Separating Crosscutting Concerns Across the Lifecycle: From Composition Patterns to AspectJ and Hyper/J

Siobhán Clarke
Department of Computer Science
Trinity College
Dublin 2, Republic of Ireland
+353 1 6083690
siobhan.clarke@cs.tcd.ie

Robert J. Walker
Department of Computer Science
University of British Columbia
201-2366 Main Mall
Vancouver, BC, Canada V6T 1Z4
walker@cs.ubc.ca

Technical Report TCD-CS-2001-15 and UBC-CS-TR-2001-05

ABSTRACT

Requirements that have a crosscutting impact on software (such as distribution or persistence) present many problems for software development that manifest themselves throughout the lifecycle. Inherent properties of crosscutting requirements, such as *scattering* (where their support is scattered across multiple classes) and *tangling* (where their support is tangled with elements supporting other requirements), reduce the reusability, extensibility, and traceability of the affected software artefacts. Scattering and tangling exist both in designs and code and must therefore be addressed in both.

To remove scattering and tangling properties, a means to separate the designs and code of crosscutting behaviour into independent models or programs is required. This paper discusses approaches that achieve exactly that in either designs or code, and presents an investigation into a means to maintain this separation of crosscutting behaviour seamlessly across the lifecycle. To achieve this, we work with *composition patterns* at the design level, *AspectJ* and *Hyper/J* at the code level, and investigate a mapping between the two levels. Composition patterns are a means to separate the design of crosscutting requirements in an encapsulated, independent, reusable, and extensible way. AspectJ and Hyper/J are technologies that provide similar levels of separation for Java code. We discuss each approach, and map the constructs from composition patterns to those of AspectJ and Hyper/J. We first illustrate composition patterns with the design of the Observer pattern, and then map that design to the appropriate code. As this is achieved with varying levels of success, the exercise also serves as a case study in using those implementation techniques.

Keywords

Composition patterns, subject-oriented design, aspect-oriented programming, AspectJ, hyperspace, subject-oriented programming, Hyper/J, separation of concerns, crosscutting requirements and functionality, reuse, development lifecycle.

1. INTRODUCTION

Requirements that have a crosscutting impact on software (such as distribution, persistence, etc.) present well-documented difficulties for software development [6, 13, 20, 24, 29]. The support for crosscutting behaviour, by its nature, needs to be *scattered* across potentially the full design and code of the system. In addition, its support may also be *tangled* with the design and code of multiple other requirements. Scattering and tangling impact comprehensibility, traceability, evolvability, and reusability of software artefacts. These problems are present throughout the development lifecycle, and must therefore be addressed across the lifecycle.

Software design is an important activity within the software lifecycle, with benefits that include early assessment of the technical feasibility, correctness, and completeness of requirements; management of complexity and enhanced comprehension; greater opportunities for reuse; and improved evolvability [8, 9]. However, the benefits of software design are often not realised; as described in [6], a structural mismatch, between the way requirements tend to be specified (in terms of features and capabilities) and object-oriented specifications, motivates a need to more closely align object-oriented software designs with the structure of requirements. This can be achieved by providing a model that supports the separation (and subsequent composition) of design models for different requirements. Decomposition in this manner removes requirement scattering and tangling properties from software design, thereby also removing their negative impact. Designs and code map well to each other when they are both within the object-oriented paradigm. The model described in [6] supported a further mapping of object-oriented designs from requirements' specifications, thereby enhancing traceability throughout the lifecycle.

In order to achieve a level of traceability for crosscutting requirements, an approach is required that provides a means to separate the designs and code of crosscutting behaviour into separate, independent models or programs. The standard object-oriented paradigm is not capable of achieving the required level of encapsulation and separation for crosscutting requirements. Recently, however, there has been considerable focus on this problem in both designs and code. In [7], we presented *composition patterns* (CPs), a means for separating the designs of crosscutting requirements into reusable, extensible design models. With CPs, the constraints and interactions of crosscutting behavioural elements may be designed independently of the elements with which they may interact or constrain. Using CPs, traceability from crosscutting requirements' specifications is achieved. In addition, approaches with support-

ing technologies (such as languages or tools) have emerged to address the separation of crosscutting behaviour for object-oriented code. Aspect-oriented programming [20] provides language mechanisms that explicitly capture crosscutting code structure, with an environment for Java provided in the AspectJ language [19] and its attendant tools. Multi-dimensional separation of concerns [29] supports the separation of multiple, arbitrary kinds (dimensions) of concern, and has a supporting environment for Java called Hyper/J [28]. Both these approaches present a paradigm for separation and encapsulation of crosscutting concerns in code.

However, to date, there has been little focus on any relationship between crosscutting designs and crosscutting code. CPs can be used without a corresponding implementation technology, as composition semantics are part of the model. CPs can therefore be composed with base designs, though the resulting output design model will have the scattering and tangling properties discussed previously. AspectJ and Hyper/J can also be used independently of a supporting design paradigm. So, what's missing? We believe that to truly realise the benefits of separation of crosscutting concerns across the lifecycle, there must be a seamless and traceable mapping from the design to supporting code. This has benefits for both designers and coders. From a designer's perspective, it remains important to be able to compose the designs for validation purposes. However, if the composed design is implemented directly, then the implementation will display scattering and tangling properties, with their related negative impact. The designer will find it difficult to communicate any changes to the crosscutting behaviour to the coders. Evolution of existing crosscutting designs, and additions of new crosscutting behaviour, will be difficult to trace, synchronise, and implement. From a coder's perspective, the previously cited benefits of software design (early assessment of technical feasibility, etc.) are unavailable when there is no approach that maps to and supports the technology used. Traceability to the requirements becomes difficult, with corresponding evolution and reuse challenges.

The primary contribution of this paper is a description of the mapping of the designs of crosscutting concerns to emerging implementation technologies. This is achieved, with varying degrees of success and evolvability, through a mapping of the constructs defined for composition patterns to AspectJ and Hyper/J code. In working through this mapping, we are closer to achieving full traceability of crosscutting requirements throughout the development lifecycle. A secondary contribution is as a case study into applying those technologies. In mapping the design to the code, we discuss the varying degrees of success we experienced in implementing crosscutting behaviour, and consider evolution issues with the approaches.

Section 2 illustrates the design of the Observer pattern [10] using composition patterns. Section 3 maps this design to AspectJ [19, 34], while Section 4 maps the design to Hyper/J [28]. Related work is described in Section 5. Section 6 presents conclusions and further discussion.

2. COMPOSITION PATTERNS

It is the nature of crosscutting behaviour that it has an impact on multiple, different elements within software. In order to design such behaviour in standard UML [23], it is necessary to explicitly specify, using interaction models, crosscutting behaviour against each of the particular elements it may supplement. Though simple templates are available in UML, no composition semantics exist that are sufficient to merge crosscutting behaviour with other behaviour it impacts. These limitations result in design models with a number of difficulties. First, any new element needing to be supplemented with crosscutting behaviour must have a new interaction model defined indicating this. Secondly, changing or eliminating crosscutting behaviour requires changes to all the interaction models specifying it. Finally, reuse of the crosscutting behaviour is not straightforward, as its specification is tangled with the specification of the behaviour it supplements.

