Composition Patterns: An Approach to Designing Reusable Aspects

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
Requirements such as distribution or tracing have an impact on multiple classes in a system. They are *cross-cutting* requirements, or *aspects*. Their support is, by necessity, scattered across those multiple classes. A look at an individual class may also show support for cross-cutting requirements tangled up with the core responsibilities of that class. Scattering and tangling make object-oriented software difficult to understand, extend and reuse. Though design is an important activity within the software lifecycle with well-documented benefits, those benefits are reduced when cross-cutting requirements are present. This paper presents a means to mitigate these problems by separating the design of cross-cutting requirements into *composition patterns*. Composition patterns require extensions to the UML, and are based on a combination of the subject-oriented model for composing separate, overlapping designs, and UML templates. This paper also demonstrates how composition patterns map to one programming model that provides a solution for separation of cross-cutting requirements in code—aspect-oriented programming. This mapping serves to illustrate that separation of aspects may be maintained throughout the software lifecycle.

1 INTRODUCTION

Software design is an important activity within the software lifecycle and its benefits are well documented [4, 5]. These 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. However, in practice, object-oriented design models have been less useful throughout the lifetime of software systems than these benefits suggest. As described in [3], a structural mismatch between requirements specifications and object-oriented software specifications cause a reduction in the expected benefits of design. A single requirement, such as distribution or synchronisation, may impact a number of classes in a system, and therefore its support is *scattered* across those multiple classes. Such requirements are described as *cross-cutting* requirements, or *aspects*. On the other hand, a single class in a system may show support for multiple requirements *tangled* with the core responsibilities of that class. Scattering and tangling have a negative impact across the development lifecycle, from comprehensibility, traceability, evolvability, and reusability points of view.

The subject-oriented design model [2, 3] removes this structural mismatch with decomposition capabilities that support the separation of the design for each requirement into different design models (called *design subjects*). Decomposition in this manner removes requirement scattering and tangling properties from software design, thereby also removing their negative impact. Corresponding composition capabilities are supported within the subject-oriented design model.

The primary contribution of this paper is the specification of a means to capture *reusable patterns of cross-cutting behaviour* at the design level: *composition patterns*. A composition pattern is a design model that specifies the design of a cross-cutting requirement independently from any design it may potentially cross-cut, and how that design may be re-used wherever it may be required. Composition patterns are based on a combination of the subject-oriented model for decomposing and composing separate, potentially overlapping designs, and UML templates [13]. Section 2 motivates the need for composition patterns. The composition patterns model is described in Section 3, with Section 4 providing examples of well-known cross-cutting behaviours, designed to be truly reusable.

A secondary contribution is an introduction to how one programming model that supports cross-cutting behaviour (aspect-oriented programming [11]) is supported at the design level by composition patterns. This support serves to illustrate that separation of aspects may be maintained throughout the software lifecycle. Suggested mappings to constructs in a particular aspect-oriented programming language (AspectJ [21]) are illustrated in Section 5. Sections 6 and 7 present related work and conclusions, respectively.

2 MOTIVATION

To motivate the need for composition patterns, we use the simple example of requiring that operations be traced at run-time. Tracing is a particularly pervasive requirement that potentially impacts every operation in every class of a system. Without a means to separate its design, the potential for scattering and tangling across a system is enormous. A simplified design for tracing an operation in class X is contained in Fig. 1 (using UML [13]).

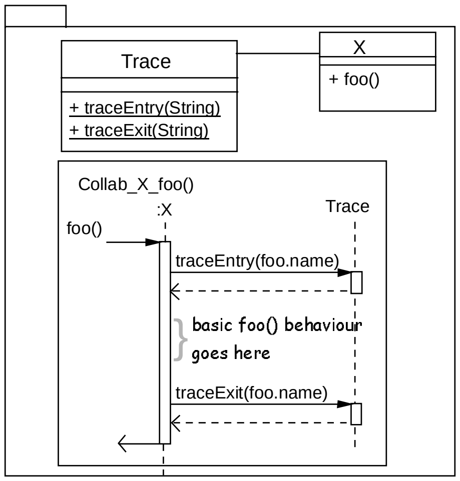

Fig. 1: Pervasive Trace Design

From a structural design perspective, the design elements supporting tracing may be separated into a class, `Trace`, upon which any class requiring trace behaviour may depend. However, to design the trace behaviour of operations, this separation is not possible to maintain. In the interaction diagram, we see that, when the `foo()` operation of class `X` is invoked, it immediately calls the `traceEntry()` operation of class `Trace` with a parameter denoting the name of the entered method, i.e., `foo`. Likewise, when the `foo()` operation has finished and is about to return, it finally calls the `traceExit()` operation of class `Trace`. Any other operations requiring trace behaviour would need to be designed analogously.

