably the most mature part of the paper is the first 3-4 patterns, so we suggest you focus on those — we look forward to your comments!*

## 1 Introduction

Aspect-orientation as a research area is at least ten years old — with AspectJ [39] first being described in the late 1990s, and key precursors such as subject-oriented design [11], Composition Filters [1, 2], and Adaptive Programming [45] emerging a good five years earlier. Aspect-oriented programming languages or systems, especially AspectJ and web architectures [33, 32], have been used in production environments for several years, and a range of new programming languages [47, 49, 50], and design and analysis methods [9, 10, 31] are emerging from the research community.

---

<sup>0</sup>Copyright © 2006-2007 by James Noble, Arno Schmidmeier, David J. Pearce, and Andrew P. Black. Permission is granted to Hillside Europe and EuroPloP 2007 participants to copy, reproduce, republish, and post on servers for the sole purpose of workshopping this paper at EuroPloP 2007.

This paper begins to characterise aspect-orientation according to the program *designs* that it enables, supports and encourages. Thus, when we speak of design, we are referring to the actual structure of the program, rather than the process used to produce that structure. We are less interested in particular language constructs than in the *shape* of the programs that aspect-oriented languages (in a broad sense) encourage.

We are also interested in programmer intent and motivation, in the rationale for particular designs, and in their benefits and liabilities. In contrast to other taxonomies or critiques of AOP (which have tended to focus on aspect-orientation as a whole [60, 9], the advocacy and analysis of various language features [22, 58, 13], or on case studies [36, 16, 35]), in this paper we pose and attempt to answer questions such as: How should we think about aspect-oriented design? What are programmers' intentions in producing AO designs? How can we evaluate aspect-oriented designs? When are particular designs appropriate, and what are their strengths and weaknesses? What examples of AO design are worth emulating? What should we teach programmers when educating them about aspect-oriented design?

The key contribution of this paper is that we describe five *patterns* of aspect-oriented design: **Spectator**, **Regulator**, **Patch**, **Extension**, and **Heterarchical Design**. We adopted a patterns approach for two reasons: first, because it encouraged us to engage with the specifics of particular designs, and second, because it became evident that aspect-oriented design was too diverse to be analysed as if it were a single style. Each of these patterns provides a solution to a different problem, resolving a range of design issues; and, each has been used in a number of exemplar systems. As with the Gamma et al. *Design Patterns* [24], our patterns are not mutually exclusive: several patterns can be used in an aspect-oriented program, and a particular pattern may appear several times. Also like the Gamma patterns, we do not propose an all-encompassing narrative or methodology of aspect-oriented design other than the patterns themselves. We do not claim that these five patterns capture all (or even most) aspect-oriented designs; but, almost every aspect-oriented program that we have seen has used one or more of these patterns. One way in which these patterns differ from most other object-oriented design patterns, which describe particular structures of objects, is that these patterns are applicable at many scales, from individual objects, via classes or aspects, up to whole systems.

We base our terminology and programming language examples in AspectJ simply because it is the most well-known aspect-oriented language. The patterns we propose can be implemented in many other aspect-oriented languages; indeed, for each pattern we give at least one example involving a *non*-aspect-oriented language. We are more interested in the key features of the designs themselves than in the language used to express those designs. By extension, we believe that the ability to express most or all of these designs cleanly and succinctly can in some way be regarded as an ostensive definition of an aspect-oriented language.

The next five sections of this paper present each of these patterns in turn. We have adopted a somewhat abbreviated pattern form: each pattern starts with an *italicised* problem statement. This is followed by a paragraph discussing the context of the problem the pattern solves; that is, the important design-level issues (often called the forces) that the pattern must resolve. The context is further illustrated by a motivating example. We then present the solution proposed by the pattern, and illustrate it by solving the motivating example. The consequences of applying the pattern are then discussed; we also describe a number of known uses ("killer examples") of this pattern in successful, long-lived systems. Finally, each pattern ends with a discussion. The concluding sections of the paper compare and contrast the patterns at some length, and place them into the wider context of the analysis of and debate around AOP.

## 2 Spectator

*How can you monitor the execution of a program?*

**Also known as:** Harmless Advice, Introspection

### Context

Programmers often need to examine or verify the execution of their programs. An execution trace or program profile can be very useful when debugging almost any kind of program. **APB** ► *hates webShopper and BeanZ; would like full names in the code*◄

### Motivating Example

Consider the following scenario. The middle tier of an enterprise web-shopping application is built from a number of business objects, which we will call *Beanz*. For example, *OrderBeanz* represent orders from the shop, taking requests from a Java server page and eventually storing them in a database. To operate in the enterprise environment, the web application must log each action for audit; this should also help in debugging. Logging is not difficult to handle — calls to system logs can be coded into the start and end of each *Beanz* method:

**KJX** ► *Do we need a second spectator concern here?*◄ **APB** ► *I don't think so, in the sense that the problem and the AspectJ solution are clear without it. However, we do need a second resolution, e.g., using a wrapper. I would be inclined to do this in Smalltalk with a wrapper class whose messageNotUnderstood method will log the call, call the real Beanz class, and then log the return. This example could then be revisited when we talk about reflection, since reflection would be the natural way to implement it. This might be reason enough to introduce a second example, since this technique can't be used in Java.*◄

```
class OrderBeanz extends Beanz {
    private int orderNumber;
    ...
    int orderNumber() {
        beanzLogger.log(M.ENT, "orderNumber");
        int rv = this.orderNumber;
        beanzLogger.log(M.RET, "orderNumber", rv);
        return rv;
    }
}
```

Such code has several problems. First, it has low cohesion: the code that handles logging is tangled with the code that handles business responsibilities, and the logging code takes up far more of the method body than the business logic, which (in this example) is just a variable access. Secondly, the code is tightly coupled with the infrastructure service that provides the systemic behaviour: changing the logging service would require that every method using it be rewritten. Third, all the code that calls into the infrastructure must be duplicated within every method on every *Beanz* class: every method needs to have the same structure of calls to the logger. Moreover, that code is stereotyped since the entry and exit code (and any necessary exception handling code) is found at the same point in every method. Making such duplicated, stereotypical code correct and consistent is a problem that should be automated; it is too easy for a programmer to miss one or two cases when inserting the code by hand, particularly during

maintenance. It seems clear that the main program logic (the business domain code) should be separated from the stereotypical *spectator* code.

**Therefore:** *Use an aspect to introspect on the behaviour of the program*

In AspectJ, and many other aspect-oriented systems, we can move this kind of concern into a separate aspect, which we call a **spectator** aspect. The resulting *Beanz* class then contains only the domain code; at system configuration time or run time (depending on the particular aspect system) we can combine the domain code—the so-called *base program*—with whichever aspects are required. Generally, spectator aspects use *homogeneous* pointcuts, which treat all method invocations on domain objects uniformly [16]. They do not normally define new fields or methods on the domain objects, that is, they do not use so-called “intertype declarations”, also known as “introductions”.

### Example Resolved

In the example, the base class code reduces to the following:

```
class OrderBeanz extends Beanz {
    private int orderNumber;
    ...
    int orderNumber() {return this.orderNumber;}
}
```

The logging concern is then implemented separately as an aspect:

```
aspect BeanzLogging {
    before(): execution(* Beanz+.*(..)) {
        logger.log(M.ENT, thisJoinPointStaticPart.getSignature().getName());
    }

    after(): execution(* Beanz+.*(..)) {
        logger.log(M.RET, thisJoinPointStaticPart.getSignature().getName());
    }
}
```

Note that the pointcut of this aspect uses the wildcard `* Beanz+.*(..)` to identify the execution of any method defined on class *Beanz* or on any of its subclasses, with any return type and any parameter list. **APB** ► *Is this right? I'm guessing, because I'm on an aeroplane and have no reference materials here.* ◀ **DJP** ► *Yes, that's right!* ◀ Thus, it is called a homogeneous pointcut because it treats all method executions uniformly.

### Consequences

The key benefit of this pattern is that cohesion is restored: the domain class now refers only to domain concerns, while each infrastructure concern is captured in an individual aspect. The nature of the coupling has changed: whereas the plain Java version relied on design rules that required the business objects to make stereotyped calls to the logging infrastructure, the aspectualized version relies on a different set of design rules that *forbid* the business objects from invoking the infrastructure services directly. This is important because such direct calls could interfere with any invariants maintained by the aspects [46].

**APB** ► *Check that this reference actually says what putting the citation here implies.* ◀

Apart from the observance of such design rules, the base code is uncoupled from (that is, it is *oblivious to* [22]) the various infrastructure services such as the logger. This is good because programmers can change the interface to these services by changing the aspects, without having to change the base code. However, the aspect code is closely coupled to the base code: depending on exactly how the pointcut is defined, changes to the names of the business object classes, or their subclassing relationships, or their result types, or their arguments, might disrupt the logging service in ways that will seem strange or mysterious to a programmer who is actually oblivious to the presence of logging.

A major benefit of this pattern is that code duplication has been eliminated: the method-level quantification provided by the homogeneous pointcuts (the ability to refer to all domain methods satisfying some condition) means that the advice in the aspect is effectively parameterised by the base method it advises. This means that a single piece of aspect code can replace many parametrically different calls in the original version. A significant feature of an AO programming language is that it provides a means to access these implicit parameters; in the example, AspectJ does this with `thisJoinPointStaticPart`, which provides access to the method being advised.

## Known Uses

The Spectator pattern captures one of the best-known applications of aspect-oriented techniques. Tracing Aspects, as discussed above, are clearly Spectators [44]; they are also the first examples in most AspectJ tutorials, and in tutorials for most other aspect-oriented systems.

Spectator aspects can be implemented using a range of different language constructs. Pointcut-based languages such as AspectJ offer direct support for them, while others provide it indirectly through special purpose preprocessors, virtual machine support (such as .Net's Interception interface) and various object-oriented design patterns (typically Proxy, also known as Interceptor, Encapsulator, or Wrapper). From our perspective, whether or not we are using a language with direct support for AOP is secondary to the question of whether we are implementing an aspect-oriented design. This is analogous to the idea that an object-oriented design can be implemented in a procedural language such as Pascal or C, or even in assembly language.