Composition patterns mitigate these problems by supporting the *separate* design of reusable, cross-cutting requirements. A crosscutting design within a composition pattern is *independent* of any base design it may potentially crosscut. How that design may be reused where it may be required is also specified—i.e., its pattern of composition.

As described in [7], encapsulation of the design of crosscutting behaviour in a reusable way is achieved using a combination of an extension to UML templates and composition semantics defining how both structural and behavioural design elements may be merged. An inherent requirement of a design approach to specifying crosscutting elements is a need to support reasoning about those elements on which they may have an impact. This is where templates are used. A template parameter in a CP denotes a placeholder element to be replaced by a "real" element in a composed design. In this way, the designer of the crosscutting behaviour may remain oblivious to the real elements that the crosscutting behaviour may impact.

Semantics for the composition of a "base" design with a composition pattern are based on *merge* semantics first introduced in [6], and detailed in [5]. This composition model supports separate design models as independent views of possibly overlapping core concepts. Composition of these separate design models is specified with a *composition relationship*, detailing which elements overlap, and how to integrate them. Merge is one strategy for integration that includes all the elements from the input design models in the composed design, reconciling conflicts where appropriate.

When a template parameter in a composition pattern is an operation, merge semantics uses delegation to ensure the execution of both the crosscutting behaviour and the real operation's behaviour. A composition relationship between a CP and base design(s) defines the elements that replace the template parameters in the CP, thereby specifying how the CP and base design are to be composed (i.e., merged).

Composition patterns are designed to be intuitive to existing UML designers, in that standard UML constructs and notations are reused where possible. Notationally, a UML-style template box is placed on the top-right corner of a CP package, which provides an ordered list of all the templates defined within the CP. A composition relationship is a new kind of relationship, but is defined in a manner similar to each of the existing relationships within the UML. A more complete description of the extensions to the UML metamodel required to support composition semantics and composition patterns (previously introduced in [4], and detailed in [5]) is beyond the scope of this paper.

We now illustrate the design of a reusable CP to support Observer, a base design supporting a small Library, and a specification of how to compose the two.

2.1 Observer Composition Pattern

The Observer pattern describes the collaborative behaviour between a subject and multiple observers. Observer objects register an interest in Subject objects, so that the observers are notified of any change in state in those subjects in which they are interested. From a composition pattern perspective, this requires both structural and behavioural template design elements. We define an Observer CP with two pattern classes (classes that are templates to be replaced by "real" classes during composition with a base design). Subject is defined as a pattern class representing the class of objects whose changes in state are of interest to other objects, and Observer is defined as a pattern class representing the class of objects interested in a Subject's change in state (see Fig. 1).

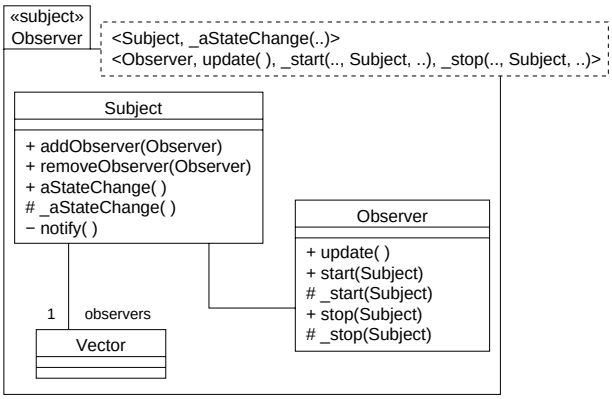

Figure 1: Observer CP Structure

This CP also contains three interaction specifications for behaviour that crosscuts template operations. Fig. 2 illustrates the behaviour required for notifying observers of changes in state. _aStateChange( ) is a template operation whose behaviour is supplemented with notification of all observers. This operation has been prepended with an underscore to denote that delegation is used during merge, and the operation must be replaced by some operation in any class that replaces Subject. notify( ) calls another template operation, update( ), which must be replaced by some operation in any class that replaces Observer. Note that _aStateChange( ) and update( ) appear in the template box in Fig. 1.

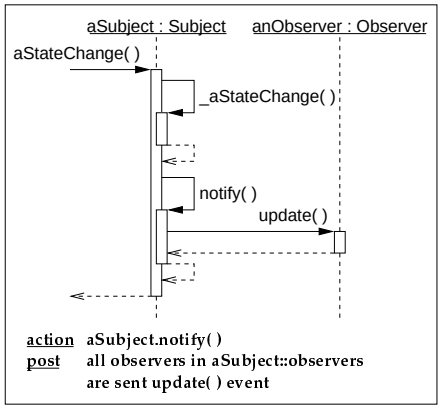

Figure 2: Notifying Observers of State Changes

The Observer CP also supports specification of crosscutting behaviour relating to both initiating and terminating an observer's interest in a subject's changes in state. Two template operations have been defined, _start(.., Subject, ..) and _stop(.., Subject, ..), where each is replaced by operations denoting the start and end, respectively, of an observer's interest in a subject (see Figs. 3 & 4). Each of the replacing operations must have a subject defined as an input parameter.

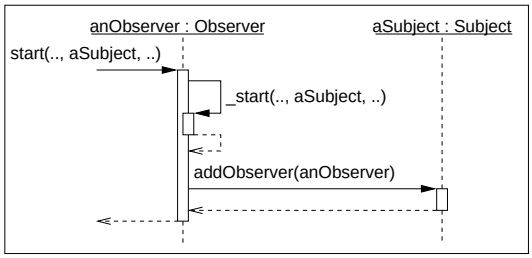

Figure 3: Initiating an Observer's Interest

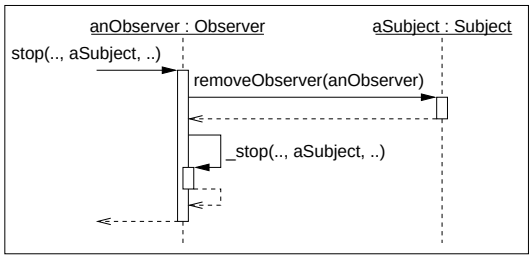

Figure 4: Terminating an Observer's Interest

2.2 Base Library Design

The base design on which the aspect examples are applied is a small library design (Fig. 5). This library has books of which all copies are located in the same room and shelf. A book manager handles the maintenance of the association between books and their locations. The book manager also maintains an up-to-date view of the lending status of book copies.