This design has a number of difficulties. First, any new operation requiring trace behaviour must specify an interaction model indicating this—a tedious and error-prone process. Secondly, changing or eliminating the trace behaviour design requires changes to all operation interaction models. Finally, reuse of this design in a different system is not straightforward. Structurally, the `Trace` class may be simply copied, but the trace behaviour must be re-defined in the interaction specification of each operation to be traced.

Composition patterns mitigate these problems by supporting the separate design of *reusable*, cross-cutting requirements. A composition pattern supports the design of a cross-cutting requirement independently from any design it may potentially cross-cut, and the specification of how that design may be re-used wherever it may be required—i.e., its pattern of composition.

3 COMPOSITION PATTERNS: THE MODEL

Composition patterns were introduced, without in-depth discussion, as part of the subject-oriented design model in [3]. They are based on a combination of the subject-oriented design model for decomposing and composing separate, potentially overlapping designs, and UML templates. The subject-oriented design model supports separate design models as independent views called *design subjects* (denoted with a «subject» stereotype on a UML package). Design subjects may specify cross-cutting behaviour to be composed with other design subjects within the subject-oriented design model, but by parameterising such a design subject, and providing a mechanism for binding those parameters to model elements in other design subjects, we can specify the composition of cross-cutting behaviour with base designs in a *reusable* way.

Specifying Templates

The UML defines a template as a parameterised model element that cannot be used directly in a design model. Instead, it may be used as the basis to generate other model elements using a "Binding" dependency relationship. A Binding relationship defines arguments to replace each of the template parameters of the template model element. The UML orders template parameters in a dotted box on the template class. Since a composition pattern is a design subject with potentially multiple *pattern classes* (classes that are placeholders to be replaced by real class elements), the representation of all the template parameters for all pattern classes is combined in a single box and placed on the subject box. Within this box, template parameters are grouped by class (each class grouped within <> brackets).

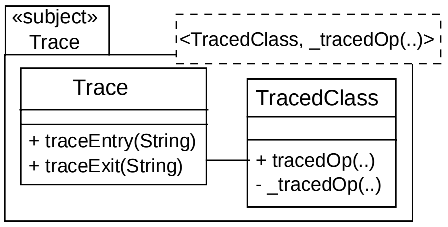

Fig. 2: Specifying templates in a `Trace` Composition Pattern

Fig. 2 illustrates a composition pattern with one pattern class, `TracedClass`, denoting that any class may be supplemented with trace behaviour. A template parameter is defined for the pattern class, called `_tracedOp()`, which represents any operation requiring tracing behaviour. One standard class, called `Trace`, is also included in the design. The design of tracing behaviour is now contained in the `Trace` composition pattern model, with references made to the pattern class and template operation as required.

Specifying Cross-Cutting Behaviour

Cross-cutting behaviour essentially supplements (or merges with) behaviour it cuts across. The subject-oriented design model supports merging of operations by allowing a designer to identify operations in different design subjects that correspond and should be merged. This means that execution of any one of the corresponding operations results in the execution of all of the corresponding operations. This is achieved within the model with the generation of an interaction model realizing the composed operation as delegating to each of the corresponding (re-named) input operations [1, 2].

This semantics can be utilized for the specification of patterns of cross-cutting behaviour. The designer may explicitly refer to the input and composed operations separately. The designer defines an input operation as a template parameter and refers to an actual, replacing operation by pre-pending an underscore to the template name. The generated output operation is referenced with the same name, but without the pre-pended underscore.

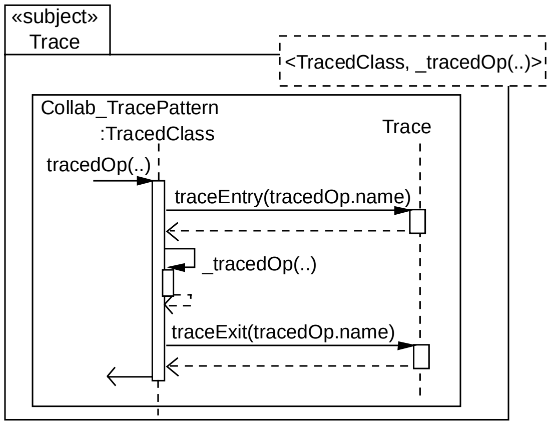

Fig. 3: Specifying Patterns of Cross-Cutting Behaviour

Parameter	Usage
`op()`	In this case, the replacing operation must have no parameters.
`op(..)`	Here, the replacing operation may have any signature.
`op(.., Type, ..)`	Here, the replacing operation may have any signature but the pattern needs a `Type` object for execution.