Spectators are indeed common in non-aspect-oriented languages. Many Lisp systems provide a tracing facility that is implemented by re-writing the source code of the function being traced. **APB** ► *I know this was true of LISP/370 around 1981, but hopefully someone will have a more current reference!* ◀ . **KJX** ► *no, that referece is cool!* ◀ Squeak Smalltalk implements services such as the ObjectTracer and the ObjectViewer using wrappers. The same technique can be applied to the problem of logging all operations on a Beanz. All that need be done is to wrap it in another object of class BeanzWrapper, which defines the following two methods.

```
doesNotUnderstand: aMessage
    "Do logging and forward message"
    ↑(tracedObject respondsTo: aMessage selector)
        ifTrue: [self pvtDoAround: aMessage]
        ifFalse: [super doesNotUnderstand: aMessage]

pvtDoAround: aMessage
    | result |
    logger logSendOf: aMessage.
    [↑result := aMessage sendTo: tracedObject]
        ensure: [logger logAnswerOf: aMessage as: result]
```

**APB** ► *This seemed like the right thing to do in Smalltalk; the corresponding Java would use a **try** ... **finally**, I believe. Do we want to put the corresponding Java in the AspectJ example?* ◀ **DJP** ► *Ok, I*

updated “after() returning” to just “after()” which catches all types of return, including via an exception.◀

Any messages (other than these two) sent to the wrapped object will not be understood, triggering the `doesNotUnderstand:` method. Provided that the wrapped object can indeed respond to the message in question, the message `pvtDoAround:` is sent to implement the “around advice”. (The `pvt` prefix designates a private method). `pvtDoAround:` delegates the actual logging actions to a logger object; this makes it possible to re-use the wrapper mechanism for other purposes.

If, as in the AspectJ example, the application requires that all messages on all instances of all subclasses of `Beanz` be logged, then all of those instances must be wrapped. This is easily accomplished by overriding the new method on `Beanz`:

```
new
  ↑BeanzWrapper wrap: super new
```

Alternatively, wrappers can be installed more selectively if that is what the application requires.

Finally, the DJProf profiler represents the archetypal example of the Spectator pattern using AspectJ [53]. This profiles various program metrics (e.g. heap usage and object lifetime) using aspects which maintain state characterising the current execution trace in various ways.

## Discussion

The Spectator pattern seems to capture one of the most straightforward and most common kinds of aspect-oriented design. The name “Spectator” comes from Clifton and Leavens’ analysis of meta-level facilities [13]; we discuss the relationship between meta-programming and aspects further in section 7.5. Katz defines “Spectative aspects” with respect to properties of a base program, rather than with respect to the program itself [34]. Thus, a particular piece of advice might be “spectative” with respect to one property, but not to another. An obvious example is a Spectator aspect that maintains an invariant on account balances, but nevertheless violates a security invariant because it prints “secret” information.

**APB** ▶ *We should check exactly what [13] and [58] claim*◀ Spectator aspects that do not change any part of the program state visible to the base program, and thus maintain whatever total correctness properties that have been established for the base, have also been called *augmentation* aspects [58].

### 3 Regulator

*How can you control the execution of a program?*

**Also known as:** Interception, Intercession

#### Context

Programmers often need to enforce invariants on the execution of their programs. Object invariants, which capture relationships between the instance variables on an object, and class invariants, which capture relationships that hold over all of the objects of a class, can be essential in any kind of object-oriented program. Particular kinds of program may require more specialised monitoring or additional behaviour while they are running. For example, enterprise applications often require that certain methods be executed only within a transaction, or when holding appropriate security privileges.

#### Motivating Example

Consider again the web-shopping application. To operate in the enterprise environment, the application must check the security of each method invoked on the `RemoveBeanz` class. Moreover, the database server requires that any database updates be performed under transaction control. As with tracing, neither of these concerns is difficult to handle individually: transaction entry and exit routines can be coded into the start of each `Beanz` method, as could calls to the system security manager. However, unlike the logging calls in Section 2, the transaction and security calls can change the control flow of the program, raising an exception when their conditions are not met:

```
class OrderBeanz extends Beanz {
    ...
    int orderNumber() {
        if (!beanzSecurityManager.check("orderNumber")) {
            throw new SecurityException(...);
        }
        Transaction txn = beanzTransactionManager.getTransaction(...);
        try {
            int rv = this.orderNumber();
            txn.commit();
            txn.close();
            return rv;
        } catch (TransactionException e) {
            txn.rollback();
            txn.close();
            throw e;
        }
    }
}
```

**APB** ▶ *I don't understand the transaction interface that we are assuming. It seems odd that the programmer has to perform explicit rollback when a commit fails — isn't this exactly what transactions are supposed to automate?* ◀ **DJP** ▶ *Well, EJB transactions require explicit rollback() calls like this, so I guess not.* ◀ **KJX** ▶ *I think the point is that application-level rollbacks can only be signalled by user code. If you're rolling back because of a low-level transaction conflict, you're right, but if it's because someone just hit a "cancel" button I don't see how to avoid this.* ◀

Such code has similar problems to the logging code in the Spectator pattern (Section 2). It has low cohesion, because the code handling infrastructure responsibilities is tangled with code handling business responsibilities. The lack of cohesion is even more painful here because of the verbosity of the exception handling machinery: security and transactions take up far more of the method body than the business logic. Once again, the stereotypical code that calls into the infrastructure must be duplicated within every method on every Beans class: every method needs to have the same structure. Moreover, the design rules that capture this structure—that tell the programmer where to place the infrastructure method calls to obtain correct application behaviour—have become much more complicated. Changes to the transaction or security policy may require that these rules be changed, and therefore that many methods be rewritten. The need to separate the main program logic (the business model domain code) from the infrastructure “boilerplate” is even more apparent.

**Therefore:** *Use an aspect to regulate the behaviour of the program*

As in the Spectator pattern, in AspectJ (and in many other aspect-oriented systems), we can move each of these concerns into a separate aspect, so that the Beans class contains only the domain code. The aspect-oriented programming system will then combine the domain code—the *base program*—with whichever aspects are applied.

Regulator aspects also generally use *homogeneous* pointcuts [16] that identify all method invocations on domain objects, and do not define new fields or methods (by means of intertype declarations or introductions). Regulator’s pointcuts and advice will typically be more complex than those of Spectator, because they may change or abort the execution of the base program.

### Example Resolved

In the example, the base code again reduces to the following:

```
class OrderBeans extends Beans {
    ...
    int orderNumber() { return this.orderNumber; }
}
```

while each of the regulatory concerns is implemented as an aspect. For security:

```
aspect BeansSecurity {
    before() : execution(* Beans+.*(..)) {
        if (!beansSecurityManager.check(
            thisJoinPointStaticPart.getSignature().getName()))
            throw new SecurityException(...);
    }
}
```

For transactions:

```
aspect BeansTransactionMonitor{
    Object around() : execution(* Beans+.*(..)) {
        Transaction txn = beansTxnManager.getTransaction(...);
        Object rv = null;
        try {
```

```

        rv = proceed();
        txn.commit();
        txn.close();
        return rv;
    } catch(TransactionException e) {
        txn.rollback();
        txn.close();
        throw e;
    }
}
}
}

```

**APB** ► *In the non-aspectual version, I changed the method signature to say that exceptions are raised (or else it won't compile.). These aspects also change the method signature: how is this handled in AspectJ? Doesn't it make a nonsense of "obliviousness"?* ◀ **DJP** ► *Ok, well I undid your change to the method signature. Strictly speaking, it will only fail to compile if TransactionException is a checked exception (since only these need to be declared in the method signature). With AspectJ, I'm pretty sure you cannot change the signature of a method (e.g. to declare a new exception can be thrown). What you can do as a work-around is to throw an unchecked exception (which doesn't need to be declared).* ◀ **KJX** ► *AspectJ has "declare soft to let you turn checked into unchecked exceptions — but I don't know what the best practice idiom is.* ◀

Note that the pointcuts of these aspects use wildcards to catch all invocations of Beans methods (homogeneous pointcuts).

## Consequences

As with Spectator, the key benefit of this pattern is that cohesion is restored: the domain class now refers only to domain concerns, while each infrastructure concern is captured in an individual aspect. The nature of the coupling has also changed in the same way as with Spectator: the design rules have changed. However, the design rules in the plain Java version were quite complex, and the design rules in the AspectJ version are far simpler: perform no security checks or exception handling.

In AspectJ, the base code cannot be entirely uncoupled from (or, *oblivious to* [22]) the various infrastructure services (transaction manager, security manager, etc.), because the services may raise exceptions that the base code must declare. Thus, if the security aspect needs to raise a new exception, the base code may need to be changed. **APB** ► *Does AspectJ do something to deal with this problem?* ◀ **DJP** ► *No, but you're referring to the same problem as before, about altering signatures? If so, it doesn't really need to solve this.* ◀ However, programmers can modify the services, and the *circumstances* under which they raise exceptions, by changing the aspects alone.

The situation with respect to code duplication, and the importance of a mechanism for parameterising the aspect code by the method that it is advising, is the same as with Spectator.

## Known Uses

Regulator aspects are the second most well-known applications of Aspect-Oriented techniques, after Spectator aspects. For example, Composition filters [2] are designed to filter messages — i.e., to act as Regulators. Moreover, most of the uses of aspect-orientation in middleware systems, such as SpringAOP and JBOSS, are applications of the Regulator pattern, for example, transactions, database access, and security. Such middleware systems often employ some Spectator functions as well, such as for tracing, profiling and logging [32, 33]. **DJP** ► *Why put the last sentence here? Surely it belongs in know uses for Spectator?* ◀

## Discussion

Regulators can be implemented using a range of different language constructs — again like Spectators.

The key difference between Regulators and Spectators is whether or not they affect the execution of the program: Regulators may intervene to regulate or control the execution of the base program, while Spectators may not. Dantas and Walker have characterised “Harmless Advice” as advice that preserves the partial correctness of the base program [17]. In other words, the base program must perform the same computation with or without the Harmless Advice. However, if the program terminates abnormally, partial correctness says nothing at all about its computation, so harmless advice can abort the program to prevent it, for example, from violating a security condition. What Harmless Advice cannot do is raise an exception which could be caught by the mainline code, since this might allow the program to terminate normally and, hence, potentially violate a property that held in the absence of the advice.

The definition of Harmless Advice also assumes that the base code cannot make any assertions about the state visible to the aspect code. Thus, harmless aspects cannot write to an output stream that is also accessible to the base program, because this would make it possible for aspects to invalidate properties of the stream that might be the subject of assertions made in the base code. (Since this would rule out one of the most common uses of Spectators and Regulators, Dantas and Walker’s formalism treats output as a special case. In the operational semantics of their language, `print s` reduces to the empty statement.) Our Regulator pattern, because it allows aspects both to raise exceptions and to manipulate state that is visible to the mainline code, is more general than Harmless Advice.

Regulator aspects that preserve partial correctness have also been called *assistants*, *narrowing* or *regulative* aspects. [13, 58, 34]. The name “Regulator” is derived from Katz’s “regulative aspects” [34].

Extensive use of Regulators is likely to result in the design of the base-program evolving to move more and more behaviour into aspects. This requires that subsequent program evolution observe design rules that ensure that the base program does not itself interfere with the concerns delegated to the aspects. For example, if synchronisation is handled by a Regulator, the base program should not provide any synchronisation directly. The overall design of the program gradually moves from Regulators to Planned Extension, and ultimately to a Heterarchical design. Current advanced web frameworks seem to be somewhere along this continuum [33].