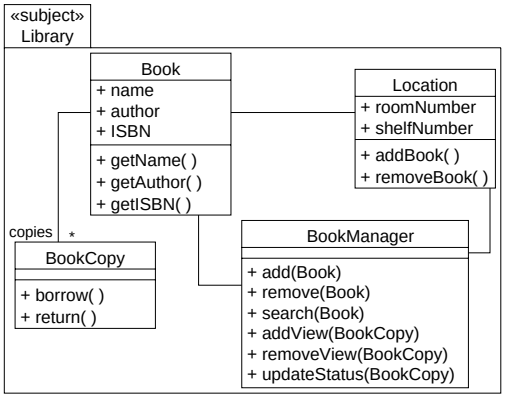

Figure 5: Base Library Design

2.3 Pattern Binding to Base Design

The composition of the Library base design with the Observer composition pattern is specified by a composition relationship between the two. Using a bind[ ] attachment to the relationship, the class(es) acting as subject, and the class(es) acting as observer may be defined. In this example, there is only one of each (see Fig. 6), BookCopy and BookManager, respectively.

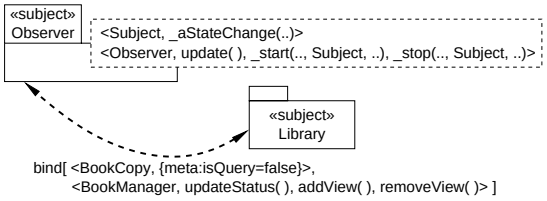

Figure 6: Composing Observer with Library

In this example, note also how the meta-properties of a design's elements may be queried to assess an element's eligibility to join a set of replacing elements. In this example, the _aStateChange( ) template operation is replaced with all operations within BookCopy that have been defined as being non-query—i.e., those operations that affect a change in state that may be of interest to an observer. The keyword meta within the set parameter specification denotes that a UML meta-property is queried, and only those operations with isQuery=false will replace _aStateChange( ) for the purposes of Observer.

3. ASPECTJ

AspectJ [19] is a prototype language to realise the aspect-oriented programming (AOP) paradigm. AOP is a programming technique that makes it possible to express programs involving encapsulated, crosscutting concerns through composition techniques, and through reuse of the crosscutting code [20]. AspectJ comprises a set of extensions to the Java language [11]. We begin with a brief discussion of AspectJ's concepts and constructs, then utilise these in different ways to attempt to map composition patterns to a compositional implementation model.

3.1 Background

The major crosscutting construct in AspectJ is called an *aspect*. Each aspect encapsulates functionality that crosscuts other classes in a system. An aspect is essentially a special form of class: it is instantiated, can contain state and methods, and may be specialised in subaspects. An aspect is then combined with the classes it crosscuts according to specifications given within the aspect. An aspect can *introduce* methods, attributes, and interface implementation declarations into types; as of version 0.8b1, introduced members may be made visible only within the aspect (private introduction), allowing one to avoid name clashes with pre-existing members.

Aside from introductions, the chief handle provided for composing an aspect with other classes is called a *joinpoint*: a joinpoint is a point in the execution of the system, such as a call to a method, the reception of a method call, an access to an attribute, an object creation, etc. Sets of joinpoints may be referred to as *pointcuts*, alluding to the fact that such sets may crosscut the system. Pointcuts can be named, allowing them to be reused. AspectJ provides various pointcut *designators* that may be combined through logical operators to build up complete descriptions of pointcuts of interest; designators include `instanceof(...)`, indicating joinpoints involving instances of the classes in "...," and `receptions(...)`, indicating joinpoints where the method in "..." receives a call. See [34] for a complete listing of possible designators.

An aspect can specify *advice* that is to execute in conjunction with a pointcut. Advice is a block of instructions that is executed `before`, `after`, or `around` a pointcut. `around` advice executes *in place* of the indicated pointcut, allowing a method to be replaced, for example; the replaced pointcut can then be continued with or not within the advice block through a special call to `proceed()`.

Aspects may be declared abstract, making them uninstantiable; by default, a concrete aspect is a Singleton (only one instance exists for the program execution [10]) although other possibilities exist and are examined further in Section 3.2.2.1. Named pointcuts can be declared abstract within an abstract aspect, allowing them to be given concrete definitions within concrete subaspects, much as abstract methods are used.

3.2 Mapping Observer to AspectJ

The question of how to map composition patterns to AspectJ depends on how faithfully one wishes to represent the design-level entities. There are two chief scenarios:

1. represent both a CP and its bind[ ] specification as a single aspect, or
2. maintain the separation of a reusable CP from its bind[ ] specification.

Scenario 1 was the approach demonstrated briefly in [7]. Here, we examine a mapping to AspectJ via each option in turn.

3.2.1 Concrete Aspects Only

As in [7], we map the design subject Observer to a single aspect. For each class being bound to the CP, namely BookManager and BookCopy, we declare introductions for the non-template methods and attributes of their associated template classes, respectively Observer and Subject.

```
aspect Observer {
  // --- Introductions ---
  private Vector BookCopy.observers;

  private void
  BookCopy.addObserver(BookManager bm) {
  }

  private void
  BookCopy.removeObserver(BookManager bm) {
  }

  private void BookCopy.notify() {
    // Post: all observers in
    // BookCopy.observers are sent
    // updateStatus() event
  }

  // --- Pointcuts ---
  pointcut start(BookCopy bc,
                 BookManager bm):
    instanceof(bm) &&
```

```
    receptions(void addView(bc));

  pointcut stop(BookCopy bc,
                BookManager bm):
    instanceof(bm) &&
    receptions(void removeView(bc));

  pointcut aStateChange(BookCopy bc):
    instanceof(bc) &&
    (receptions(void return()) ||
     receptions(void borrow()));

  // --- Advice ---
  after(BookCopy subject,
        BookManager observer):
    start(subject, observer) {
    subject.addObserver(observer);
  }

  before(BookCopy subject,
         BookManager observer):
    stop(subject, observer) {
    subject.removeObserver(observer);
  }

  after(BookCopy subject):
    aStateChange(subject) {
    subject.notify();
  }
}
```

A pointcut is defined for each of the template methods _aStateChange( ), _start( ), and _stop( ). Note that each of these template methods is subject to *merge semantics* (where supplementary functionality is being merged with them) as indicated by the underscore prepending each. Template methods not supplemented with additional behaviour simply have all occurrences replaced with the actual method bound to them (i.e., update( ) is replaced by updateStatus( ) in this example), rather than having a pointcut defined for them.

Each pointcut is defined to represent the joinpoints that are depicted by the initial message received in the interaction diagram associated with each template method supplemented with crosscutting behaviour. Each can simply be mapped to an `instanceof()` designator, indicating the receiving object, and a `receptions()` designator, indicating the method being called. The formal parameters of the pointcut can be determined by looking at the template box specification; for each template operation, the instance of its pattern class and any formal parameters it explicitly declares must be exposed as formals in the pointcut. For example, the pointcut for _stop( ) must declare a formal parameter to represent the instance of `Observer` on which _stop( ) is being called plus another for the argument of type `Subject` that gets passed to it. The isQuery=false constraint in the bind[ ] specification needs to be translated into the actual methods for which this constraint holds (which can be determined from the design of the class being bound), since Java has no support for the UML notion of an isQuery property.