Table 1: Parameter Scope

As specified by the composition pattern in Fig. 3 for pattern class `TracedClass`, execution of any operation that replaces the `_tracedOp(..)` template will, in the output subject, result in the execution of `traceEntry()` before the execution of the replacing operation and in the execution of `traceExit()` after the execution of the replacing operation. The "`..`" parameter specification of `_tracedOp(..)` indicates that an operation of any signature may replace the template. The parameter possibilities are defined in Table 1. This `Trace` composition pattern subject effectively specifies the merging of trace behaviour with any operation replacing `_tracedOp(..)`.

Composition Binding Specification

The subject-oriented design model defines a composition relationship to support the specification of how different subjects may be integrated to a composed output, and the UML defines a Binding relationship between template specifications and the elements that are to replace those templates. The UML restricts binding to template parameters for instantiation as one-to-one. The composition patterns model combines the two notions by extending standard composition relationships with a `bind[]` attachment that defines the (potentially multiple) elements that replace the templates within the composition pattern. Ordering of parameters in the `bind[]` attachment matches the ordering of the templates in the pattern's template box. Any individual parameter surrounded by brackets `{}` indicates that a set of elements, with a potential size > 1, replace the corresponding template parameter.

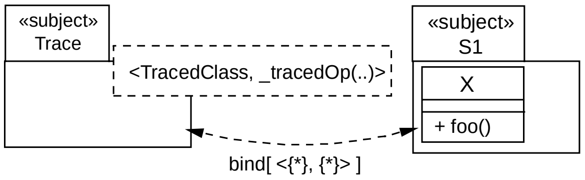

Fig. 4: Specifying Binding for Composition

In Fig. 4, all classes within `S1` are replacements for pattern class `TracedClass`, with every operation (denoted by `{*}`) in each class (in this example, just `S1.X`) supplemented with the pattern behaviour specified for `_tracedOp(..)`. Where specific elements from classes replace templates, they may be explicitly named.

Composition Output

As illustrated in Fig. 4, a composition relationship's `bind[]` attachment may specify multiple replacements for pattern classes and template operations within those classes. Where multiple replacements are specified for a pattern class, the pattern class's properties are added to each of the each replacement classes in the output subject. For example, in Fig. 5, class `X` has class `TracedClass`'s properties. Where multiple replacements are specified for operations, each operation is supplemented with the behaviour defined within the pattern subject. Non-template-parameter elements are added to each *result scope* once. For example, non-pattern classes are added once to the result. Class `Trace` is a non-pattern class defined in the `Trace` composition pattern (see Fig. 2), and it therefore appears in the output subject.

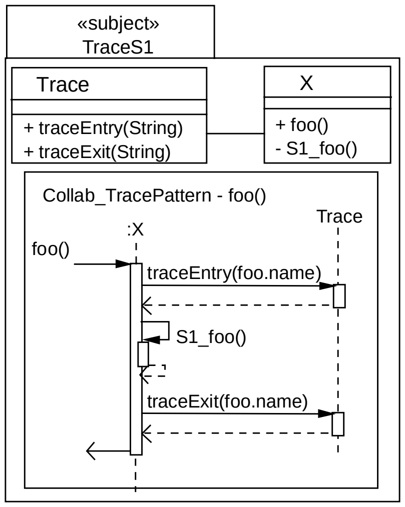

Fig. 5: Output from Composition with `Trace` Subject

Wherever a pair of operations has been defined (e.g., `tracedOp()` and `_tracedOp()`) and referenced within the same pattern class, and one is a template parameter for that class, composition applies merge operation semantics. For each operation substituting the template operation, each reference to `_tracedOp()` is replaced by the suitably re-named substituting operation, and a new interaction for `tracedOp()` is also defined. Each operation's delegation semantics is realized by a new collaboration as specified within the composition pattern.

Other implications of composition relating to the subject-oriented design model not demonstrated in this example are discussed elsewhere [2]. For example, merging generalizations may result in multiple inheritance in the composed subject, where there was only single inheritance in each of the input subjects. Multiple inheritance, supported in the UML, can be eliminated through the process of flattening [20]; this process can be automated during composition.

Composition of the design subjects can occur during the design phase (via a design composition tool, for example), which would be useful for the purposes of checking the semantics of the composed subject and the correctness of the composition relationships. Implementation may be based on such a composed design, but ideally, composition should be delayed until after the implementation phase, with each design subject being implemented separately, and being composed afterwards. We discuss a means of delaying the composition process until after the implementation phase in Section 5.

4 ASPECTS AS PATTERNS: EXAMPLES

We now look at some more examples of well-known aspects designed using composition patterns. These examples illustrate how cross-cutting requirements may be designed independently of any base design, making aspect design truly reusable. The aspects illustrated are synchronisation and the observer pattern.

