Patterns of Aspect-Oriented Design

James Noble[1,2], Arno Schmidmeier[3], David J. Pearce[2], Andrew P. Black[4]

(1) Imperial College London
(2) Permantent: Victoria University of Wellington, {kjx,djp}@mcs.vuw.ac.nz
(3) AspectSoft, arno@aspectsoft.de
(4) Portland State University, black@cs.pdx.edu

January 18, 2008

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.

1 Introduction

Aspect-orientation as a research area is at least ten years old — with AspectJ [30] first being described in the late 1990s, and key precursors such as subject-oriented design [8], Composition Filters [1, 2], and Adaptive Programming [33] emerging a good five years earlier. Aspect-oriented programming languages or systems, especially AspectJ and web architectures [26, 25], have been used in production environments for several years, and a range of new programming languages [35, 37, 38], and design and analysis methods [6, 7, 24] are emerging from the research community.

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 [48, 6], the advocacy and analysis of various language

features [14, 46, 9], or on case studies [29, 11, 28]), 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* [16], 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 et al. 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 can 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.

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:

```
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 [11]. 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. 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 [34].

Apart from the observance of such design rules, the base code is uncoupled from (that is, it is *oblivious to* [14]) 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 [32]; 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 [23] . 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]
```

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" (note, 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 [41]. This profiles various program metrics (e.g. heap usage and object lifetime) using aspects which maintain state that characterises 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 [9]. Katz defines "Spectative aspects" with respect to properties of a base program, rather than with respect to the program itself [27]. 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.

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 [46].

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 of 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;
        }
    }
}
```

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 Beanz 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 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 Beanz 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 [11] 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 OrderBeanz extends Beanz {
    ...
    int orderNumber() { return this.orderNumber; }
}
```

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

```
aspect BeanzSecurity {
    before() : execution(* Beanz+.*(..)) {
        if (!beanzSecurityManager.check(
                thisJoinPointStaticPart.getSignature().getName()))
            throw new SecurityException(...);
    }
}
```

For transactions:

```
aspect BeanzTransactionMonitor{
    Object around() : execution(* Beanz+.*(..)) {
        Transaction txn = beanzTxnManager.getTransaction(...);
        Object rv = null;
        try {
            rv = proceed();
            txn.commit();
            txn.close();
            return rv;
        } catch(TransactionException e) {
            txn.rollback();
            txn.close();
```

```
            throw e;
        }
    }
}
```

Note that the pointcuts of these aspects use wildcards to catch all invocations of Beanz 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* [14]) 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. 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 [25, 26].

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 [12]. 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 [9, 46, 27]. The name "Regulator" is derived from Katz's "regulative aspects" [27].

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 [26].

4 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. Because re-use places new demands on old code, however, 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 [10] 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 [10]. 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#[1]. 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:

```
package org.hacker.webShop;
...
class StringUtil {
```

[1] 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.

```
    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);
    }
}
```

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) [11] 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]);
        }
    }
}
```

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 [42]: 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 [51]; patch aspects are a structured, language supported alternative to patching [13].

Discussion

The key difference between Patch and Regulator is that, in this pattern, the base code is not oblivious to 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

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 [36] 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 its subclasses) to identify where highlighting is implemented and then to work out how to override some parts of them to support VisualAlgol. 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* [16], 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 [11] 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 reify a potential extension point for highlighting by separating out the highlighting behaviour into a method, just as in the Template Method pattern. 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() {};
}
```

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(..)));
}
```

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);
        }
    }
}
```

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.

These abstract definitions are the key liability 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.

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 [47], Smalltalk's dependencies [18], Eclipse's extension points, and .Net's delegates. The J& [37] 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 [15] describes how extensions can be reified in object-oriented design: this pattern shows how aspects can provide more straightforward extensions.

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 [20], eXtension Point Interfaces (XPI)s [19], and pointcut declarations in open modules [3, 39]), and hook or delegate or observer pattern based implementations, where the base code explicitly[2] 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
```

[2]We call these "explicit" calls because the base code contains explicit message sends to the extensions. These kind of message sends are also known as "implicit invocation" [17] 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.

```
    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 [4] 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 [19, 20].

6 Heterarchical Design

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" [5]. 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) [49] 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 that 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, buses, 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 [43]. 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.

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?

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

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 and out of the Lot class.

```
aspect MailDelivery {
    MailDeliveryPostcode Lot.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 [45, 28, 22]. 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 [21].

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 .

```
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
    Component Component.parent;

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

    // declare Composite methods and fields
    List<Component> Composite.children = new ArrayList<Component>();
    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 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: 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, other 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 [31] built using the WEB weaver; and later versions of the Self programming language which uses a weaver known as the Transporter [50].

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 [40] or design patterns [21] as we've sketched in the example above. At an intermediate level, role-based designs techniques produce heterarchical systems [22, 28], although generally the last step in such methodologies converts the design to use object-oriented programming languages [44, 52] 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 [8].

As well as the AspectJ style used in the example, heterarchical designs can be built with languages supporting forms of multiple inheritance, open classes, module composition, nested types, or open modules [3, 35, 37, 38, 39]. The difference between these techniques parallels (in some sense) the variants of the Planned Extension pattern between implicit pointcut-based extension 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" [49] because they do not *have* a single dominant decomposition, this does not mean that a heterarchical design is necessarily better (against some criteria) than a hierarchical design, or that two heterarchical designs cannot be distinguished. Measuring coupling via option values [34], for example, can distinguish between two alternative heterarchical designs.

