Mapping 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

ABSTRACT

In earlier work, we demonstrated the successful separation of the designs of crosscutting concerns into *composition patterns* [1]. In this paper, we demonstrate the mapping of crosscutting designs to two programming models that support similar approaches to separation within code—AspectJ [7] and Hyper/J [5]. We first illustrate the design of the Observer pattern using the composition pattern approach, and then map that design to the appropriate code.

1 INTRODUCTION

Requirements that have a crosscutting impact on software elements present well-documented difficulties for software development [3, 4, 5, 7]. The difficulties 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.

Composition patterns (CPs) present 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 to crosscutting requirements' specifications is achieved. In order to maintain the traceability from requirements to code, it is necessary to be able to map CP designs to code. Achieving such a feat would support the encapsulation of all the software artefacts associated with a crosscutting requirement into a truly reusable package.

In this paper, we demonstrate how design constructs from composition patterns may be mapped to different programming paradigms, with varying degrees of success and evolvability. Section 2 illustrates the design of the Observer pattern [2] using composition patterns. Section 3 maps this design to AspectJ, while Section 4 maps the design to Hyper/J. Section 5 presents conclusions and further discussion.

2 COMPOSITION PATTERNS

As described in [1], 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 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. Where a template parameter is an operation, merge semantics uses delegation to ensure the execution of both the crosscutting behaviour and the real operation's behaviour. Notationally, a UML-style template box is placed on the top-right corner of a CP package, which lists all the templates defined within the CP. The means to specify how to compose the CP with base design(s) is provided by a *composition relationship*. A composition relationship between a CP and base design(s) defines the elements that replace the templates in the CP, thereby specifying how the CP and base design are to be composed (or *merged*).

In this section, we 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.

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).

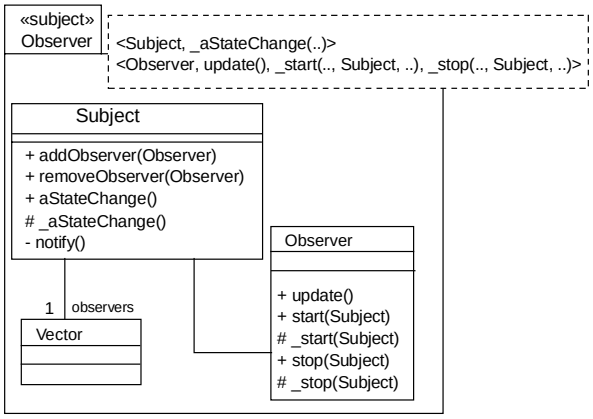

Fig 1: Observer CP Structure

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 semantics apply, and 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.

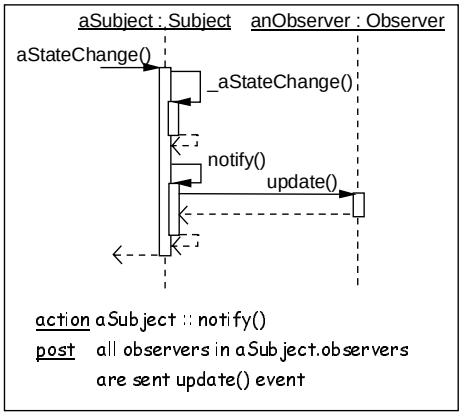

Fig 2: Notifying Observers of State Changes

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.