Library Base Design

The base design on which the aspect examples are applied is a small library design (see Fig. 6). 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.

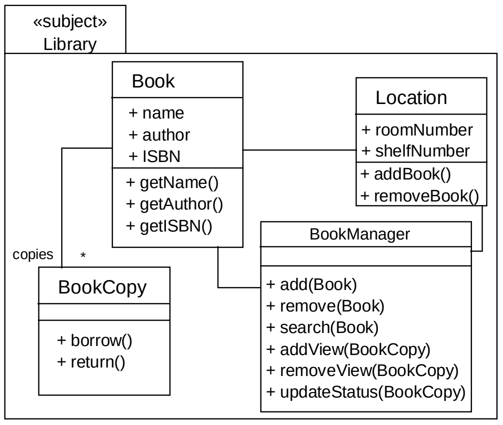

Fig. 6: **Base Library Design**

Example 1: Synchronisation Aspect

This first cross-cutting requirement is that the book manager should handle several requests to manage books and their locations concurrently. This aspect example, first described in [12], supports the book manager handling several "read" requests concurrently, while temporarily blocking "write" requests. Individual "write" requests should block all other services.

Pattern Specification

Synchronisation of concurrent processes is a common requirement, and therefore it is useful to design this behaviour without any reference to our library example. Fig. 7 illustrates how this can be achieved. The `Synchronize` composition pattern has one pattern class, `SynchronizedClass`, representing any class requiring synchronisation behaviour.

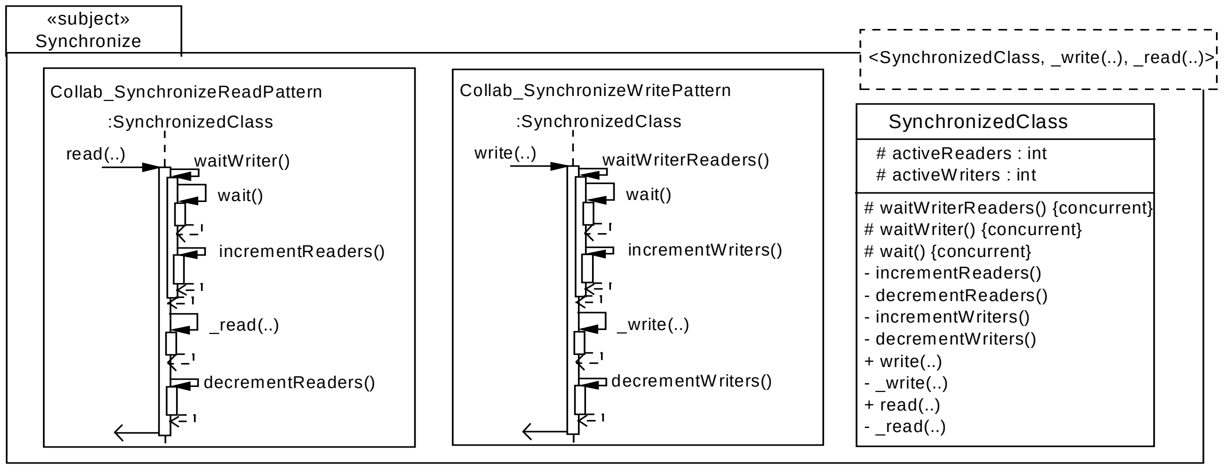

Fig. 7: Synchronisation Aspect Design

Within this pattern class, two template parameters are defined, called `_read(..)` and `_write(..)`, to represent reading and writing operations. This example also illustrates how non-template elements may be specified within a pattern class to define the inherent cross-cutting behaviour. These elements are merged unchanged into the composed class. Synchronisation behaviour introduces a number of such elements, both structural and behavioural, to synchronised classes. Structural properties `activeReaders` and `activeWriters` maintain counts of the number of read and write requests currently in process (for write, this number will never be > 1). Two interaction patterns define the required behaviour for reading and writing. The read pattern ensures that any currently writing process is complete prior to processing a read request. The write pattern ensures that all currently reading and writing processes are complete prior to processing a write request. The designer represents the actual replacing read and write operations with an "_" pre-pended to the template parameter name—that is, using `_read(..)` and `_write(..)`, and the generated operations realized by the interactions as `read(..)` and `write(..)`. In this way, when a replacing operation is executed in the context of synchronisation, the required behaviour is clearly defined within the interactions.

Pattern Binding

Specifying how to compose the `Library` base design subject with the `Synchronize` composition pattern is a simple matter of defining a composition relationship between the two, denoting which class(es) are to be supplemented with synchronisation behaviour, and which read and write operations are to be synchronised.

In this case, the library's `BookManager` class replaces the pattern class in the output, `add()` and `remove()` operations are defined as write operations, and the `search()` operation defined as read (see Fig. 8).