## 4 Patch

**KJX** ▶ *What do we want to call it?*◀ **APB** ▶ *We should use a consistent lexicon: Spectator and Regulator both name a person or thing that does the critical action. So this would be Ad-hoc Extender*◀  
**DJP** ▶ *Patcher?*◀ **KJX** ▶ *now going with Patch*◀ .

*How can you extend a program that you are unable to change directly?*

**Also known as:** patch aspect; repair-person pattern; ad-hoc extension; hack aspect

### Context

Software development is increasingly dependent on the re-use of existing programs: libraries, frameworks, product lines, etc. However, because re-use places new demands on old code, it is likely that programmers will need to modify, extend or replace features in the programs that are being reused.

### Motivating Example

The standard libraries of many languages include a String class with, for example, methods to make a string upper or lower case. But some applications may need to transform a String into “title case” in which the initial letter of each word is capitalised but the remainder of the word is in lowercase. For example, title case might be used to give a common “look” to the names of products sold from the web shop.

The first generation of object-oriented languages allow programmers to extend classes directly by simply adding methods to existing classes. This feature is known as open classes [15] or class extensions, and is supported by Smalltalk and Objective C. In contrast, Java’s classes are closed, although Clifton et al. have proposed opening them in MultiJava [15]. With these extensions, it is easy to add a new method `titleCase` to `String`:

```
package org.hacker.webShop;
...
public String String.titleCase() {
    for (i=0; i < length; ++i) {
        if ((i == 0) || Character.isSpace(val[i-1]))
            val[i] = Character.toUpperCase(val[i]);
        else
            val[i] = Character.toLowerCase(val[i]);
    }
}
```

Unfortunately, standard Java does not allow this; neither do Eiffel or C#<sup>1</sup>. So how can we deal with this problem in Java? We can edit the source code of the String class, provided that we are legally entitled to do so. Even then, this means that our program is using an incompatible version of the library. It also means that whenever the “standard” library is updated, the edits will need to be repeated on the locally modified version, and that the modified library must be shipped with all products that use it.

An alternative approach is to include the extension code in the client. Suppose that we create a utility class in our application:

<sup>1</sup>C# 3.0 does include a provision for “extension methods”, which at first sight appear to be class extensions. However, closer examination shows that they are not: extension methods are syntactic sugar for static procedure calls; they are not dispatched, and hence cannot be changed in subclasses.

```

package org.hacker.webShop;
...
class StringUtil {
    static String titleCase(String s) {
        char result[] = new char[s.length()];
        for (i=0; i < s.length(); ++i) {
            if ((i == 0) || Character.isSpace(s.getChar(i-1)))
                result[i] = Character.toUpperCase(s.getChar(i));
            else
                result[i] = Character.toLowerCase(s.getChar(i));
        }
        return new String(result);
    }
}

```

However, such code is inevitably in the “wrong” place: attached to the wrong class, unable to access encapsulated implementation details and so likely to be inefficient, and unavailable to other components of the system that may reuse the library. Furthermore, using client-side auxiliary methods is harder to read and to understand than invoking the operation directly on the objects concerned.

Note that subclassing, the standard object-oriented extension mechanism, doesn’t really work here. For the moment, let’s ignore the fact that `java.lang.String` is declared to be “final” and thus cannot be subclassed, and pretend that we could create a subclass `WebString`. The problem is that all existing code will still create `java.lang.String` instances, which will not respond to the `titleCase` method.

**Therefore:** *Use an aspect to modify the behaviour of an existing component*

In AspectJ (and, again, many other aspect-oriented languages), we can provide the extension code as an aspect within the client system. Because it is an aspect rather than a class, the extension code can modify or replace the base code, adding functionality or fixing bugs as required. Extension code can access encapsulated representations, so it can be implemented as efficiently and straightforwardly as would be possible if it had been part of the base class. Nevertheless, the aspect remains part of the client code: the library code is not modified. When the system is woven together, the aspect code is inserted into the library class, so is available wherever it is needed; and, if the library is upgraded, the extension code may still weave correctly (if the changes to the base do not interfere with the aspect).

Generally, patch aspects will use *introductions* (also known as intertype declarations) [16] to add behaviour into the base code, and *around* advice to modify existing code. Pointcuts will primarily be heterogeneous: that is, they will target a few pieces of base code that need to be modified or extended.

## Example Resolved

The `titleCase` method can be supplied as a standalone aspect:

```

privileged aspect TitleCase {
    void String.titleCase() {
        for (i = 0; i < length; ++i) {
            if ((i==0) || Character.isSpace(val[i-0]))
                val[i] = Character.toUpperCase(val[i]);
            else
                val[i] = Character.toLowerCase(val[i]);
        }
    }
}

```

```

    }
}

```

**DJP** ► *Hmmm, this aspect doesn't strictly work in AspectJ, since you cannot modify anything in java+. \*. If it was any other library though, it would work!* ◀ The body of the method is exactly the same as the changes that would be made to the String class if it could be directly modified. That is, the aspect code can access the internal representation of the string.

## Consequences

The key benefit of this pattern is that the base code can be extended without having to be physically modified: the aspect applies to the code, but remains separate from it.

An important liability of this pattern is that because there is no explicit, predefined interface between base code and extension — precisely because these extensions are ad-hoc — the pattern is brittle. Changes to the base code may break the aspect. Moreover, it is just as easy for the aspect to break the base code in return. In our example, the class extension in MultiJava and the patch in AspectJ both violate the invariant of the java.lang.String that instances are immutable. Because this invariant is implicit (there are no immutability declarations in Java), such violations cannot be caught by the compiler, and may well go unnoticed until they cause errors.

Another problem is that multiple patches affecting the same piece of base code will be very tightly coupled, and can easily adversely interfere with each other.

Furthermore, patch aspects reduce the cohesion of the whole system. The aspect appears to be cohesive with the base code, not the extension, so it is in some sense in the “wrong place” within the static structure of the program. Of course, that is the underlying problem — the right code was not included in the base program, so we have to put it somewhere else! This pattern also increases the coupling of the client to the base code, especially if the extension aspect (which is still part of the client) uses privileged access to the base code.

## Known Uses

Early Smalltalk systems were developed this way [54]: patches were packaged into “Goodies” that — like aspects — could redefine methods anywhere in the system. Current Smalltalk systems (both Visualworks and Squeak) support package systems that explicitly institutionalise this usage: a package consists of a collection of new classes, and a collection of extensions to existing classes. Finally, many systems are maintained by Larry Wall's *patch* program **APB** ► *cite* ◀ ; patch aspects are a structured, language supported alternative to patching [18].

## Discussion

The key difference between Patch and Regulator is that, in this pattern, the base code can explicitly see **APB** ► *I don't know what this means* ◀ **DJP** ► *Well, the “base code” is the code which uses titleCase, so it must be able to “see” it. In Spectator, the base code is oblivious to the aspect.* ◀ the introduced aspect (i.e. it must be able to use titleCase, since this was the objective) and the changed behaviour — but, unlike later patterns, without any pre-planning.

We have seen Patch aspects used in two different situations. The titleCase example is about missing functionality: the titleCase method would have been a reasonable one to include in the String class if anyone had thought to do so, and, once the need for it has become apparent, we could over time expect refactorings to move these extension aspects into the base code. In practice, this will happen only when it is easy for developers to change the base code. If the base code is a third-party library, even seeing

its source code may be impossible for legal reasons. Even for open-source base code, the time and cost for releases to incorporate the extensions, or to fork the project and support the fork, will typically be greater than simply writing a local extension aspect. Thus, in Java code that does not use AOP, we find a proliferation of extension libraries and static helper methods because Java library classes cannot be extended.

The other situation has to do with the interface between a modular extension and the existing system. Consider adding a regular expression package to the system, perhaps to support user-defined searches. The package will define several new classes to represent the different kinds of regular expression and to recognise them. However, it would also be convenient to add a conversion method `toRE` to `String`, which would translate a string such as “a+b” into an instance of an `REUnion` class. It would *never* be reasonable for such a method to migrate into the base `String` class: the method makes sense only in the context of the regular expression extensions. Indeed, the implementation of `toRE` would depend on the existence of the RE classes.

Compared with the other patterns in this paper, this pattern may seem to have only a peripheral connection with the philosophy of “separation of concerns” — in many cases all the *technical* concerns such as cohesion, coupling, understandability and efficiency mean that rather than patching we should just refactor the base code. *Non-technical* concerns also affect the structure of systems, however. Patch aspects are often used to organise systems according to non-technical concerns, be they legal, political, or sociological. These concerns cannot be ignored when developing large systems using two or more development teams and building on diverse third-party components that are used across multiple projects.

## 5 Extension

*How can you design a program so that it can be extended in multiple ways?*

**Also known as:** Extension Aspect, Extension Points, XPIs, Hotspots, Plug-in Architecture, Planned Extension

**KJX** ► *changed name from “Planned Extension” to just “Extension”* ◀

### Context

Many programs, systems, and software product lines have to be designed so that they can be extended later — even though the ways in which they will be extended cannot generally be predicted at design time. Perhaps the system is a framework that other programmers or systems will extend; perhaps resources for building the whole system are unavailable at the start of the project; perhaps only the core requirements are known at the outset, and while some additional features are expected, the details of those features are not known in advance.

### Motivating Example

Consider the design of an Integrated Development Environment (IDE) that provides syntax-dependent text highlighting. Obviously, the highlighting must depend on the programming language being edited — if the IDE is to be extensible, programmers must be able to add support for more programming languages.

In the earliest formulations of object-oriented design [48] systems were supposed to be extended by two mechanisms: inheritance and component composition. Using inheritance, for example, one could provide support for a new programming language (say, Visual Algol) by extending the part of the system that provided the highlighting — the `TextEditor` class.

```
class VisualAlgolEditor extends TextEditor {
    void drawDisplay() {
        // draw display using Algol highlighting
    }
}
```

The problems with this approach are well known. The main problem is that (re)writing the `drawDisplay` method will not be a simple task. Programmers will need to read and understand the full code of the `TextEditor` class (and probably of its subclasses) to identify where highlighting is implemented and then to work out how to override some parts of them to support Visual Algol. Moreover, even if a working `VisualAlgolEditor` class can be written, the rest of the system will still have to be modified somehow to be aware of the new class. For example, the IDE may have to instantiate different `TextEditor` subclasses when editing different programming languages, which implies that other parts of the IDE will need to be augmented.