Finally, a piece of advice is declared for each interaction diagram associated with a supplemented template method. For example, after the concrete method bound to _start( ) is received by an instance of Observer (`BookManager`), this instance registers itself as an Observer of the Subject (`BookCopy`) passed as a parameter.

As this example illustrates, by using a combination of the information in a CP with its base design composition and binding specification, mapping to a concrete aspect may be achieved in an algorithmic fashion suitable for automation.

But this mapping is not without its problems. For every bind[ ] specification on a CP in a design, a separate aspect must be created. Each of these aspects contains a portion of the Observer pattern—in other words, the Observer pattern remains crosscutting functionality. As a result, should the details of the Observer pattern need to change, every aspect representing a particular bind[ ] specification would need to be modified. Furthermore, the mapping, while algorithmic, is not simple. Both of these problems suggest that tool support would be required to perform the mappings. Unless the entire implementation were automatically generated from the design-level, such a tool could only produce a skeleton for each aspect that would need to be filled-in after the fact. Thus, regenerating the mappings, should a change to the CP ever be required, would force many aspects skeletons to be filled-in manually again.

3.2.2 Mapping Reusable CPs to Abstract Aspects

The difficulties with evolving mapped CPs would be minimised if we could produce an implementation-level construct that represented a CP alone, without its bind[ ] specification. Then, any changes to this CP would affect only this one construct. This construct should then be more reusable, since it would not be specific to a single bind[ ] specification.

Abstract aspects provide such a means of separating the code for crosscutting behaviour in a reusable way. We therefore assess how a more direct mapping, from CPs to abstract aspects, might be achieved. We look at three possible approaches, varying depending on the number of aspects involved and whether their instances contain state.

3.2.2.1 A Single, Abstract Aspect with State

As a first approach to realising such an implementation mapping to composition patterns without their bind[ ] specifications, we attempt to represent each CP again by a single aspect. Each aspect instance will contain state pertinent to the instance of the Observer pattern that it handles.

Each pattern class within the CP defines an interface within the aspect. These interfaces declare methods for each template method for which no supplementary behaviour has been defined in its associated pattern class, e.g., update( ) in Observer. This interface serves to provide a handle on known operations within the scope of the abstract class. If no non-supplemented template methods exist for a pattern class (as is the case for Subject), we do not need to define an interface for it. All non-template methods and attributes are added as instance members of the aspect itself.

```
abstract aspect Observer {
  // --- Type declarations ---
  interface ObserverI {
    public void update();
  }

  // --- Aspect instance state ---
  Vector observers;

  // --- Aspect instance methods ---
  void notify() {
    // Post: all observers in observers
    // are sent update() event
  }
```

```
  void addObserver(ObserverI observer) {
  }

  void removeObserver(ObserverI observer){
  }

  // --- Pointcuts ---
  abstract pointcut aStateChange();

  abstract pointcut
    start(ObserverI observer);

  abstract pointcut
    stop(ObserverI observer);

  // --- Advice ---
  after(ObserverI observer):
    start(observer) {
    addObserver(observer);
  }

  after(ObserverI observer):
    stop(observer) {
    removeObserver(observer);
  }

  after(): aStateChange() {
    notify();
  }
}
```

As before, a pointcut is declared for each behaviourally supplemented template method, although each is made abstract in this scenario. Each pointcut is given a concrete definition when the CP is bound to actual classes. Finally, advice that is analogous to that described in the first scenario is declared here.

To bind a CP to concrete classes, we declare a concrete aspect that extends the abstract `Observer` aspect. Binding this aspect to `BookCopy` and `BookManager` yields the concrete aspect below.

Any concrete class that is bound to a pattern class, for which an interface was declared in the abstract aspect representing the CP, must receive an introduction that it implements that interface. In addition, an implementation must be provided for each operation declared in that interface. The implementation of each such method delegates to the existing method that has been bound to the associated non-supplemented template method. For example, the Observer CP defines a non-supplemented update( ) template method for the Observer template class; since the updateStatus( ) method of BookManager gets bound to update( ), `BookManager` must define `update()` to delegate to `updateStatus()`.

The concrete aspect must also give each abstract pointcut that it inherits a concrete definition. This is done identically to the case in Section 3.2.1.

```
aspect ObserverBookCopyBookManager
  extends Observer of <context> {
  // --- Introductions ---
  BookManager +implements ObserverI;

  public void BookManager.update() {
    updateStatus();
  }

  // --- Pointcuts ---
  pointcut start(BookCopy bc,
                 BookManager bm):
    instanceof(bm) &&
    receptions(void addView(bc));

  pointcut stop(BookCopy bc,
                BookManager bm):
    instanceof(bm) &&
    receptions(void removeView(bc));

  pointcut aStateChange(BookCopy bc):
    instanceof(bc) &&
    (receptions(void borrow()) ||
     receptions(void return()));
}
```

But there is one piece missing from the puzzle: what should `<context>` be? AspectJ uses this declaration for two purposes: to decide where aspect instances should be created and in what part of the system's execution (called the *execution context*) aspect instance state should be accessible—the two are not separable here. This is a problem. There are only three varieties of `<context>` available in AspectJ:

1. `eachJVM()`, which produces a singleton instance for the entire execution[1];
2. `eachobject(...)`, where an instance is created for each instance of "..." ; and
3. `eachcflowroot(...)`, where an instance is temporarily created for a portion of the execution while "..." is on the call stack.

The intent with the design is to create one aspect instance for each observed `BookCopy`; this can be roughly achieved by creating one aspect instance for *every* instance of `BookCopy`. But this would mean that the aspect instance state would only be available within the execution context of methods defined in `BookCopy`, i.e., only while a method in BookCopy was on top of the execution stack. But, by definition, the execution of `addView()` or `removeView()` will violate this constraint (being methods in `BookManager` and not `BookCopy`), and so, the `start()` and `stop()` pointcuts will never occur.

To take this approach of having an abstract aspect represent a CP without a bind[ ] specification, we would need to be able to separate the mechanisms of specifying the execution context from the specification of what aspect instance to retrieve in that context. This would require modifications to AspectJ.

3.2.2.2 Two Abstract Aspects

Our second approach requires two separate, interacting aspects, one per template class defined in the CP. The chief difference here is that the `Observer` aspect instances must explicitly locate the `Subject` aspect instance associated with the object to be observed. Each `Observer` aspect instance must also record the concrete instance with which it is associated.