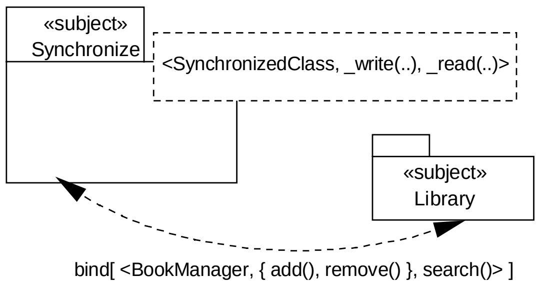
«subject»
Synchronize
<SynchronizedClass, _write(..), _read(..)>
«subject»
Library
bind[<BookManager, { add(), remove() }, search()>]

Fig. 8: Composing `Synchronize` with `Library`

Composition Output

Pattern specification and binding, as previously illustrated, is all the designer has to do to define truly reusable aspects patterns, and specify how they are to be composed with base designs. The composition process, utilizing UML template semantics, produces the design illustrated in Fig. 9, where `BookManager` now has synchronising behaviour. Note also that the object names in the interactions have been renamed as appropriate.

Example 2: Observer Aspect

The second example is the observer pattern [6], which, unlike the synchronisation example, describes the pattern of collaborative behaviour between more than one object—a subject and observers. This example also illustrates how non-pattern classes may be used within a composition pattern.

Pattern Specification

In the `Observer` composition pattern, two pattern classes are defined. `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. 10).