Since the publication of *Design Patterns* [24], however, there has been another object-oriented design practice: use patterns to make it possible to reuse particular parts of a larger framework. Indeed, this is the aim of many of the patterns, as captured in the subtitle of the book: *Elements of Reusable Object-Oriented Software*. In the IDE example, when designing the `TextEditor` class we could use the Strategy pattern: we could define a `HighlightStrategy` interface, and have the editor delegate highlighting decisions to an object implementing that interface.

```
class TextEditor {
    TextBuffer buffer;          // text to be edited
    HighlightStrategy highlighter = new NullHighlighter;

    void setHighlighter(HighlightStrategy h) { highlighter = h; }

    void drawDisplay() {
        ...
        highlighter.highlightBuffer(buffer)
        ...
    }
}
```

The wider IDE framework will still need some way to configure the `TextEditor` class with the correct `Highlighter`. There are pattern-based solutions to this problem too, but increasingly complex ones. We might use the Abstract Factory pattern to combine all the extensions (say, for file name conventions, highlighting, editing, compiler interfacing, and debugging) required for a particular programming language. We might use the Prototype pattern so that we have pre-existing instances to query so that we can determine the appropriate configuration for a particular file that we wish to create.

In spite of this complexity, a pattern-based design is a great improvement on a naïve object-oriented design. The strategy pattern works best when separate programs in a product line each make one choice of configuration, as when a company sells separate editors for VisualPascal, Visual Fortran and VisualAlgol, and produces them by re-using the same `TextEditor` class with different strategies. The complexity increases, but is still manageable, if all the editors must be available together in one program, because the program must now also include the logic to instantiate and inject the correct strategy.

However, this approach has a number of problems. Most importantly, the patterns are implicit: there is nothing in the code that says “here we use the strategy pattern” or “here we use the abstract factory pattern”. This makes the code hard to read, even by programmers who have been educated about patterns, and a veritable puzzle for programmers who have not met patterns before. Programmers must build (or generate) extra mechanism to implement the structure of the pattern, for example, the `HighlightingStrategy` interface, the `AbstractHighlightingStrategy` and `NullHighlightingStrategy` classes, along with a more-or-less complex mechanism to instantiate the correct strategy object. In larger systems, especially where different configurations are required simultaneously, these issues become quite significant. Finally, extending these systems can still be a black art, because different patterns must be related: supporting a new language will require not only a new highlighting strategy, but also new file name conventions, new compiler and debugger interfaces, and so on. Of course, we can deploy further patterns (Façade? Builder?) to address these problems, but each pattern increases the complexity and amount of structural code in the system, and makes the actual logic of the IDE harder to follow.

**Therefore:** *Define extension points to extend the base component’s behaviour*

In aspect-oriented languages we can explicitly mark *extension points* in the base code, using features such as abstract aspects, delegates, or virtual classes, depending on the language. Then, to extend the base component, we write concrete *extension aspects* that bind to those extension points and provide state and behaviour to support the new concern. Generally, while Extension aspects will use introductions [16] to supply behaviour, or *around* advice to modify existing methods, they do so in a very stylised manner. As a result, pointcuts are primarily heterogeneous: they will target one or more of the defined extension points.

## Example Resolved

In our example, we can create an explicit extension point **APB** ► *there is nothing explicit about this extension point* ◀ for highlighting by separating out the highlighting behaviour into a method — for the default case (no highlighting) we simply write a null method.

```
class TextEditor {
    TextBuffer buffer; // text to be edited

    void drawDisplay() {
        ... highlight(); ...
    }

    void highlight() {};
}
```

**APB** ► *So far, this is the template method pattern. We should say this.* ◀

Note that compared with the strategy-pattern solution all we have to do is untangle the highlighting behaviour from the rest of the display behaviour. We do *not* have to establish a complex supporting structure involving a HighlightStrategy interface, NullHighlighter default implementation, and a range of subclasses for different subclassing algorithms. All that we must (somehow) do is separate highlighting from display drawing.

Then, we can reify the extension point by declaring an abstract aspect, including an extension point for highlighting. In fact, this aspect can declare a number of other extension points related to language-specific behaviour, such as recognising that a particular filename should be handled by this extension.

```
aspect LanguageExtension {
    public pointcut handleFile(String s): args(s) && (call(static bool Buffer.canOpen(s)));
    public pointcut highlight(Buffer b): target(b) && (call(void Buffer+.highlight(..)));
}
```

**APB** ► *I had to go back and look at the forces section (whcih is missing). one of the forces is: all of th extensions necessar for a particular language should be packaged together/ Your AJ soln adchives this (as would a Smalltalk soln with a number of class extensions* ◀

**KJX** ► *somehwere we need to talk about multiple extensions coexisting. (or not coexisting)* ◀

Finally to implement an extension, we can simply extend the abstract aspect, and provide advice on pointcuts as necessary to implement the extension.

```
aspect VisualAlgol extends LanguageExtension {

    // Wrap all calls matching handleFile pointcut to check whether
    // file extension for language supported by this LanguageExtension.
    bool around(String s) : handleFile(s) {
        return s.endsWith(".alg") ? true : return proceed(s);
    }

    // Wrap all calls matching highlight pointcut and intercept those
    // intended for this LanguageExtension
    void around(Buffer b) : highlight(b) {
        if(b.fileName().endsWith(".alg")) {
            // draw display using Algol highlighting
        } else {

```

```

        // Continue original call (including any remaining intercepts)
        proceed(b);
    }
}

```

**DJP** ► *Andrew, is above the kind of commentry you wanted??*◀

## Consequences

The core of this pattern is parameterising the base code to facilitate extensions. Compared with the ad-hoc patch aspect pattern which enables patches to be attached anywhere, planned extension points are defined explicitly within the base code we are extending, and extension aspects identify one or more extension points to which they belong. In other words, rather than language level joinpoints and pointcuts, each base level component defines abstract, domain-level joinpoints to which extensions bind using domain-level pointcuts.

**APB** ► *I see the distinction, AspectJ does not. Erik M told me that his link extension to C# does let the programmer declare explicit extension points.*◀

These abstract definitions are the key liability **APB** ► *benefit? surely*◀ **KJX** ► *liability from an AOP-is-obliviousness POV*◀ of the pattern: base code is not oblivious to the possibility of extension but must define (in some way) the extension points. (Of course, base code does remain oblivious to any particular extension or even the presence or absence or any extension). On the other hand, the definitions provide the key benefit of the pattern: by establishing an extension interface, the base code and extensions can evolve independently — the extensions (that is the aspects) are much more oblivious (i.e. much more loosely coupled) to base code than in any of the other patterns we present in this paper. The extension points provide an interface (or a contract) which simultaneously insulates base code from the aspects, and aspects from the base code, providing each side interacts only via the interface.

**APB** ► *I don't know if you're a saying the pointcut defns are part of the base, or part of the extension*◀ .

**KJX** ► *I'm saying something needs to be in the base. Dependign on which style you adopt, they may be pointcuts, template methods, or whatever. In AspectJ you can supply defns into those extension points via pointcut defns — but you could also do it by realising abstract pointcuts from the base without definition any of your own. XPIs (I think) has pointcuts in the base - perhaps we go that way and deal with the rest via consequences/discussion sections*◀

As with many aspect-oriented designs, this pattern increases the cohesion of the whole system. But this pattern also decreases the coupling of the system, as base code and aspect are coupled via the interface defined by the extension points, rather than directly to each other.

## Known Uses

Building systems with explicit extension points has a long history: some notable examples include Emacs' hooks [59], Smalltalk's dependencies **KJX** ► *bluebook*◀ , Eclipse's extension points **KJX** ► *cite!*◀ , and .Net's delegates. The J& [49] nested type language has been used to declare and combine multiple extensions to the Polyglot compiler. Many uses of design patterns in OO frameworks are to provide precisely this extension capability. The Extension Object pattern [23] describes how extensions can be reified in object-oriented design: this pattern shows how aspects can provide more straightforward extensions.

**APB** ► *Bershad (I think) used an event mechanism for extensible operating systems — I should find that reference.*◀

## Discussion

There are a range of techniques used (and proposed) to support Extension aspects. We can distinguish a continuum of implementation techniques: pointcut-based “implicit” definitions (such as extension interfaces [28], eXtension Point Interfaces (XPI)s [27], and pointcut declarations in open modules [3, 51]), and hook or delegate or observer pattern based implementations, where the base code explicitly<sup>2</sup> calls the extension point as in Emacs hooks, Eclipse extension points, or C# delegates. For example, in something approaching C# programmers would write:

```
class TextEditor {
    TextBuffer buffer; // text to be edited

    public delegate void Highlighter(Buffer); // delegate type declaration

    public event Highlighter highlight(); // event declaration

    void drawDisplay() {
        ... highlight(); // invokes event
    }
}
```

to declare an extension point (C# delegate type and an event of that type), and then to invoke that event from the mainline code.

AspectJ 5’s annotation-based pointcuts [5] are in some way a mid-point in the continuum. As with explicit extensions, programmers decorate base code to indicate where it may invoke extensions, but as with implicit extensions, no changes are made to executable code. In the annotation-based style, the parameterisation mechanism is metadata annotations: the annotations form part of the interface used by extension pointcuts. However they may be implemented, all forms of Extension aspect designs rely on a “two-way” contract between base code and extension, The base code will call the extension as described in the interface, and the extension code will not interact with the base code except as permitted via the interface [27, 28].

---

<sup>2</sup>We call these “explicit” calls because the base code contains explicit message sends to the extensions. This kind of messages sends are also known as “implicit invocation” [26] because the “target” of the calls, that is the methods and objects that will be run (and whether or not there are *any* methods attached to the extension point that will be run) are **not** explicit in the source code. **APB** ►... which is always the case in with OO programs◄ **KJX** ►no. because when you write *a.b()* or *a b: c* you see the receiver *a* and message *b*. But here they’ll all be indirect, will be resolved via dynamic data structures, and impossible to follow statically◄ **APB** ►What I’m trying to say is that OO already decouples the message send from the method execution; the connection is made via dynamic data structures, and is impossible (in general) to follow statically◄

## 6 Heterarchical Design

**APB** ► *since you're coining the term, why the extra "AL"?* ◀

**KJX** ► *I'm not coining the term, comes in from evil continental philosophy and semiotics — and also it should clearly parallel hierarchical design. Unless you say "hierarchic design".* ◀ **APB** ► *... which I do. Hierarchy is the noun, hierarchic is the adjective.* ◀

*How can you design a program as a network of interpenetrating concerns?*

**Also known as:** Subject-oriented design; role-based design; big ball of mud

### Context

Alan Peris has described the structure of the systems that we build as an “intricately interlocked software elephant” [8]. For some systems, especially smaller ones, the hierarchical structure of a Pascal-style functional decomposition, or an OO inheritance hierarchy suffices to capture its essential complexity. For other systems, especially large ones, there is no dominant decomposition (whether tyrannous or benign) [61] that can possibly do the system justice. This is because the problem space imposes a network of interpenetrating, interdependent concerns, none of which is clearly more important or more significant than all others across the whole domain of the program.