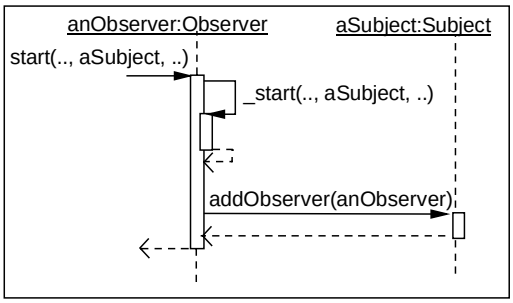

Fig 3: Initiating an Observer's Interest

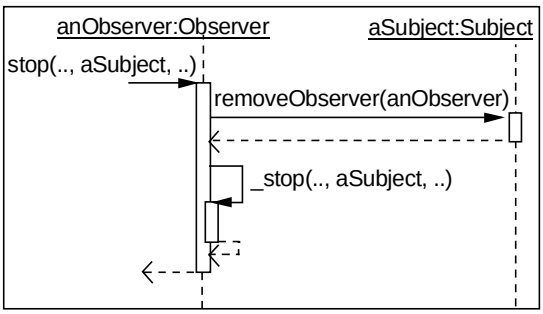

Fig 4: Terminating an Observer's Interest

Base Library Design

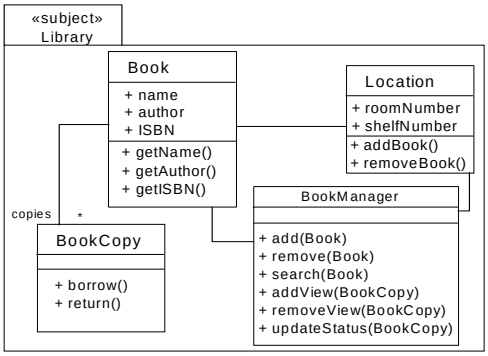

Fig. 5: Base Library Design

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.

Pattern Binding to Base Design

The composition of the Library and Observer composition patterns 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.

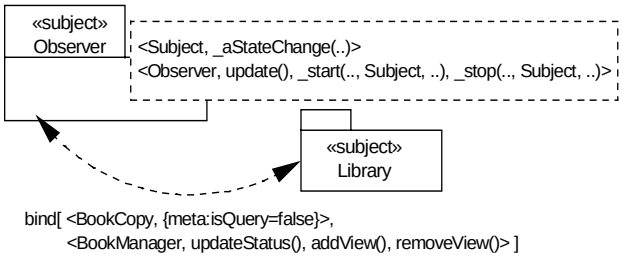

Fig 6: Composing Observer with Library

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 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 binding specification as a single aspect.
2. Maintain separation of a re-usable CP from its binding specification by mapping the CP to an abstract aspect, and the information from the bind[ ] specification with a base design to a concrete aspect that extends the abstract aspect.

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

Concrete Aspects Only

As in [1], we map the design subject Observer to a single aspect. We declare an `introduction` for each class being bound to the CP, namely BookManager and BookCopy, and declare the non-template methods and attributes of their associated template classes, respectively Observer and Subject, within each `introduction`.

```
aspect Observer {
  introduction BookManager {
  }

  introduction BookCopy {
    Vector observers;

    void addObserver(BookManager bm) {
    }

    void removeObserver(BookManager bm) {
    }

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

  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()));

  static after(BookCopy subject,
               BookManager observer):
    start(subject, observer) {
    subject.addObserver(observer);
  }

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

  static 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.

Mapping Reusable CPs to Abstract Aspects

The difficulties with evolving the 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.

As a first attempt at realising such an implementation mapping to composition patterns without their bind[ ] specifications, we attempt to represent each CP again by a single aspect.

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 their 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 {
  interface ObserverI {
    void update();
  }

  Vector observers = new Vector();

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

  void addObserver(ObserverI observer) {
  }

  void removeObserver(ObserverI observer){
  }

  abstract pointcut aStateChange();
  abstract pointcut
    start(ObserverI observer);
  abstract pointcut
    stop(ObserverI observer);
```

```
  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 and 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 as in the first scenario.

```
aspect ObserverBookCopyBookManager
  extends Observer of <context> {
  introduction BookManager {
    implements ObserverI;
    void update() {
      updateStatus();
    }
  }

  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;
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.