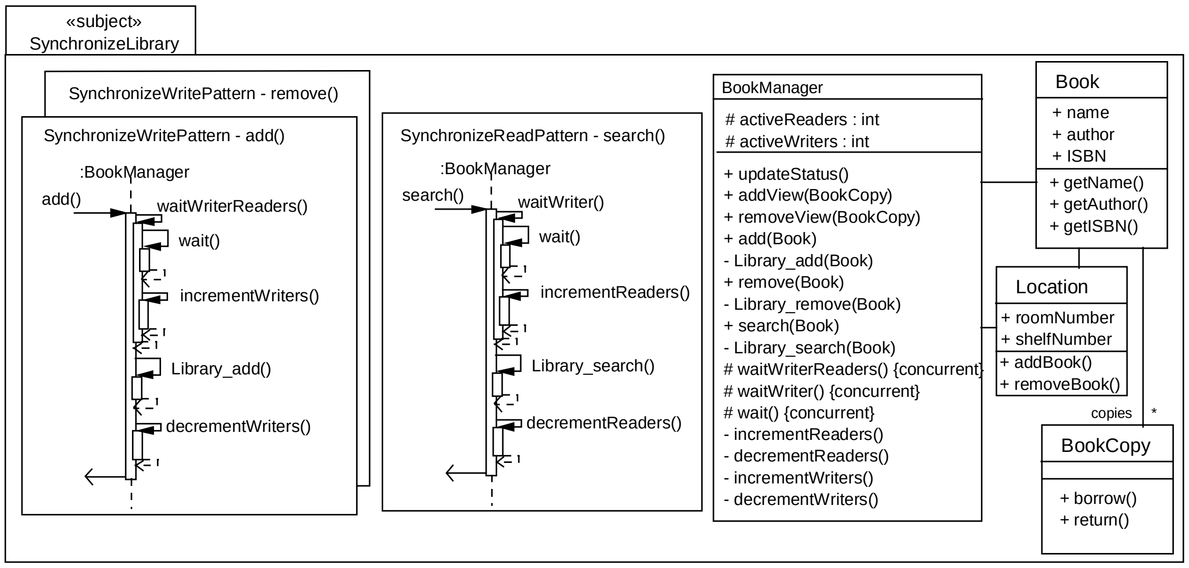

Fig. 9: `Library` Design with `Synchronize`

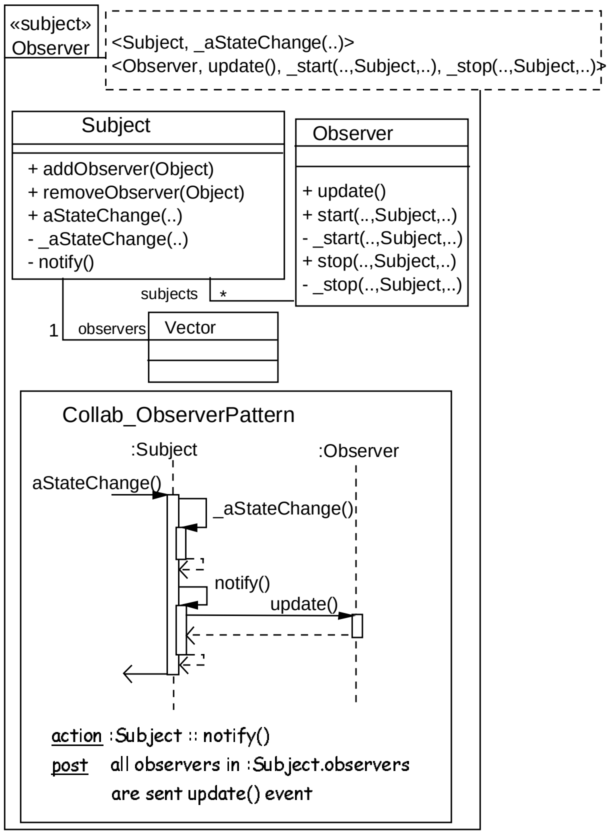

Fig. 10: Observer Aspect Design

The interaction in Fig. 10 illustrates another example of specifying behaviour that cross-cuts templates, with `Subject`'s template parameter `_aStateChange(..)` supplemented with behaviour relating to notifying observers of changes in state. Again, this achieved by referring to the actual replacing operation with a pre-pended "_", i.e., `_aStateChange(..)`. Here also is an example of an operation template parameter that does not require any delegating semantics. The `update()` operation in observers is simply called within the pattern, and is not, itself, supplemented otherwise. It is defined as a template so that replacing observer classes may specify the operation that performs this task.

This pattern also supports the addition and removal of observers to a subject's list using `_start(.., Subject, ..)` and `_stop(.., Subject, ..)` template parameters, where each is replaced by operations denoting the start and end, respectively, of an observer's interest in a subject. For space reasons, the interactions are not illustrated here, as they do not illustrate any additional interesting properties of the composition pattern model.

Pattern Binding
As with the `Synchronize` pattern, specifying the composition of `Library` with the `Observer` pattern is done by specifying a composition relationship between the two, defining the class(es) acting as subject, and the class(es) acting as observer. In this example, there is only one of each (see Fig. 11), `BookCopy` and `BookManager`, respectively.

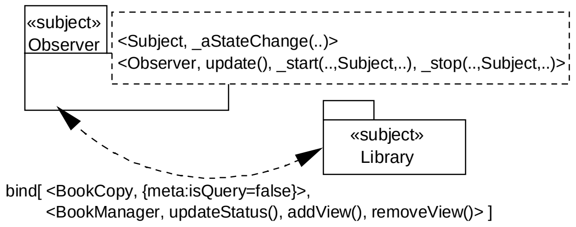
«subject» Observer
<Subject, _aStateChange(..)>
<Observer, update(), _start(..,Subject,..), _stop(..,Subject,..)>
«subject» Library
bind[<BookCopy, {meta:isQuery=false}>,
<BookManager, updateStatus(), addView(), removeView()>]

Fig. 11: Composing `Observer` with `Library`

One additional point of interest with this binding is how the meta-properties of a design subject's elements may be queried to assess an element's eligibility to join a set of replacing elements. The UML's Operation metaclass defines more properties for operations than most coding languages—for example, in addition to its signature, a designer may specify that an operation is a query operation. The composition patterns model supports the discrimination of replacing operations on the basis of any of a design element's properties.

In this example, the `_aStateChange()` template parameter 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` .

Composition Output
The output of composing `Observer` with `Library`, illustrated in Fig. 12, shows `BookCopy` demonstrating subject behaviour, with the operations `borrow()` and `return()` initiating the notification of observers, as they are the only state-changing operations. `BookManager`, as an observer, has defined `updateStatus()` as the operation to be called for notification purposes. Though not shown, `addView(..)` and `removeView(..)` initiate a `BookCopy` adding and removing a `BookManager` from its list of observers.

5 MAP TO ASPECTJ
The previous section illustrated the output resulting from the integration of composition patterns with base design models. Of course, each output demonstrates the tangling properties aspects are designed to avoid. While composition at the design level is useful to validate the design of a composition pattern, and its impact on a base design, it is also possible to maintain the separation to the code phase, using an appropriate implementation model.

Conceptually, subject-oriented design evolved from the work on subject-oriented programming [8,14]. Efforts into the development of a tool to support multi-dimensional separation of concerns (as evolved from subject-oriented programming) are currently centred around Hyper/J [18]. Though the precise mappings from constructs in composition patterns to constructs in Hyper/J has not yet been done, it is expected that it should be relatively straightforward. As a demonstration of the independence of composition patterns as a *design* model, from any particular *programming* model, this section looks at the mapping of composition pattern concepts to the aspect-oriented programming paradigm.