[1] As of version 0.8b1, `eachJVM()` is the default context for concrete aspects, and so the `of`-clause may be elided.

```
abstract aspect Observer {
  // --- Aspect instance state ---
  protected Object observer;

  // --- Aspect instance methods ---
  abstract void update();

  abstract Subject
    getSubjectAspect(Object subject);

  // --- Pointcuts ---
  abstract pointcut
  start(Object subject,
        Object observer);

  abstract pointcut stop(Object subject);

  // --- Advice ---
  after(Object subject, Object observer):
    start(subject, observer) {
    Subject s = getSubjectAspect(subject);
    s.addObserver(this);
    this.observer = observer;
  }

  before(Object subject): stop(subject) {
    Subject s = getSubjectAspect(subject);
    s.removeObserver(this);
  }
}

abstract aspect Subject {
  // --- Aspect instance state ---
  Vector observers;

  // --- Aspect instance methods ---
  void notify() {
    // Post: all observers in observers
    // are sent update() event
  }

  void addObserver(Observer observer) {
  }

  void removeObserver(Observer observer) {
  }


  // --- Pointcuts ---
  abstract pointcut aStateChange();

  // --- Advice ---
  after(): aStateChange() {
    notify();
  }
}
```

And now, the concrete aspects become:

```
aspect ObserverBookManager
  extends Observer
  of eachobject(instanceof(BookManager)) {
  // --- Aspect instance methods ---
  void update() {
    ((BookManager)observer).
      updateStatus();
  }

  Subject getSubjectAspect(Object subject) {
    return SubjectBookCopy.
      aspectOf(subject);
  }

  // --- Pointcuts ---
  pointcut start(BookCopy bc):
    receptions(void addView(bc));

  pointcut stop(BookCopy bc):
    receptions(void removeView(bc));
}


aspect SubjectBookCopy
  extends Subject
  of eachobject(instanceof(BookCopy)) {

  // --- Pointcuts ---
  pointcut aStateChange():
    (receptions(void borrow()) ||
     receptions(void return()));
}
```

There are still problems here, though. First, the Observer pattern is conceptually a single aspect, so splitting it into multiple constructs is unnatural—a crosscutting concern persists in a scattered form. Each `Observer` instance assumes that it is associated with a single object, but cannot enforce this constraint. The concrete observer needs to know about particular concrete subjects, since `aspectOf()` is only defined for concrete aspects—AspectJ assumes that there is at most one instance of a concrete aspect associated with an object. We also end up with an extra object for each subject and each observer even if they are not actually doing any observing or being observed.

This mapping to an abstract aspect plus extending, concrete aspects is more complicated, and hence error-prone, than in the first approach. Our concerns over the reusability of the implementation-level CPs have not been completely alleviated. It is up to the application programmer to correctly define the concrete pointcuts in such a way as to fulfill the behavioural constraints implied by the Observer pattern; it is not clear that this will always be as straightforward a process as filling in template parameters in CPs is.

3.2.2.3 A Single, Abstract Aspect without State

Our third and final approach forgoes any attempt to maintain state within aspect instances themselves. This is only possible since the Observer pattern explicitly accounts for each Subject possessing multiple Observers. The state involved in the pattern can therefore be divided on an individual-object basis, rather than an aspect instance having to maintain crosscutting state. If this were not the case, we could still encounter the problems of `of`-clauses discussed in Section 3.2.2.1.

In our abstract aspect, we declare one interface for each pattern class that exists in the CP being mapped. These interfaces declare any non-supplemented template methods within the CP. Any non-template methods or attributes on these pattern classes are introduced onto the corresponding interface. An introduction on an interface has the effect of adding that method or attribute into all concrete classes that implement the interface. Finally, our pointcuts and advice remain the same as in Section 3.2.2.1 with one important difference: they must refer to the `SubjectI` instance that is involved in the instance of the Observer pattern. In the first approach, this was not necessary since this instance did not contain the state and methods that maintained the Observer pattern behaviour.

```
abstract aspect Observer {
  // --- Type declarations ---
  interface SubjectI {
  }

  interface ObserverI {
    public void update();
  }

  // --- Introductions ---
  private Vector SubjectI.observers;

  private void SubjectI.notify() {
    // Post: all observers in SubjectI.observers
    // are sent update() event
  }

  private void
  SubjectI.addObserver(ObserverI observer) {
  }

  private void
  SubjectI.removeObserver(ObserverI observer) {
  }

  // --- Pointcuts ---
  abstract pointcut
    aStateChange(SubjectI subject);

  abstract pointcut
    start(SubjectI subject, ObserverI observer);

  abstract pointcut
    stop(SubjectI subject, ObserverI observer);

  // --- Advice ---
  after(SubjectI subject, ObserverI observer):
    start(subject, observer) {
    subject.addObserver(observer);
  }

  after(SubjectI subject, ObserverI observer):
    stop(subject, observer) {
    subject.removeObserver(observer);
  }

  after(SubjectI subject):
    aStateChange(subject) {
    subject.notify();
  }
}
```

The bind[ ] specification on the Library base design is then represented by a single concrete aspect. Each concrete class being bound to a pattern class requires an introduction that it implements the interface corresponding to that pattern class; for example, since BookCopy is bound to Subject, `BookCopy` must implement `SubjectI`. Any non-supplemented template methods that were declared by these interfaces must have concrete implementations introduced for them; since BookManager::updateStatus( ) is bound to Observer::update( ), `BookManager.update()` must be introduced and it must delegate to `updateStatus()`. The inherited, abstract pointcuts must have concrete definitions provided, as in the earlier approaches.

```
aspect ObserverBookCopyBookManager
  extends Observer {
  // --- Introductions ---
  BookCopy +implements SubjectI;
  BookManager +implements ObserverI;

  public void BookManager.update() {
    updateStatus();
  }

  // --- Pointcuts ---
  pointcut start(BookCopy bc, BookManager bm):
    instanceof(bm) &&
    receptions(void addView(bc));

  pointcut stop(BookCopy bc, BookManager bm):
    instanceof(bm) &&
    receptions(void removeView(bc));

  pointcut aStateChange(BookCopy bc):
    instanceof(bc) &&
    (receptions(void borrow()) ||
     receptions(void return()));
}
```

This approach appears the best candidate given the current semantics of AspectJ, but there remain potential problems.

The introduction of an interface that declares methods can lead to name clashes. In the example above, if `BookManager` had already declared an `update()` method, but this method did not fulfill the purposes of update( ) within the CP, invasive modifications would have been required to resolve the conflict. AspectJ does not currently provide a reconciliation mechanism between differing views, such as those of composition filters [1], subject-oriented programming [13], or implicit context [32].

The form of the concrete aspect still puts a large onus on the user of the CP in the absence of tool support, not only to identify which design elements should be bound to template parameters, but to correctly designate pointcuts and introductions. The result could be more flexible than pluggable templates, but also more error prone.

In any situation where the aspect instance must maintain state, but its execution context differs from the context in which it must be instantiated, the troubles with `of`-clauses discussed above would crop up. The question is, will such situations ever actually occur? Regardless, AspectJ still does not support CPs as cleanly as we would like.

4. HYPER/J

Hyper/J [28] is a prototype language to realise the multi-dimensional separation of concerns (MDSOC) paradigm [29]. MDSOC is a modelling and implementation paradigm that supports the separation of overlapping concerns along multiple dimensions of composition and decomposition [29]. Hyper/J is a programming environment that facilitates the adaptation, composition, integration, improved modularisation, and non-invasive remodularisation of Java software components [28]. This section gives a brief introduction to the concepts and inputs to Hyper/J, and demonstrates these inputs for the Observer CP.