Instead, our second attempt 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.

```
abstract aspect Observer {
  protected Object observer;

  abstract void update();
  abstract Subject
```

```
    getSubjectAspect(Object subject);

  abstract pointcut start(Object subject,
                          Object observer);
  abstract pointcut stop(Object subject);

  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 {
  Vector observers = new Vector();
  abstract pointcut aStateChange();

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

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

  void addObserver(Observer observer) {
  }

  void removeObserver(Observer observer) {
  }
}
```

And now, the concrete aspects become:

```
aspect ObserverBookManager
  extends Observer
  of eachobject(instanceof(BookManager)) {
  void update() {
    ((BookManager)observer).
      updateStatus();
  }

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

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

aspect SubjectBookCopy
  extends Subject
  of eachobject(instanceof(BookCopy)) {
  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. Each `Observer` instance assumes that it is associated with a single object, but cannot enforce this constraint (although planned changes to AspectJ may help this situation). 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 and extending, concrete aspects is more complicated, and hence error-prone, than in the first scenario. 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. Regardless, AspectJ does not support CPs as cleanly as we would like.

4 MAPPING OBSERVER TO HYPER/J

There are three main inputs the developer provides when using Hyper/J. A *hyperspace* file and a *concern mapping* file describe the Java class files being composed, and the pieces of Java within those files that map to different concerns of interest. A *hypermodule* file describes how integration between concerns of interest should be done. CPs, with their inherent merge semantics, evolved from ideas within subject-oriented programming [3, 5], as did the implementation of Hyper/J. As such, at a high-level, there is a more direct map from CPs to the inputs of Hyper/J than was demonstrated with AspectJ.[1] 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.

[1] In AspectJ, there were different possible approaches based on the possibilities associated with using a combination of abstract and concrete aspects, or just concrete aspects.

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 [6], and not its more limited implementation in the current version of the Hyper/J tool.

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 `nonCorrespond-ingMerge` 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 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

[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.

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. It is possible to define constraints on the execution of operations within an interaction in UML. These have the potential to map to the `around` advice code in AspectJ, but there is no equivalent in Hyper/J.

5 CONCLUSIONS

Designs of crosscutting concerns using composition patterns are readily reusable. Since a composition pattern encapsulates details within it, these details can be altered while the concrete classes bound to the CP remain untouched. Thus, CPs serve as reusable and evolvable design constructs.

AspectJ, as currently defined in version 0.7b12, does not preserve the reusability and evolvability inherent in CPs well, largely 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.

Based on the plans for Hyper/J as defined in [6], 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 re-usability and extensibility of the code. We also refer to this mapping as only having 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 the difficulties to some extent, we believe we should work towards reducing any inherent mismatch between the reusable, extensible design capabilities of CPs, and the constructs within AspectJ and Hyper/J. In doing this, we would be closer to achieving an across-the-lifecycle encapsulation of the software artefacts associated with a crosscutting requirement into a reusable package.

REFERENCES

1. S. Clarke and R. Walker. “Composition Patterns: An Approach to Designing Reusable Aspects.” To appear, in *Proc. ICSE*, 2001.

2. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable, Object-Oriented Software*, Addison-Wesley, 1994.

3. W. Harrison and H. Ossher. “Subject-Oriented Programming (a critique of pure objects).” In *Proc. OOPSLA*, pp. 411–428, 1993.

4. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. “Aspect-Oriented Programming.” In *Proc. ECOOP*, vol. 1241 of *LNCS*, pp. 220–242, 1997.

5. H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal. "Specifying Subject-Oriented Composition." In *Theory and Practice of Object Systems*, **2**(3):179–202, 1996.

6. P. Tarr and H. Ossher. *Hyper/J User and Installation Manual*. IBM Research, 2000.

7. P. Tarr, H. Ossher, W. Harrison, and S. Sutton. "*N* degrees of separation: Multi-dimensional separation of concerns." In *Proc. ICSE*, pp. 107–119, 1999.

8. Xerox Corporation. *AspectJ 0.7b12 Design Notes.* http://www.aspectj.org