At the conceptual level, composition pattern design and aspect-oriented programming also have the same goals. Composition patterns provide a means for separating and *designing* reusable cross-cutting behaviour, and aspect-oriented programming provides a means for separating and *programming* reusable cross-cutting behaviour. This section introduces possibilities for mapping composition pattern constructs to current aspect-oriented programming constructs. Research into, and development of, technology support for the aspect-oriented programming paradigm is currently centred around AspectJ, and so, using the synchronisation example, we assess how composition patterns map to AspectJ programming constructs.

AspectJ Programming Elements
AspectJ is an aspect-oriented extension to Java that supports programming cross-cutting code (i.e., aspects) as separate aspect programs. As described in [21], AspectJ adds four kinds of program elements to Java. These are: an *aspect*, a *pointcut*, a piece of *advice* and an *introduction*.

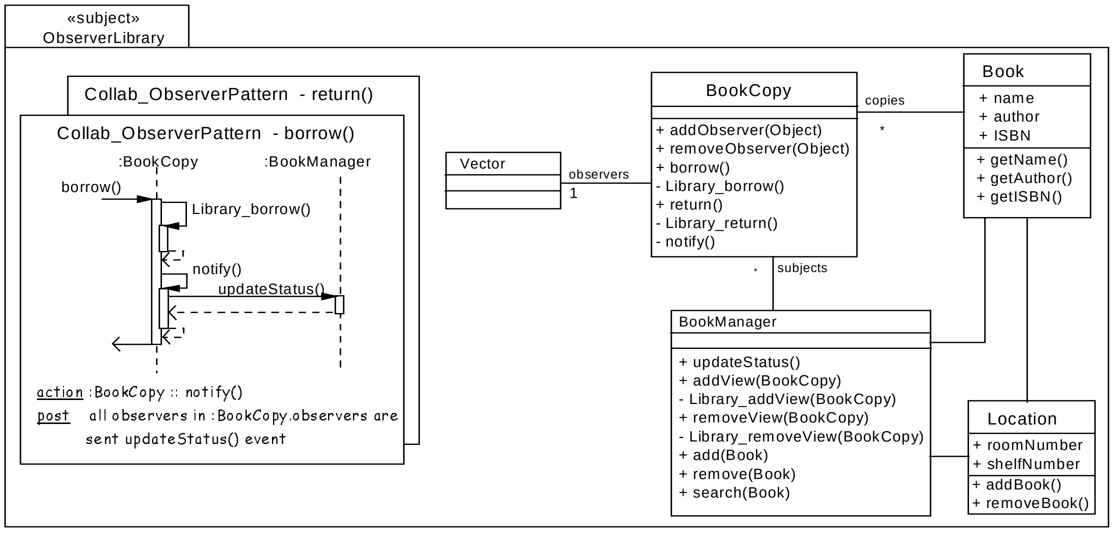

Fig. 12: `Library` Design with `Observer`

Element	AspectJ	Composition Patterns
Aspect	An aspect is cross-cutting type, with a well-defined interface, which may be instantiated and reasoned about at compile time. Keyword: `aspect`	A composition pattern subject is a design equivalent to an aspect.
Pointcut	During the execution of a program, and as part of that execution's scope, there are points where cross-cutting behaviour is required. These points are *join points.* A pointcut is a *cross-cutting set of join points.* Keyword: `pointcut`	Operation template parameters may be defined and referenced within interaction specifications, denoting that they are join points for cross-cutting behaviour. These templates may be replaced by actual operations multiple times, and are therefore equivalent to pointcuts.
Advice	A piece of advice is code that executes at a pointcut, using some of the execution scope. Keywords: `before`, `after`, `around`	Within an interaction diagram, cross-cutting behaviour may be specified to execute when a template operation is called. This behaviour is equivalent to advice code.
Introduction	An introduction is a programming element, such as an attribute, constructor or method, that is added to a type that may add to or extend that type's structure. Keyword: `introduction`	Design elements that are not template elements may be defined within composition patterns. These may be classes, attributes, operations or relationships, and are equivalent to an introduction.

Table 2: Mapping AspectJ Program Elements to Composition Pattern Elements

Table 2 describes these elements and maps them to the corresponding design elements in composition patterns.