### Motivating Example

The MetroSim urban planning project simulates a range of different concerns in the urban environment: political structures (city, county and district boundaries), transport (car, busses, rail, public and private), housing, mail delivery, recycling and solid waste removal, water supply, electricity supply, sewerage, telecommunications, road and city infrastructure maintenance, and so on. Each of these concerns takes its own view of the city, which may or may not be related to other views. Similarly, each concern can be modelled with its own individual class hierarchy — which may or may not relate to the decomposition required for other concerns.

For example, in the political and legal structure of a city, the basic unit is a `Lot` — a plot of land with a legal designation. Lots are aggregated into wards, suburbs, localities and then eventually cities, metropolitan boroughs, or counties. Furthermore, for town planning reasons, each Lot may also contain a number of `Buildings`.

Mail delivery can be simulated in MetroSim, again using Lots. For Mail delivery, each Lot belongs to a delivery area, identified by a postcode — and these areas may have very little relationship to the political or utility or even transport networks. Refuse collection, again, forms another overlapping system, as do roads, public transport, and so on.

The key problem with this approach is that in an object-oriented design (indeed in most program designs) the resulting program is a monolith: all these crosscutting *domain* concerns are tangled together [55]. The resulting design will be hard to understand, to modify, and to use, particularly because most programmers will not need to understand the entire design. Rather, programmers will want to work on small concerns — such as mail delivery or refuse collection — rather than the system as a whole.

**APB** ► *Your assumption here is that, just because one object, say a LOT, is part of several structures (mail, refuse, etc) that the cone will be tangled. You've done nothing to convince me of this, and I don't believe it* ◀

**KJX** ► *But what about all the stuff in the Lot class — see below?* ◀ .

Classical, object-oriented design patterns once again are no help here: or rather, they are crucial to a *functioning* OO design but do not address its architectural confusion. Every concern can use many patterns: Composite to model hierarchical structure; Strategy to allow choice of algorithms; Observer to handle updates to shared state, and so on; but these patterns will be implicit in the code, will themselves be tangled across concerns, and also lead to concerns further tangling together. To see how this plays out, consider part of the interface of the *Lot* class:

```
class Lot {
    // intrinsic state
    String designation;

    // political hierarchy — Composite pattern
    Section locality; // upwards
    List<Buildings> buildings; // downwards

    // mail delivery
    MailDeliveryPostcode postcode;
    int incoming_mail_volume;

    // refuse collection
    RefuseCollection route;
    double refuse_load;
    double recycle;
    //... many more concerns in here too
}
```

This shows how the various concerns are tangled into the *Lot* class; but each concern will also be scattered over a number of other classes in the system. How can you design such a complex system so that each concern is a separate module in the program's structure?

**APB** ▶ You're confusing class and package again◀

**KJX** ▶ Yes. or rather NO: I'm assuming most languages without open classes, and particularly languages and OO design "theory" (or "narrative" or "discourse") which — even for Smalltalk — has not separated class from packages. Certainly the underlying theory or ontology or whatever of OO design underlying the GOF generally doesn't make that distinction. Following Bertrand Meyer, say, we take a class to be closed (or open only to inheritance) — that is, a class IS a package — see comment earlier. I want to claim Smalltalk, ModularSmalltalk, Objective-C, and the rest, as implementing some form AOP◀

**Therefore:** *Build your program from mutually interrelated aspects, where each aspect models a single concern.*

**APB** ▶ Aha! aspect = package◀

**KJX** ▶ yep. and we need to work out how to handle that, subtly, so that separating to two comes as part of patterns' solutions◀

**KJX** ▶ or rather aspect = package where class  $\not\subset$  package◀

While — like classes — aspects can define the structure and behaviour of their own instances, unlike classes, aspects can also add to or modify the behaviour of other parts of the system. Heterarchical designs model crosscutting concerns explicitly as individual aspects — without a distinguished "base" program to which aspects are applied. Programs are composed of modules (which may be classes, or

aspects, or some combination) that can communicate and interact via a wide range of mechanisms, such as message sending, crosscutting advice, implicit invocation, events, depending on the implementation technologies.

### Example Resolved

Using aspects, we can separate out the various concerns of the MetroSim design so that each can be described implicitly and independently. First, we can write a class declaring only Lot's intrinsic state:

```
class Lot {
    String designation;
}
```

Then, we can model mail delivery as aspect that uses intertype declarations (introductions) to define the `postcode` and `incoming_mail_volume` fields into the Lot class. The code that manipulates and models routes can also be moved into this aspect out of the Lot class.

```
aspect MailDelivery {
    Lot.MailDeliveryPostcode postcode;
    int Lot.incoming_mail_volume;

    ... // code to manipulate routes
}
```

Similarly we can model refuse collection as a second aspect which inserts further fields into the Lot class. Again, this has the advantage that every Lot instance contains these fields, but they are not defined in the Lot class — the textual Lot class is a partial description of the Lot class that will exist in the final program, because each aspect can continue further attributes to the Lot definition.

```
aspect RefuseCollection {
    RefuseRoute Lot.route;
    double Lot.refuse_load;
    double Lot.recycle;

    ... // refuse collection methods
}
```

Finally, the political organisation — buildings, lots, suburbs, localities — modelled via the Composite pattern can also be described using an aspect. The key here is to realise that Composite is effectively a role-based design, and then to employ Aspect-oriented design techniques [57, 35, 30]. Because Composite is a reusable pattern, we will use two aspects: an abstract `CompositePattern` aspect to model the generic pattern, and a concrete `PoliticalModel` aspect to describe how that pattern is instantiated in this case [29].

The `CompositePattern` aspect defines inner interfaces `Composite`, `Component`, and `Leaf` to represent the three main roles (participants) in the pattern. Then the aspect uses intertype declarations to describe the fields and methods that belong to those roles. This is a common AspectJ idiom for supporting a role-based design **KJX** ► *Does it have a name? Should we collect idioms as well as patterns?* ◀ .

```
public abstract aspect CompositePattern {
    // declare top level Composite interface
    interface Component {
        void add(Component);
    }
}
```

```

    void remove(Component);
    List<Component> children();
}

// declare Leaf interface (mostly a placeholder)
interface Leaf extends Component {
}

// declare Composite interface
interface Composite extends Component { }

// declare Component field
void Component Component.parent;

// declare Leaf methods – as in GOF they throw exceptions
void Leaf.add(Component c) {throw UnsupportedOperationException(); }
void Leaf.remove(Component c) {throw UnsupportedOperationException(); }
List<Component> Component.children() {throw UnsupportedOperationException(); }

// declare Composite methods and fields
List<Component> Composite.children;
void Composite.add(Component c) {children.add(c); c.parent = this;}
void Composite.remove(Component c) {children.remove(c); c.parent = null;}
List<Component> Composite.children() {
    return Collections.unmodifiableList(children); }
}

```

The key to the aspect-oriented implementation is that a second `PoliticalModel` aspect that binds the roles in the `CompositePattern` to the classes in the program, including, eventually, `Lot`. Of these classes, `Building` is the only leaf class: all the others play the composite role in the pattern (because they can contain other components in the pattern). In idiomatic `AspectJ`, the aspect binds roles to classes by declaring that the classes implement the interfaces representing the roles: then, the introductions in the `CompositePattern` aspect will add the appropriate fields into every class that implement the roles.

```

aspect PoliticalModel extends CompositePattern {
    // classes implementing the Leaf role
    declare parents: Building implements Leaf;

    // classes implementing the Composite role
    declare parents: Suburb implements Composite;
    declare parents: Ward implements Composite;
    declare parents: Lot implements Composite;
}

```

Again, we can see that the `PoliticalModel` aspect describes how the composite pattern is instantiated in the program, and the abstract `CompositePattern` aspect gathers all the code and data structures required for the composite pattern. The aspect weaver will combine all the relevant features into the `Lot` class (and into the other base classes in the program) resulting in the same effective class definition as the object-oriented design shown above — but with very different modularity structures in the source code.

## Consequences

The key benefit of this pattern is decomposing a monolithic design into a series of smaller, local, crosscutting aspects that are then recomposed to reconstitute a whole system. As with most other aspect-oriented

designs, this pattern prefers high cohesion to low coupling: individual aspects will model only one concern in the program, but can be highly coupled to many other aspects in the program. The key liability of this pattern is this coupling: either the aspects have to be carefully designed to avoid malign interactions, or their interference must somehow be mediated. Where these interactions cannot be managed, they will make programs more difficult to understand and debug than monolithic designs: on the other hand, inasmuch as each aspect can be understood *locally* the program will be easier to use, extend, or modify.

Compared with the other patterns, the distinguishing feature of Heterarchical design is that there is generally no privileged “base” program that aspects monitor, patch, or extend: **APB** ► *not for your example*◄ **KJX** ► *so we need to fix the example if we can*◄ rather the structure of the program is an heterarchy, a rhizome, a collage of interconnected crosscutting parts. The distinction between classes (or components) and aspects dissolves: as all the modules are partial definitions of the system, acting simultaneously as components — defining objects, types, fields, methods, — and crosscutting aspects, advising and introducing behaviour onto other system components — as required by the overall nature of the problem concerns. Designing this kind of system is hard! In practice, it is most likely that “*some aspects will be more equal than others*”, that is some aspects will define fundamental structure, while others may decorate that structure. But because this distinction is not hardwired into the programming language or the pattern, there can be many aspects that act as a base program for certain concerns, others aspects can work as extensions to that concern, and different programmers can consider different aspects in different ways depending on the concerns they are focused upon.

## Known Uses

There are relatively few large, long-lived systems built as heterarchical designs — mostly due to software engineering’s long focus on *hierarchical* designs. Two important precursor systems are Knuth’s TeX [42] built using the WEB weaver; and later versions of the Self programming language which uses a weaver known as the Transporter [62].

**APB** ► *This is a a surprised to me*◄

**KJX** ► *Well the point is you can leave out (or not) various bits and pieces of the system via the Transporter — you don’t have to cart around a monolithic image. But I never worked out how to USE it anyway: I still used lots of little load files*◄

## Discussion

The principles of heterarchical design can be applied at many scales within a system. At a relatively small scale, aspects can be used to implement relationships [52] or design patterns [29] as we’ve sketched in the example above. At an intermediate level, role-based designs techniques produce heterarchical systems [30, 35], although generally the last step in such methodologies converts the design to use object-oriented programming languages [56, 63] in the same way early object-oriented methods included a step to translate designs into procedural languages, although a range of studies have described implementing (or refactoring) role-based designs to use aspects. At a large scale, Subject-oriented design aims to merge whole or partial programs [11].