4.1 Background

Unlike AspectJ, Hyper/J does not have constructs whose instances appear at runtime. Hyper/J works with Java `.class` files, supporting sophisticated reasoning about their modularisation (and remodularisation), and composition. In other words, you may describe the internals of Java `.class` files, and describe how you would like this code to be integrated differently. Hyper/J produces new

Java `.class` files, where structure and behaviour of (parts of) input `.class` files are integrated as defined by the programmer.

There are three main inputs the developer provides when using Hyper/J [28].

1. A *hyperspace* file describes the Java `.class` file being composed. Here, selection of classes to be composed (and by implication, classes *not* be to composed) is specified.
2. A *concern mapping* file describes the pieces of Java within those files that map to different concerns of interest.
3. A *hypermodule* file describes how integration between concerns of interest should be done. Here, different kinds of composition strategies may be specified (e.g., merge or override), with the possibility of defining a *match* relationship for method invocations so that invocation of one results in the invocation of all matched methods. Most interestingly for crosscutting concerns, the notion of a *bracket* relationship supports the specification of which methods should be executed before and/or after a method to be crosscut with additional behaviour.

CPs, with their inherent merge semantics, evolved from ideas within subject-oriented programming [13, 24], as did MDSOC and Hyper/J. As such, at a high-level, there should be a more direct map from CPs to the inputs of Hyper/J than was demonstrated with AspectJ.

In our attempt to map CPs to Hyper/J, we have chosen to consider Hyper/J in terms of the full specification of its potential as defined in [28], and not its more limited implementation in the currently available version of the Hyper/J tool.

4.2 Mapping Observer to Hyper/J

The internals of the Observer CP, and the Library base design may be described using hyperspace and concern mapping files, while the bind[ ] specification of the composition relationship may be mapped to the hypermodule file. However, at a more detailed level, mapping becomes more difficult, as we shall see.

First, we look at the Java source code implementing the classes defined in the Observer composition pattern. `Subject` and `Observer` classes are defined in an `Observer` package.

```
class Subject {
  Vector observers;

  void addObserver(Observer observer) {
  }

  void removeObserver(Observer observer){
  }

  void aStateChange() {
    notify();
  }

  void notify() {
    // All observers in observers are
    // sent update() event
  }
}
```

```
class Observer {
  void update() {
  }

  void start(Subject subject) {
    subject.addObserver(this);
  }

  void stop(Subject subject) {
    subject.removeObserver(this);
  }
}
```

Code supporting the Library design model is not illustrated here, though we assume it to be defined within a `Library` package. Each of these packages are considered to be in the space within which we are working, and are defined in a hyperspace file:

```
hyperspace ObservedLibrary
  composable class Observer.*;
  composable class Library.*;
```

Concern mappings may be defined as:

```
package Observer : Feature Observer
package Library  : Feature Library
```

However, this mapping of the reusable Observer CP to code is not as straightforward as it may appear. Hyper/J imposes a restriction that operations to be merged must have the same signature. CPs support a mechanism for specifying considerable flexibility in the signatures of operations that are allowed to replace template operations. For our Observer example, the template operations _start(.., Subject, ..) and _stop(.., Subject, ..) specify that one of the parameters must be an object of type Subject, but that there may be any other parameters. This flexibility does not map to Hyper/J. The `Observer` class illustrated here has defined a single `Subject` parameter for both the `start()` and `stop()` methods. This mapping could only occur after examining the signatures of the replacing operations, as defined in the bind[ ] attachment to the composition relationship. The signatures of the template operations in the `Observer` class were then defined appropriately. Clearly therefore, the `Observer` package is not reusable as currently defined. Prior to being merged with any other package, the signatures of all methods with which `start()` and `stop()` are to be merged must be examined, with overloaded methods defined for any methods with differing signatures.[2]

We now look at the hypermodule file, which specifies how the packages should be integrated.[3] The concern mapping identified two features, `Library` and `Observer`, to be composed. A `nonCorrespondingMerge` relationship is defined between the two features, indicating that any elements with the same name in the different features do not correspond, and are not to be merged. This is chosen because the correspondences between the Observer pattern and elements within any potential hyperslice with which it

[2] It is not clear that Hyper/J's bracket declaration would correctly handle overloaded methods; if not, method renaming would be required to differentiate between them.

[3] The `nonCorrespondingMerge` and `override` relationships are not currently enabled in the Hyper/J tool, and so, this code has not been compiled.

is to be merged are explicitly defined, and any name matching otherwise is coincidental.

The replacement of the Observer and Subject pattern classes with BookManager and BookCopy, respectively, can be mapped directly to `equate` relationships. An `override` relationship may be used to map the replacement of `update()` with the `updateStatus()` method. Each of the methods that are replacements for operations supplemented by crosscutting behaviour have a `bracket` relationship defined to specify the invocation of the appropriate methods before or after their own execution. This interactive behaviour is gleaned from the interactions within the CP itself, not the composition relationship. One point of note: the bind[ ] attachment to the composition relationship supports reasoning about the meta-properties of operations—in this example, any operations whose isQuery property is false replace the _aStateChange( ) template operation (see Fig. 6). Since there is no equivalent specification in Hyper/J, the mapping process must examine each of the operations in BookCopy, and add a `bracket` relationship for any operation that passes the isQuery test—borrow( ) and return( ) in this case.

```
hypermodule ObserverLibrary
  hyperslices:
    Feature.Library,
    Feature.Observer;

  relationships:
    nonCorrespondingMerge;

  equate class
    Feature.Library.BookManager
    Feature.Observer.Observer;

  equate class
    Feature.Library.BookCopy,
    Feature.Observer.Subject;

  override action
    Feature.Observer.Observer.update
      with Feature.Library.BookManager.
           updateStatus;

  bracket "addView" with
    (after Feature.Observer.Observer.start,
           "BookManager");

  bracket "removeView" with
    (before Feature.Observer.Observer.stop,
            "BookManager");

  bracket "borrow" with
    (after
       Feature.Observer.Subject.
         aStateChange, "BookCopy");

  bracket "return" with
    (after
       Feature.Observer.Subject.
         aStateChange, "BookCopy");
end hypermodule
```

As we can see, the hypermodule file specifying how to integrate the `Library` and `Observer` features has the potential to provide a clean mapping from CPs with simple interactions specified. However, though not illustrated with the Observer example, limitations with the `bracket` relationship, as currently defined, may present difficulties for more complicated interactions in the design.

5. RELATED WORK

While we have examined the mapping from a particular compositional-design mechanism (composition patterns) to two particular compositional-implementation mechanisms (AspectJ and Hyper/J) in this paper, other possibilities abound in many dimensions.