7 Conclusion

The immediate contribution of this paper is the identification and presentation of five patterns of aspect-oriented design: Spectator, Regulator, Patch, 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-Orientation 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] The AspectJ 5 development kit developers notebook, http://www.aspectj.org/.

[5] F. P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4), Apr. 1987.

[6] 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.

[7] S. Clarke and E. Baniassad. *Aspect-Oriented Analysis and Design: The Theme Approach*. Addison-Wesley, 2005.

[8] S. Clarke, W. Harrison, H. Ossher, and P. Tarr. Subject-oriented design: Towards improved alignment of requirements, design, and code. In *OOPSLA*, 1999.

[9] C. Clifton and G. T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In *FOAL*, 2002.

[10] 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.

[11] A. Colyer and A. Clement. Large-scale aosd for middleware. In *AOSD*, 2004.

[12] D. S. Dantas and D. Walker. Harmless advice. In *POPL*, 2006.

[13] R. Dunn-Krahn, J. Maple, and Y. Coady. The crisis in systems code maintenance: Sourceforge, we have a problem. In *OOPSLA Onward! Film*, 2005.

[14] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Aspect-Oriented Software Development*. Addison-Wesley, 2005.

[15] E. Gamma. Extension object. In *Pattern Languages of Program Design*, volume 3. Addison-Wesley, 1997.

[16] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[17] D. Garlan, G. E. Kaiser, and D. Notkin. Using tool abstraction to compose systems. *IEEE Computer*, 25(6):30, June 1992.

[18] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[19] 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.

[20] 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.

[21] J. Hannemann and G. Kiczales. Design pattern implementations in Java and AspectJ. In *OOPSLA*, pages 161–173. ACM Press, 2002.

[22] J. Hannemann, G. C. Murphy, and G. Kiczales. Role-based refactoring of crosscutting concerns. In *AOSD*, pages 135–145. ACM Press, 2005.

[23] IBM Corporation. *LISP/370 Program Description/Operations Manual*, Mar. 1978. SH20-2076. Available through IBM branch offices.

[24] I. Jacobson and P.-W. Ng. *Aspect-Oriented Softare Development with Use Cases*. AW, 2005.

[25] JBoss.org. JBoss AOP. `http://labs.jboss.com/jbossaop`, 2006.

[26] R. Johnson. The spring framework - reference documentation. `http://www.springframework.org/docs/reference`, 2006.

[27] S. Katz. Aspect categories and classes of temporal properties. *TAOSD*, LNCS 3880, 2006.

[28] E. A. Kendall. Role model designs and implementations with aspect-oriented programming. In *OOPSLA*, pages 353–370. ACM Press, 1999.

[29] M. Kersten and G. Murphy. Atlas: A case study in building a web-based leadning environment using aspect-oriented programming. In *oopsla*, 1999.

[30] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect oriented programming. In *ECOOP*, 1997.

[31] D. E. Knuth. *TEX: The Program*, volume B of *Computers & Typesetting*. Addison-Wesley, 1986.

[32] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning, 2003.

[33] K. Lieberherr. Controlling the complexity of software designs. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 2–11. ACM Press, 2004.

[34] C. Lopes and S. Bajracharya. Assessing aspect modularizations using design structure matrix and net option value. *TAOSD*, LNCS 3880, 2006.

[35] S. McDirmid and W. Hsieh. Aspect-oriented programming in Jiazzi. In *AOSD*, 2003.

[36] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[37] N. Nystrom, X. Qi, and A. C. Myers. J&: Nested intersection for scalable software composition. In *OOPSLA*, 2006. To Appear.

[38] M. Odersky and M. Zenger. Scalable component abstractions. In *OOPSLA*, 2005.

[39] N. Ongkingco, P. Avgustinov, J. Tibble, L. Hendren, O. de Moor, and G. Sittampalam. Adding open modules to AspectJ. In *AOSD*, 2006.

[40] D. J. Pearce and J. Noble. Relationship aspects. In *AOSD*, 2006.

[41] D. J. Pearce, M. Webster, R. Berry, and P. H. Kelly. Profiling with AspectJ. *Software: Practice and Experience*, 37(7):747–777, 2007.

[42] S. Putz. Managing the evolution of Smalltalk-80 systems. In *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, 1984.

[43] A. Rashid and A. Moreira. Domain models are not aspect free. In *MODELS*, 2006.

[44] T. Reenskaug. *Working with Objects: The OOram Software Engineering Method*. Manning Publications, 1996.

[45] D. Riehle. Composite design patterns. In *ECOOP*, 1997.

[46] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *FSE*, 2004.

[47] R. M. Stallman. EMACS the extensible, customizable self-documenting display editor. In *Sym. on Text Manipulation*, 1981.

[48] F. Steimann. The paradoxical success of aspect-oriented programming. In *OOPSLA*, 2006.

[49] 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.

[50] D. Ungar. Annotating objects for transport to other worlds. In *OOPSLA*, 1995.

[51] L. Wall. *patch (1)*, 1985. Posted to `mod.sources`.

[52] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. P-H, 1990.