As well as the AspectJ style used in the example, heterarchical designs can be built with language supporting forms of multiple inheritance, open classes, module composition, nested types, or open modules [3, 47, 49, 50, 51]. **KJX** ► *make sure names all lines up citations end*◄ The difference between these techniques parallels (in some sense) the variants of the Planned Extension pattern between implicit pointcut-based extension **APB** ► *explicit from the pov of the extension*◄ **KJX** ► *implicit from the pov fo the base code*◄ points and explicit calls to extensions: aspects “push” behaviour implicitly into other classes, while nested classes, modules, etc. “pull” that behaviour explicitly into the class being defined.

In the explicit style the intrinsic behaviour of a `Lot` could be moved into a `BasicLot` class, and then the `Lot` class configured into the system would inherit from `BasicLot` and bind the roles from other nested classes, something like:

```
class Lot extends BasicLot,  
    extends Politics<Section>.Leaf,  
    extends MailDeliver.Source,  
    extends RefuseCollection.Sink {  
}
```

noting that this example uses multiple inheritance directly.

The advantage of the explicit style here is that it is much easier to ascertain everything that has gone to make up the `Lot` object; the disadvantage is that `Lot`'s definition now must mention all the aspects that will be combined (rather than leaving that to the aspect definition). The overall structure of the design is the same in both cases.

Finally, although heterarchical designs can avoid the “tyranny of the dominant decomposition” [61] because they do not *have* a single dominant decomposition, this does not mean that an heterarchical design is necessarily better (against some criteria) than an hierarchical design, or that two heterarchical designs cannot be distinguished. Measuring coupling via option values [46], for example, can distinguish between two alternative heterarchical designs.

## 7 Discussion and Related Work

**KJX** ▶ *this section is not really revised, still uses the old pattern names, and still merges spectators and regulators under the name “intercession”* ◀

### 7.1 Comparing the Patterns

	Symmetry	Weave	Obliv.	Visibility <sup>†</sup>	Quant.	Introd.	Poly
<b>Intercession</b>	asym	either	yes	neither	homo	no	yes
<b>Ad-hoc Extension</b>	asym	either	yes	aspect to class	hetero	yes	no
<b>Planned Extension</b>	asym	either	no	class to aspect	hetero	few <sup>‡</sup>	yes
<b>Heterarchy</b>	sym	static	no	arbitrary	hetero	yes	yes

Figure 1: Taxonomy of Aspect-Oriented Patterns

<sup>†</sup> Kersten and Murphy’s terms are “closed”, “class-directional”, “aspect-directional”, and “open”. [36]    <sup>‡</sup> Only as permitted by extension interface

**APB** ▶ *rotate the figure* ◀    **KJX** ▶ *uncomment the other version?* ◀

Figure 1 presents a comparison of the four patterns. The first category in the table is whether the pattern is *asymmetrical* (makes a strong distinction between aspects and the base program) or *symmetrical* (makes little or no distinction). The first three patterns — Intercession, Ad-hoc and Planned Extension — are asymmetric; monitoring, patching and extending aspects are defined with respect to some base program that they monitor, patch, or extend. Heterarchical Design, on the other hand, is symmetric, making no distinction between classes and aspects, or base and monitoring program. The second column (tightly correlated with the first) considers whether the weaving is *static* (before the program begins running) or *dynamic* (afterwards). The first three patterns can be either static or dynamic: the dynamicity of the weave simply determines *when* the patterns can take effect. Again, due to the fact that it has no base program, a heterarchical design requires some static weaving for there to be a program that can be run: although in this case, further dynamic weaving may be possible to extend the design, perhaps employing one of the other patterns.

**APB** ▶ *fi the patterns are independent of the impl language, how can you assume “weaving” at all* ◀

**KJX** ▶ *well I guess you have to put it together somehow. the terms are standard in AOP I think — so e.g. comp filters or even CLOS can be described as “dynamic weaving”* ◀

The next category in the table is *obliviousness*, based on Filman and Friedman’s seminal definition of AOP as *Quantification and Obliviousness* [22] (quantification returns later in the table). We take obliviousness to mean that the base program is not aware of the aspects that may be bound into it: this is the case in the Intercession and Ad-Hoc extension pattern, but generally not in the planned extension and heterarchical design patterns. The following column uses Kersten and Murphy’s classification [36] of visibility (i.e. coupling) between aspects and classes to draw a finer distinction: in the Intercession pattern, classes and aspects are mutually unaware; in ad-hoc extension the extension aspects are tightly coupled to the code they are extending, while Planned extension requires extension points in the base code that are aware of the aspects (while the aspects see only the extension points but not the details of the base code). Finally in the Heterarchical Design pattern, every module in the program may function as both aspect and class; defining new structure and extending existing structure, so components are mutually aware. In both these categories in the table (as in all of them in fact) we do not intend to be prescriptive: we aim to describe the kind of structures built to express programmers’ intent in each

pattern. So Intercession will tend to have the base code oblivious of the aspect, or as Kersten and Murphy describe it, that the aspect is “closed” so neither class nor aspect are aware of each other. **APB** ► *This is completely false, as you say in the next sentence*◀ **KJX** ► *I’m paraphrasing the Kerstem & Murphy paper: “closed” is there term. Can someone else check back at that paper*◀ In a practical system, however, things will not be so simple: a basic Intercession aspect to trace execution will need to be configured so that it traces only some of the classes in the target program. Likewise, at the other end of the scale, in an Heterarchical design, although every module could in principle cross-cut every other module, such a design would be unmaintainable for any but the smallest program: the crosscutting will have to be carefully designed to capture the program’s concerns. For Heterarchical design, this crosscutting structure is not restricted *a priori* by the language constructs or some overarching design rule.

The quantification column then describes the type of relationship between between aspects and classes used in the pattern — the kind of crosscut primarily used in the design. Following Colyer and Clement [16] we distinguish *homogeneous* crosscutting from *heterogeneous* crosscutting: a homogeneous concern results in the “*consistent application of the same or similar policy in multiple places*” while heterogeneous crosscutting affects different places differently. **APB** ► *SHould have defined these terms before first us (in patterns)*◀ **KJX** ► *true. Question: how much of this discussion can be refactored into a forces section of a patterns paper?*◀

Here we find that the Intercession pattern uses primarily homogeneous crosscutting, while the other three patterns are primarily heterogeneous. The reasons for this are straightforward (and consistent with Colyer and Clement): Intercession concerns primarily monitoring, inspecting, or checking program execution and cannot affect program correctness, **APB** ► *Yes it can, which is why we shold split regulators and spectators*◀ **KJX** ► *yep. OK*◀ while the other patterns to a greater or lesser amount, can affect the correctness of the program by introducing or advising code — and so those interventions must be tailored to fit the precise join points to which they will apply.

The following category, also from Colyer and Clement [16], is *introductions*: whether patterns typically define new or additional fields or methods into classes or heterarchical aspects. We find intercession and planned extension tend not to use introductions, while ad-hoc extension and (especially) heterarchical design rely on them.

The final column is polymorphism, that is, the extent to which the pattern may take advantage of different implementations of similar aspects. This is similar, but more general than, the “Relationship Polymorphism” identified by Pearce and Noble [52]. **APB** ► *I don’t get it*◀ . **KJX** ► *Hmm — too much gratuitous self-citation?*◀ In some ways this is a counterpart of obliviousness, or visibility, which says which parts of a system have knowledge of other parts of a system: aspect polymorphism requires that there is an abstract interface between the base code and the aspect, and that the aspect may be changed (either statically or dynamically) as part of the pattern [21]. All the patterns (except ad-hoc extension) generally take advantage of that form of aspectual polymorphism: new kinds of monitoring may be added underneath the language in Intercession; the parameterisation in Planned extension explicitly (via delegates or events) or implicitly (via design rules or XPIs) generates an abstract interface to facilitate extension; Heterarchical designs are built out of this kind of interrelationship. Ad-hoc extension is the exception that proves the rule: because the patch aspect is tightly tailored to the class it is extending, it is not clear what aspect polymorphism could apply in this case.

Our classification of these patterns is at a relatively high level, because we are considering large-scale patterns. The taxonomy of patterns we present here is similar to the four categories of coupling between aspects and classes Kersten and Murphy developed from their AOP case study [36]; our categories are also defined by the (primary) use of introductions and obliviousness. Since we started this work by finding patterns, and then proceeded to analyse them, we consider that these points support our argument that these patterns classify much (but not all) of aspect-oriented design in practice.

There are also a number of finer-grained classifications of the properties of aspect-oriented designs and languages [41, 43, 58]. As with other kinds of patterns, different instantiations of the same pattern will differ in their implementation details: we have chosen this classification as we consider it best characterises the salient features of these patterns. In some ways, Clifton and Leavens' [13, 14] categories of *spectator* (originally *observer*) and *assistant* aspects, and then *superposition* and *extension* stories is also quite similar to our classification; our intercession pattern is similar to their observers, and our ad-hoc extension pattern is some ways similar to their local extension, however the classifications differ in detail. Similar ideas reoccur in Katz' classification into *spectative* and *regulative* aspects (intercession preserving total or partial correctness), versus more *invasive* aspects [34]. Perhaps the most well-known classification is Kiczales et al's *development* and *production* aspects: all our patterns may be used to design production aspects. Their development aspects, however, must use intercession because development aspects are designed to be removed from programs in production use, so the program cannot depend on their behaviour [38].

## 7.2 Combining Patterns

As with other kinds of pattern, these patterns may be used not just individually, but can also in combination. As a qualitative description of designs, there will always be cases where some design fits as well into one pattern as another. In *Design Patterns*, for example, the boundaries between Decorator and Proxy, or between State and Strategy, are often blurred.

We have found two ways in which these patterns are often combined, or how systems progress from using one pattern to using another. The first combination of patterns we call *abstraction recovery*. The problem here is that later patterns, Planned Extension or Heterarchical Design, are not oblivious they require program modules to be design in particular way. For Planned Extension, say, the design must be parameterised by the interfaces to which extensions can be attached. Similarly, to “play well with others” in an Heterarchical design, a class may need to follow particular design rules or conventions. In the real world, most software components may not provide the correct extension points or follow the right rules. In these cases, Intercession or Ad-hoc Extension can be used to modify a base component so that it does fit into the context of the other pattern.