Collaboration-based design or role modelling is a compositional design approach that concentrates on decomposing designs on the basis of the roles that objects play in particular collaborations [3, 14, 17, 25]. For role modelling within OORam in particular [25], the goals are similar to those motivating separation of non-crosscutting concerns in subject-oriented design [5]. Kendall looked at role modelling and how one might map it to AspectJ [18], concluding that AspectJ did not adequately support a required level of composition for roles (e.g. merge or override). Catalysis [9] also supports the decomposition of software designs along ”vertical” and ”horizontal” lines, providing the ability to separate both functional and technical concerns. Subject-oriented design [6, 5] with its composition patterns [7] is a more generic approach, including support for both functional separation (like roles) and separation of patterns of crosscutting behaviour. Thus, this paper is an investigation into how one can map this generic design approach to a compositional implementation.

Others have looked to mixins [30] and mixin layers [26] as a means of realising compositional implementations of collaboration-based designs. Mixin layers are useful for product-line architectures, where features are understood from conception to be optional between different configurations of a product. We have begun a preliminary look at mixin layers, but they appear to be problematic for our purposes: they require adherence to strict class hierarchy constraints that are not easily evolvable or reusable, and they suffer from Decorator pattern [10] drawbacks[4] when applied to evolve existing components.

Other approaches to providing design support for crosscutting concerns appear more firmly rooted in the aspect-oriented programming paradigm exclusively. For example, approaches exist to extend the UML with stereotypes specific to particular crosscutting functionality, such as synchronisation [15] and the Command design pattern [16]. Such approaches, while clearly allowing an easy mapping from design to implementation, place the onus on design and implementation of extensive sets of aspect languages that require knowledge of the specific behaviour to provide; AspectJ itself abandoned this approach for the sake of a more general language at an early stage. Suzuki and Yamamoto [27] have attempted a more generalised way of supporting aspect-oriented programming within the UML, but by tying itself to a particular realisation of a compositional-implementation language, the design language must evolve as rapidly as the particular implementation language, and expresses design concepts only as well as the implementation does—such limitations can be a strength or a weakness in different contexts. Subject-oriented design has taken the more independent route in extending the UML [4] to provide just those constructs required to support the decomposition (and subsequent composition specification) of design models based on requirements specifications. These requirements may be functional or crosscutting, and

[4]These drawbacks include the need to alter all components to use the decorator objects rather than the decorated objects, and the so-called “object schizophrenia” problem, where it is uncertain if a decorated object should or does call its decorated or undecorated self.

new design constructs are focused on how to compose the separate models, *not* on providing constructs to map to any particular implementation paradigm. This approach makes the model more concern centric, not implementation-paradigm centric.

While we have focussed on only AspectJ and Hyper/J in this paper, other compositional implementation mechanisms exist. Composition filters [1] are a means of intercepting and rerouting messages as they arrive at objects; they can be used to separate crosscutting concerns such as synchronisation, and have been described as an aspect-oriented technique [2]. Adaptive software [21] has also been described as a (special case) aspect-oriented technique. It provides a means to separate the algorithms on data from the structure of that data, allowing the structure of the data to change without requiring related changes to the algorithms. Implicit context [32] is a recently introduced structuring mechanism and philosophy concentrating on removing knowledge of the large-scale from smaller-scale components; while there is some relationship between such knowledge and the crosscutting functionality that concerns us in this paper, the two are distinct problems [33].

The importance of separation of concerns to the evolutionary phase of development has been examined lightly in the context of an early version of AspectJ [31]. In that work, evidence was found that having aspects with a weak separation of concerns was actually more detrimental to the evolutionary tasks studied than having a traditional, object-oriented modularity.

There has been some recognition of the need for separating crosscutting concerns throughout the lifecycle. For example, Griss has proposed a development process for e-commerce, component-based product-lines that draws together high-level analysis- and design-composition techniques with supporting implementation-composition techniques [12]. But this process does not advise on how to map the differing constructs within the combination of approaches that may be used. The difficulties reported in re-engineering implementations to take advantage of compositional implementation techniques[5] [22], for which they were not originally designed, highlights the importance of separating crosscutting concerns across the lifecycle. Being forced to manually untangle and unscatter the concerns that were identified was a difficult and error-prone process; if the systems discussed in that work had been designed with their crosscutting concerns separated in the first place, porting the implementations between the different compositional techniques studied would have been more tractable.

6. CONCLUSIONS

We have identified a need for a means to separate crosscutting concerns seamlessly across the lifecycle, a need to which existing work points as well. Such an approach would help realise the benefits of software design by supporting early technical assessment of crosscutting behaviour and the evolution and non-invasive addition of such behaviour to the software artefacts across the lifecycle—e.g., designs and code. To investigate current possibilities to support this need, this paper worked with composition patterns at the design level, and with AspectJ and Hyper/J at the code level.

With the Observer pattern example, composition patterns demonstrated a level of encapsulation, independence and reusability for the designs of crosscutting concerns. Since a composition pattern encapsulates details within it, these details can be altered while the concrete classes bound to the CP remain untouched. Thus, a CP serves as a reusable and evolvable design construct, and is therefore a good candidate for design phase separation of crosscutting concerns. From this design base, the paper then investigated a map from CP constructs to compositional implementation models.

The Observer composition pattern provides a good overview of the constructs and concepts that have been added to standard UML to support composition patterns. Existing, standard UML is, of course, also available to composition pattern designers within the composition pattern package. For example, as with all interaction diagrams, constraints may be defined on the execution of operations. Such constraints may have an impact on the execution of crosscutting behaviour that would need to be mapped to the implementation. These have the potential to map to the `around` advice construct in AspectJ, but there is no equivalent in Hyper/J. While a complete mapping from UML to AspectJ and Hyper/J is beyond the scope of this paper, we have captured the essentials of the extensions to the standard object-oriented paradigm, and illustrated a mapping for those.

AspectJ, as currently defined in version 0.8b1, does not preserve the reusability and evolvability inherent in CPs as well as we would like, in part due to difficulties with its `of`-clause construct. As a result, the crosscutting functionality defined in a CP remains scattered and tangled in the aspects that are generated from the mapping. An approach which appears to alleviate this using aspects without state may have other difficulties relating to reconciliation between conflicting methods. The reusability and evolvability of aspects representing CPs are also potentially decreased by the onus on framework users to correctly designate pointcuts.

Based on the plans for Hyper/J as defined in [28], there is potential for a relatively clean mapping from simple CPs to Hyper/J code. However, the restriction that only methods with the same signature may be merged could present difficulties. Overcoming the difficulties with overloaded methods reduces the reusability and extensibility of the code. It is also probable that the `bracket` relationship, with `before` and `after` bracketing only, will not be sufficiently powerful to capture complex interactions specified in the design. In addition, we refer to the mapping in this paper as having only potential, as it will be necessary to implement the mappings to a version of Hyper/J that contains the required relationships.