For example, consider a shopping cart bean which can be used in some web-application, except for two problems. The first is that the bean raises exceptions which the rest of the application does not handle; and the second is that the application needs to customise shipping costs based on a user-supplied preference (courier, airfreight, instore pickup). The user-preference could be implemented as an instance of the Strategy Pattern (that is, as a Planned Extension) if the bean was coded to use a Strategy; the choice of exceptions is about conformance to wider design rules. Both of these problems can be resolved by applying other patterns to recover abstractions latent within the component. An ad-hoc extension can isolate the final phase in the bean's pricing calculation, effectively parameterising the component so that different strategies can be supplied by extending the interface established by the adhoc patch aspect. Thus, the ad-hoc extension is facilitating a later planned extension. Similarly, Intercession can be used to monitor and catch the unwanted exceptions thrown by the bean and, if necessary, to re-throw them to the rest of the application — here, the Intercession pattern is being used to support an Heterarchical design.

The second way the patterns work together is that over time, as a program is refactored and its design evolves, uses of one pattern may be replaced by uses of another pattern. Ad-hoc extension is often the pattern that is most likely to evolve: if an extension is simply a bugfixing patch then presumably the code will be moved into the next release of the base program and the need for the patching aspect will go away. On the other hand, both intercession and ad-hoc extensions can end up being used in stereotypical ways and to carry a significant amount of a system's behaviour; rules may be established about what intercessions or extensions should be applied to which parts of the program; and libraries of monitoring

or extension aspects be compiled. Then the base code can evolve to take account of the aspects that are likely to be used along with it — conventionally always invoking methods that will become join points for later pointcuts; adhering to naming conventions to match XPI pointcuts; including annotations to guide weaving; and being sure not to address certain concerns (security, synchronisation, transaction management) that will be handled by the aspects. At this point, the design is better described as using the Planned Extension pattern. Further evolution in the same direction — refactorings that eat away at the monolithic structure of the base program, and then increasing cooperation between aspects and components, can lead to an Heterarchical Design.

### 7.3 Modular Reasoning

Modular reasoning is increasingly of concern in aspect-oriented design [40, 60, 27, 51]. The question is what statements can be made about the behaviour of one module in an aspect-oriented program, when any other module may contain advice which can completely change the behaviour of the module we are considering — or indeed of the whole program. If such advice is common, then it is indeed difficult to see how aspect-oriented designs can be reliable (or indeed even be comprehensible to humans or machines).

It is important, then, to realise that each of these patterns has a different effect on the ease (or otherwise) of modular reasoning about the program. If we can identify which patterns are being used, we can have some idea about the likely ramifications for modular reasoning, alternatively, if we wish to preserve certain modular reasoning properties we can choose to use (or not use) particular patterns in each design.

The Intercession pattern should not affect the modular reasoning properties of the program, at least as far as the core “functional” behaviour of the program is concerned. Aspects used in the Intercession pattern should be *harmless* [17] **APB** ▶ *true of spectators but not regulators*◀ — that is they should not affect the partial correctness of the program: certainly monitoring its behaviour or prematurely terminating execution, but not changing the result of a computation that terminates normally. This means that the base program can be reasoned about as if no aspects were involved.

**APB** ▶ *The very word interceed meas to (try to) change the outcome! Don't use the word if you mean something else.* ◀

**KJX** ▶ *Again this is a term from Bracha and Ungar; but splitting the patterns will solve the problem*◀

The advice in a patch aspect introduced by the Ad-hoc Extension pattern, however, may certainly affect the base program: indeed that is the point of the pattern. There are two advantages for reasoning about programs using this pattern, however. The first is that, again by the nature of the pattern, its pointcuts will be heterogeneous and particularly tight — often affecting only one method in one class. This means that programmers will have to reconsider only a small amount of the program in the presence of such an aspect. The second is that because the system as a whole is to continue to operate even in the presence of such an aspect, the effects of any changes introduced by the advice must be small scale, compared to the program as a whole.

The Planned Extension pattern introduces aspects that are more integral to the structure and behaviour of the program. The crucial point of this pattern is the interface between base code and extensions — the extension interfaces, design rules, XPIs, or contracts about when delegates will be called or events sent. These interfaces should support modular reasoning about either the base program or the extension aspects in isolation. Ultimately, the behaviour of the system — especially when the base code delegates decisions to its extensions — can only be reasoned about as a whole, but again the explicit interfaces will limit the scope of modifications introduced by aspects.

Finally, the behaviour of a system built using Heterarchical Design can generally only be reasoned about as a whole — or rather, only reasoned about *when the whole of the system configuration is present*. Given a system, techniques such as aspect-aware interfaces, or IDEs which compute aspect weaving and method combinations and then visualise the result [40, 12] can allow modular reasoning about any

particular configuration of the system. The advantage of this pattern is that most concerns can be reified as aspects, and so interconcern interactions will be made explicit through interfaces and pointcuts (albeit only when all the modules that will comprise a system configuration are known). This means that reasoning about the global behaviour of the program can be guided by the interfaces and the pointcuts, rather than that information having to be somehow intuited or analysed from the tangled code of a non-aspect-oriented view of the system.

More importantly, we expect that the fundamental tradeoffs evident in reasoning about the the other patterns (and visible in Figure 1 above) will hold even in the case of heterarchical design. That is, we expect that aspects that use homogeneous pointcuts and so affect large amounts of other code will not alter the base computation of the program, and aspects which do make changes to a base program will use heterogeneous pointcuts that only affect a small part of the systems' behaviour or will be applied via well-defined interfaces. Figure 1 does not include aspects with both wide, homogeneous pointcuts and significant effects upon the program's domain computation — such as “add three to every int variable on every read”. We call these designs “*pathological aspects*” and consider them to be anti-patterns — that is, designs to be avoided, because the combination of wide applicability and significant changes to the base program make effective reasoning about a program practically impossible. We have not found any examples of pathological aspects in practical systems.

#### 7.4 Patterns vs Aspects

There has been a range of other work on interrelating aspects and patterns. The earliest is probably work using aspects to reify and implement the Gamma et al *Design Patterns* [29, 25, 19]. More recently, particular (generally small-scale) patterns have been written to describe idioms or designs for aspect-oriented programming [44]. We expect that most smaller-scale AOP patterns should fit in to a category described by the larger-scale patterns we present here.

Furthermore, Figure 2 shows how the *Design Patterns* can be classified in our taxonomy. We find the majority of the patterns are planned extensions: this should not be surprising since the key technique underlying *Design Patterns* is to parameterise the design of object-oriented frameworks by introducing “hot spots” (i.e. extension points). The next most common category of patterns are those supporting Heterarchical Design — designs where two or more concerns (often structure and behaviour, or interface and implementation) are tangled in a classical OO decomposition, leading to several patterns that introduce dual inheritance hierarchies and then have to describe the programming idioms necessary to make them work in object-oriented languages. Four patterns are primarily Intercession: Singleton, Flyweight and Memento manage the allocation, duplication, and retention of objects' state, while Proxy describes the classic “wrapper” or “interceptor” implementation of intercession. Finally, only the Facade pattern seems to have little to do with supporting aspect-oriented design: rather it introduces a classical *hierarchical* module interface around a subsystem.

This analysis certainly lends support to the thesis that the *Design Patterns* are symptoms of weaknesses in object-oriented languages. We take a complementary view: that the design patterns are best practice to encode aspect-oriented designs in object-oriented languages, and that (again, unsurprisingly) aspect-oriented languages can provide significantly better support for such designs! That the patterns are prevalent in OO programs, however, probably indicates that aspect-oriented designs are a good fit to many problems currently addressed using objects.

#### 7.5 AOP vs Reflection

Given that AOP emerged from reflective systems — and that one of our patterns, Intercession, is a category of reflexive programming [7] — it is interesting to consider the relationship between these

<b>Creational Patterns</b>		
Abstract Factory	creates related objects	planned extension
Builder	creates complex objects	planned extension
Factory Method	creates subclasses	planned extension
Prototype	exemplary object	planned extension
Singleton	single instance	intercession
<b>Structural Patterns</b>		
Adaptor	converts interfaces	heterarchical design
Bridge	decouple abstraction and implementation	heterarchical design
Composite	model recursive tree structure	heterarchical design
Decorator	add responsibilities to objects	planned extension / heterarchical design
Facade	interface for a subsystem	<i>heterarchical design</i>
Flyweight	save memory of similar objects	intercession
Proxy	surrogate for access control	intercession
<b>Behavioural Patterns</b>		
Chain of Responsibility	handle requests	heterarchical design
Command	request as an object	planned extension
Interpreter	interpret a language	planned extension
Iterator	iteration cursor	heterarchical design
Mediator	encapsulate interactions	heterarchical design
Memento	snapshot of object's state	intercession
Observer	update dependents	heterarchical design / planned extension
State	change behaviour	heterarchical design
Strategy	vary algorithms	planned extension
Template Method	subclasses change algorithms	planned extension
Visitor	represent traversal operations	planned extension / heterarchical design

Figure 2: Classification of Patterns

patterns and reflection more generally.

**KJX** ► *note: changes terms to “syntax” and “semantics”. Why was I too stupid not to see that before? All this should probably end up in another paper anyway (“reflecting on aspects”)◀*

Filman and Friedman’s obliviousness criteria [22], for example, is quite closely related to reflection: by definition, a base program in a reflexive system is oblivious to its meta-level. We consider that the two key differentiators between aspect-oriented programming’s support for reflection (particularly intercession) are the *semantics* of the reflective model (how does the aspect model the base code?) and the *syntax* used to represent that model (how do you program the aspect). From this perspective, a key to AOP (rather than reflexive OO systems) is that AOP’s syntax and semantics often differ strongly from the base language.

Consider the CLOS MOP [37], often considered to be the epitome of meta-reflexive designs. The semantics of the MOP’s model of CLOS is complex, based primarily (and simultaneously) around generic functions and classes, and its syntax is an object-oriented framework. This means both that the ontology of the reflexive layer is tied tightly to that of the underlying language, and secondly that to program against the MOP, one has to extend an object-oriented framework, often a difficult task.

We can compare this design to that of Composition Filters for example [2]. Composition filters have a rather more restrictive model of the program (primarily, an individual object receives messages), and a simplified, specialised syntax (programmers mostly configure, compose, and install preexisting filters). Comparing further with AspectJ [39], we see the earliest AspectJ model semantics were much less object-oriented than either CLOS or Composition Filters, being primarily based on wildcarded function names with support for objects, inheritance, etc, being secondary, and an extended subset of Java as the syntax for writing pointcuts and advice. Spring AOP [33] has a similar model semantics to AspectJ, but this time implemented using the more traditional syntax of an OO framework. The effect of these design choices is that AOP languages can provide the power of MOP-based stems but with much less programmer effort: reducing the OO focus from the reflexive model makes designs that crosscut objects much easier to express; a specialised “little language” — once learnt — is simpler than instantiating an object-oriented framework.

Finally, note that compared to CLOS or Smalltalk, Java’s reflexive features are very weak. This may explain a key advantage of AspectJ, particularly for common intercession tasks such as profiling and logging, since it provides the easiest implementation of Intercession and Ad-hoc Extension for Java programmers, while Parametric Extension and Heterarchical Designs can be addressed by using OO Design Patterns [64].