While tool support may alleviate these difficulties to some extent, we believe working towards reducing any inherent mismatch between the reusable, extensible design capabilities of CPs and the constructs within AspectJ and Hyper/J is preferable. In doing this, we would be closer to achieving a seamless across-the-lifecycle encapsulation of the software artefacts associated with a crosscutting requirement. We would then be closer to achieving the benefits of separation of concerns for both designers and coders even when those concerns are crosscutting.

7. REFERENCES

[1] Mehmet Akşit, Lodewijk Bergmans, and Sinan Vural. An object-oriented language-database integration model: The composition-filters approach. In Ole Lehrmann Madsen, editor, *ECOOP '92: European Conference on Object-Oriented Programming*, volume 615 of *Lecture Notes in Computer Science*, pages 372–395, Utrecht, The

[5] AspectJ, Hyper/J, and a lightweight separation of concerns mechanism that did not utilise tool support were the compositional techniques studied. These were applied to an FTP system (`jFTPd`), and a regular expression matcher (`gnu.regexp`) written in Java.

Netherlands, 29 June–3 July 1992.

[2] Mehmet Akşit and Bedir Tekinerdogan. Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters. Position paper for the Aspect-Oriented Programming Workshop, 12th European Conference on Object-Oriented Programming, 21 July 1998.

[3] Kent Beck and Ward Cunningham. A laboratory for teaching object-oriented thinking. In Norman Meyrowitz, editor, *OOPSLA'89 Conference Proceedings: Object-Oriented Programming: Systems, Languages, and Applications*, pages 1–6, New Orleans, USA, 1–6 October 1989.

[4] Siobhán Clarke. Composing design models: An extension to the UML. In Andy Evans, Stuart Kent, and Bran Selic, editors, *Proceedings of the 3rd International Conference on the Unified Modeling Language*, pages 338–352, York, UK, 2–6 October 2000.

[5] Siobhán Clarke. *Composition of Object-Oriented Software Design Models*. PhD thesis, Dublin City University, Dublin, Ireland, January 2001.

[6] Siobhán Clarke, William Harrison, Harold Ossher, and Peri Tarr. Subject-oriented design: Towards improved alignment of requirements, design and code. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 325–339, Denver, USA, 1–5 November 1999.

[7] Siobhán Clarke and Robert J. Walker. Composition patterns: An approach to designing reusable aspects. In *Proceedings of the 23rd International Conference on Software Engineering*, Toronto, Canada, 12–19 May 2001. To appear.

[8] Steve Cook and John Daniels. *Designing with Objects: Object-Oriented Modelling with Syntropy*. Prentice-Hall, Englewood Cliffs, USA, 1993.

[9] Desmond F. D'Souza and Alan Cameron Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach*. Object Technology Series. Addison-Wesley, Reading, USA, 1998.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, USA, 1994.

[11] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java™ Language Specification*. The Java Series. Addison-Wesley, Reading, USA, second edition, 2000.

[12] Martin L. Griss. Implementing product-line features by composing component aspects. In *Proceedings of the 1st International Software Product Line Conference*, pages 271–288, Denver, USA, 28–31 August 2000.

[13] William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 411–428, Washington, USA, 26 September–1 October 1993.

[14] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In Norman Meyrowitz, editor, *OOPSLA/ECOOP '90 Proceedings*, pages 169–180, Ottawa, Canada, 21–25 October 1990. ACM SIGPLAN.

[15] José Luis Herrero, Fernando Sánchez, Fabiola Lucio, and Miguel Toro. Introducing separation of aspects at design time. Position paper for the Aspects and Dimensions of Concerns Workshop, 14th European Conference on Object-Oriented Programming, 11–12 June 2000.

[16] Wai-Ming Ho, François Pennaneac'h, Jean-Marc Jézéquel, and Noël Plouzeau. Aspect-oriented design with the UML. Position paper for the Multi-Dimensional Separation of Concerns in Software Engineering Workshop, 22nd International Conference on Software Engineering, 4 June 2000.

[17] Ian M. Holland. Specifying reusable components using Contracts. In Ole Lehrmann Madsen, editor, *ECOOP '92: European Conference on Object-Oriented Programming*, volume 615 of *Lecture Notes in Computer Science*, pages 287–308, Utrecht, The Netherlands, 29 June–3 July 1992.

[18] Elizabeth A. Kendall. Role model designs and implementations with aspect-oriented programming. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 353–369, Denver, USA, 1–5 November 1999.

[19] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming*, Budapest, Hungary, 18–22 June 2001. To appear.

[20] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, 9–13 June 1997.

[21] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, USA, 1996.

[22] Gail C. Murphy, Albert Lai, Robert J. Walker, and Martin P. Robillard. Separating features in source code: An exploratory study. In *Proceedings of the 23rd International Conference on Software Engineering*, Toronto, Canada, 12–19 May 2001. To appear.

[23] Object Management Group. *The Unified Modeling Language Specification, Version 1.3*, 1999.

[24] Harold Ossher, Matthew Kaplan, Alexander Katz, William Harrison, and Vincent Kruskal. Specifying subject-oriented composition. *Theory and Practice of Object Systems*, 2(3):179–202, 1996.

[25] Trygve Reenskaug, Per Wold, and Odd Arild Lehne. *Working with Objects: The OORam Software Engineering Method*. Manning Publications Co., Greenwich, USA, 1995.

[26] Yannis Smaragdakis and Don Batory. Implementing layered designs with mixin layers. In Eric Jul, editor, *ECOOP '98—Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 550–570, Brussels, Belgium, 20–24 July 1998.

[27] Junichi Suzuki and Yoshikazu Yamamoto. Extending UML with apsects: Aspect support in the design phase. Position paper for the Aspect-Oriented Programming Workshop, 13th European Conference on Object-Oriented Programming, 14–18 June 1999.

[28] Peri Tarr and Harold Ossher. *Hyper/J User and Installation Manual*. IBM Research, 2000.

[29] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. *N* degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 107–119, Los Angeles, USA, 16–22 May 1999.

[30] Michael VanHilst and David Notkin. Using role components to implement collaboration-based designs. In *OOPSLA '96 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*, pages 359–369, San Jose, USA, 6–10 October 1996.

[31] Robert J. Walker, Elisa L. A. Baniassad, and Gail C. Murphy. An initial assessment of aspect-oriented programming. In *Proceedings of the 21st International Conference on Software Engineering*, pages 120–130, Los Angeles, USA, 16–22 May 1999.

[32] Robert J. Walker and Gail C. Murphy. Implicit context: Easing software evolution and reuse. In David S. Rosenblum, editor, *Proceedings of the ACM SIGSOFT Eighth International Symposium on the Foundations of Software Engineering (FSE-8)*, pages 69–78, San Diego, USA, 8–10 November 2000.

[33] Robert J. Walker and Gail C. Murphy. Joinpoints as ordered events: Towards applying implicit context to aspect-orientation. Position paper for the Workshop on Advanced Separation of Concerns in Software Engineering, 23rd International Conference on Software Engineering, 15 May 2001.

[34] Xerox Corporation. Aspectj 0.8b1 design notes. `http://www.aspectj.org/`, 2001.