**APB** ► *Are you suggesting that an aspect language is a DSL for doing certain kinds of reflexive programming in a “safe” way?◀*

**KJX** ► *Yes! I think I am ... but that insight probably goes back to Christa Lopes (who deserves a mention here, I think). Or rather, I think that’s about half of it: the other half being separating modelling and reuse, i.e. classes vs packages. the section following develops that theme: that the OO patterns are mostly about perveting OO modelling mechanisms to provide flexible packaging◀*

**KJX** ► *And I’ve realised you don’t like my overuse of “rather”. Perhaps I should put in some more “actually”s too :-)*◀

## 7.6 From Patterns to Language Features

As programming languages evolve, it is common that constructs are added to a next generation of programming languages to support idioms or patterns required in previous versions. To take two relatively recent examples: comparing the inheritance designs of Java and C# with earlier languages such as C++ and Eiffel, the newer languages have explicit support for (or differentiate between) concrete classes, abstract classes, and interfaces; and implementation and interface inheritance. An interface is a pattern or

idiom in C++ or Eiffel but is a language feature in Java. Even over the shorter lifetimes of Java and C# we can see, for example, explicit iteration using the Iterator pattern being subsumed into collection-aware forms of the `for` loop.

The four patterns in this paper raises a similar question about language support for aspect-oriented design: how should the next generation of AO languages support these patterns? Should they attempt to support only a subset of these patterns? Should they attempt to support *all* these patterns within a single, general “aspect” construct? Or would separate specialised language constructs better support each pattern?

We have neither time, space, nor inclination to provide detailed designs for future AOP languages in this paper. We do, however, think that considering these patterns can assist in the design of further languages — especially since support for our more general AOP patterns will support many of the object-oriented patterns as well. Looking at the solutions and known uses we have described for each pattern, it does seem that the clearer solutions to one pattern are not generally the most appropriate solutions to other patterns. New language proposals have generally (albeit implicitly) favoured one or two of these patterns and excluded others: XPIs [27], for example, addresses Planned Extension; Harmless Advice [17] addresses only Intercession; nested type languages [4, 49, 50, 20] address Heterarchical Design and some kinds of Planned Extensions (where only static extensions are required); delegates provide dynamic extensions; support for class versions (e.g. .Net assemblies or Classboxes [6]) may suffice for Ad-hoc Extensions. Some comparative analyses [64, 9] have already been undertaken: we hope that these patterns may enable such analyses to be carried out across the spectrum of aspect-oriented design styles.

## 8 Conclusion

The immediate contribution of this paper is the identification and presentation of five patterns of aspect-oriented design: Spectator, Regulator, Ad-hoc Extension; Planned Extension; and Heterarchical design. For each pattern we present a brief example, analysis, and known uses in several successful systems. The underlying virtue of identifying these four patterns is the hypothesis that there is not one just one “ideal” or “exemplary” style of aspect-oriented software development, but rather a range of styles applicable in different circumstances, addressing different concerns, and with a range of benefits and liabilities. Rather than treating Aspect-Oriented as though it is a monolith, we hope that researchers, critics and programmers will be able to use these patterns to identify which aspect-oriented design is being attempted, and then use the appropriate patterns for the task at hand.

## References

- [1] M. Aksit and L. Bergmans. Obstacles in object-oriented software development. In *OOPSLA*, 1992.
- [2] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. In *Proceedings of the ECOOP '93 Workshop on Object-Based Distributed Programming*, LNCS 791, 1994.
- [3] J. Aldrich. Open modules: Modular reasoning about advice. In *ECOOP*, 2005.
- [4] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. Overview of CaesarJ. *TAOSD*, LNCS 3880, 2006.
- [5] The AspectJ 5 development kit developers notebook, +<http://www.aspectj.org/+>.

- [6] A. Bergel, S. Ducasse, and O. Nierstras. Classbox/J: Controlling the scope of change in Java. In *OOPSLA*, 2005.
- [7] G. Bracha and D. Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In *OOPSLA*, 2004.
- [8] F. P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4), Apr. 1987.
- [9] R. Chitchyan, I. Sommerville, and A. Rashid. An analysis of design approaches for crosscutting concerns. In *Ws. on Identifying, Separating and Verifying Concerns in the Design*, 2002.
- [10] S. Clarke and E. Baniassad. *Aspect-Oriented Analysis and Design: The Theme Approach*. Addison-Wesley, 2005.
- [11] S. Clarke, W. Harrison, H. Ossher, and P. Tarr. Subject-oriented design: Towards improved alignment of requirements, design, and code. In *OOPSLA*, 1999.
- [12] A. Clement, A. Colyer, and M. Kersten. Aspect-oriented programming with ajdt. In *ECOOP Workshop on Analysis of Aspect-Oriented Software*, 2003.
- [13] C. Clifton and G. T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In *FOAL*, 2002.
- [14] C. Clifton and G. T. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy. Technical Report TR #03-15, Department of Computer Science, Iowa State University, 2003.
- [15] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 130–145, 2000.
- [16] A. Colyer and A. Clement. Large-scale aosd for middleware. In *AOSD*, 2004.
- [17] D. S. Dantas and D. Walker. Harmless advice. In *POPL*, 2006.
- [18] R. Dunn-Krahn, J. Maple, and Y. Coady. The crisis in systems code maintenance: Sourceforge, we have a problem. In *OOPSLA Onward! Film*, 2005.
- [19] E. Eide, A. Reid, J. Regehr, and J. Lepreau. Static and dynamic structure in design patterns. In *ICSE*, 2002.
- [20] E. Ernst. *gBeta—A Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, University of Aarhus, Denmark, 1999.
- [21] E. Ernst and D. H. Lorenz. Aspects and polymorphism in aspectj. In *AOSD*, 2003.
- [22] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Aspect-Oriented Software Development*. Addison-Wesley, 2005.
- [23] E. Gamma. Extension object. In *Pattern Languages of Program Design*, volume 3. Addison-Wesley, 1997.
- [24] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.

- [25] A. Garcia, C. Sant’Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa. Modularizing design patterns with aspects: A quantitative study. In *AOSD*, pages 3–14. ACM Press, 2005.
- [26] D. Garlan, G. E. Kaiser, and D. Notkin. Using tool abstraction to compose systems. *IEEE Computer*, 25(6):30, June 1992.
- [27] W. G. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan. Modular software design with crosscutting interfaces. *IEEE Software*, Jan/Feb 2006.
- [28] S. Gudmundson and G. Kiczales. Addressing practical software development issues in AspectJ with a pointcut interface. In *ECOOP Workshop on Advanced Separation of Concerns*, 2001.
- [29] J. Hannemann and G. Kiczales. Design pattern implementations in Java and AspectJ. In *OOPSLA*, pages 161–173. ACM Press, 2002.
- [30] J. Hannemann, G. C. Murphy, and G. Kiczales. Role-based refactoring of crosscutting concerns. In *AOSD*, pages 135–145. ACM Press, 2005.
- [31] I. Jacobson and P.-W. Ng. *Aspect-Oriented Software Development with Use Cases*. AW, 2005.
- [32] JBoss.org. JBoss AOP. <http://labs.jboss.com/jbossaop>, 2006.
- [33] R. Johnson. The spring framework - reference documentation. <http://www.springframework.org/docs/reference>, 2006.
- [34] S. Katz. Aspect categories and classes of temporal properties. *TAOSD*, LNCS 3880, 2006.
- [35] E. A. Kendall. Role model designs and implementations with aspect-oriented programming. In *OOPSLA*, pages 353–370. ACM Press, 1999.
- [36] M. Kersten and G. Murphy. Atlas: A case study in building a web-based leading environment using aspect-oriented programming. In *oopsla*, 1999.
- [37] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
- [38] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10), Oct. 2001.
- [39] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect oriented programming. In *ECOOP*, 1997.
- [40] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *ICSE*, 2005.
- [41] G. Kiczales and M. Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *ECOOP*, pages 195–213. Springer-Verlag, 2005.
- [42] D. E. Knuth. *T<sub>E</sub>X: The Program*, volume B of *Computers & Typesetting*. Addison-Wesley, 1986.
- [43] S. Kojarski and D. H. Lorenz. Modeling aspect mechanisms: A top-down approach. In *ICSE*, 2006.
- [44] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning, 2003.
- [45] K. Lieberherr. Controlling the complexity of software designs. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 2–11. ACM Press, 2004.

- [46] C. Lopes and S. Bajracharya. Assessing aspect modularizations using design structure matrix and net option value. *TAOSD*, LNCS 3880, 2006.
- [47] S. McDirmid and W. Hsieh. Aspect-oriented programming in Jiazzi. In *AOSD*, 2003.
- [48] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.
- [49] N. Nystrom, X. Qi, and A. C. Myers. J&: Nested intersection for scalable software composition. In *OOPSLA*, 2006. To Appear.
- [50] M. Odersky and M. Zenger. Scalable component abstractions. In *OOPSLA*, 2005.
- [51] N. Ongkingco, P. Avgustinov, J. Tibble, L. Hendren, O. de Moor, and G. Sittampalam. Adding open modules to AspectJ. In *AOSD*, 2006.
- [52] D. J. Pearce and J. Noble. Relationship aspects. In *AOSD*, 2006.
- [53] D. J. Pearce, M. Webster, R. Berry, and P. H. Kelly. Profiling with aspectj. *Software: Practice and Experience*, 37(7):747–777, 2007.
- [54] S. Putz. Managing the evolution of Smalltalk-80 systems. In *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, 1984.
- [55] A. Rashid and A. Moreira. Domain models are not aspect free. In *MODELS*, 2006.
- [56] T. Reenskaug. *Working with Objects: The OOram Software Engineering Method*. Manning Publications, 1996.
- [57] D. Riehle. Composite design patterns. In *ECOOP*, 1997.
- [58] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *FSE*, 2004.
- [59] R. M. Stallman. EMACS the extensible, customizable self-documenting display editor. In *Sym. on Text Manipulation*, 1981.
- [60] F. Steimann. The paradoxical success of aspect-oriented programming. In *OOPSLA*, 2006.
- [61] P. Tarr, H. Ossher, S. M. Sutton Jr., and W. Harrison. N degrees of separation: Multi-dimensional separation of concerns. In *Aspect-Oriented Software Development*, pages 37–61. Addison-Wesley, 2005.
- [62] D. Ungar. Annotating objects for transport to other worlds. In *OOPSLA*, 1995.
- [63] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. P-H, 1990.
- [64] B. Xin, E. Eide, W. C. Hsieh, and S. McDirmid. A comparison of Jiazzi and AspectJ for feature-oriented software product line development. Submitted for publication, 2005.