`Synchronize` in AspectJ

The `Synchronize` composition pattern (Fig. 7) with its composition specification to the Library subject (Fig. 8) provides the information required for the structure of an aspect program. The composition pattern has one class defined, which is a pattern class, and therefore is replaced with a concrete design class. The composition relationship's binding specification indicates that `BookManager` replaces the `SynchronizedClass` pattern, and therefore, all non-pattern elements defined within `SynchronizedClass` are *introduced* to `BookManager` (only those relating to write operations are discussed here for space reasons).

```
public aspect Synchronize{
 introduction BookManager{
  int activeWriters;
  int activeReaders;
  private void incrementWriters();
  private void decrementWriters();
  protected synchronized void
              waitWriterReaders();
  protected synchronized void wait(); } }
```

First, the composition pattern's name may be used for the `aspect` declaration. Also, the operation template parameter defined in `Synchronize`, `write(..)`, may be seen as a pointcut in replacing classes. The composition relationship between `Synchronize` and `Library` indicates that the `BookManager` operations `add(Book)` and `remove(Book)` replace `write(..)`. As regards the advice code, the interaction (sequence) diagrams specified within the `Synchronize` composition pattern indicate when "advice" operations should be called relative to the template operations. These directly translate to the `before` and `after` constructs of the AspectJ advice element. This information maps to the following programming elements of aspects:

```
public aspect Synchronize{
  pointcut write(BookManager b):
    instanceof(b) & receptions(
         void addBook(Book),
         void removeBook(Book));
  before(BookManager b): write(b) {
     b.waitWriterReaders(); }
  after(BookManager b): write(b) {
     b.decrementWriters(); }
}
```

This example illustrates the possibilities for mapping composition pattern constructs to AspectJ programming elements. The advantages of this are two-fold. First, from a design perspective, mapping the composition pattern constructs to constructs from a programming environment ensure that the clear separation of cross-cutting behaviour is maintained in the programming phase, making design changes easier to incorporate into code. Secondly, from the programming perspective, the existence of a design approach that supports separation of cross-cutting behaviour makes the design phase more relevant to this kind of programming, lending the standard benefits of software design to the approach.

6 RELATED WORK

Conceptually, the subject-oriented design and composition patterns model has evolved from the work on subject-oriented programming [8, 14]. Different subjects may be designed (or programmed) to support separate requirements, be they functional (and conceptually overlapping) or cross-cutting requirements. Subsequent composition of separated subjects is specified with composition relationships (or defined by composition rules in subject-oriented programming). Without composition patterns, reusing cross-cutting behaviour in subject-oriented design typically requires the detailed specification of multiple composition relationships for each instance of reuse—a tedious and error-prone process that should be simple. From the perspective of cross-cutting requirements, this paper has illustrated how the subject-oriented design and composition patterns model also closely relates to the aspect-oriented programming model [11]. While there are many approaches to flexible separation of concerns at the programming level, space precludes their inclusion.

There are some interesting approaches that start with the aspect-oriented programming paradigm, and attempt to extrapolate the ideas as extensions to the UML. Two general approaches to this are evolving. On the one hand, there are approaches to extending the UML with stereotypes specific to particular aspects (e.g., synchronisation [9] or command pattern [10]). In such approaches, the constructs required by each particular aspect are stereotyped so that a weaver (an automated composition tool) can determine which elements match the appropriate aspect construct. In both these examples, many of the behavioural details of synchronisation and of the command pattern are not explicitly designed in the UML—the onus appears to be on the weaver to provide the aspect behaviour. Other general approaches attempt a more generalised way to support aspect-oriented programming in UML. For example, in [17] a new metaconstruct called `Aspect` is created, and stereotypes are defined for advice behaviour. Operations requiring advice behaviour are constrained by the advice stereotypes. Where the composition patterns model distinguishes itself from both these general approaches is with its generic approach to designing re-usable cross-cutting behaviour in a manner that is independent of a particular programming environment.

Two approaches that emphasise a more flexible separation of concerns than exists in standard object-oriented design are OORam [15] and Catalysis [5]. In role modelling in OORam, large systems are described with multiple different role models that may be synthesised to create derived models. This is similar to merge integration in subject-oriented design. Catalysis also supports the decomposition of software designs along “vertical” and “horizontal” lines, providing the ability to separate both functional and technical kinds of concerns. While both approaches provide advances in the separation capabilities in object-oriented design, neither addresses the specification and composition of patterns of cross-cutting behaviour.

A focus of work in the field of collaboration-based design is on separation of each of the roles classes may play in different collaborations into different modules. For example, modularisation of roles within collaborations are supported by mixins in the work described in [19], utilising a C++ template class for each role in each collaboration; complete classes are “composed” by placing these mixins in a hierarchy. This work is extended in [16] to overcome problems of scalability by grouping sets of roles within each collaboration. There are two main drawbacks to these approaches, relative to the composition patterns model. First, while classes can be mixed together simply by adding mixins to a class hierarchy, individual methods cannot; this is part of the reason tool support is needed in subject-oriented and aspect-oriented programming. Secondly, whenever the mixins are interdependent, one needs to tangle the dependencies between them by embedding details of the dependencies within each mixin, typically via explicit calls to super-methods and other constraint information. In any situation where prior knowledge of future changes is unknown, such dependencies would require error-prone modifications to pre-existing mixins. As a result, each mixin cannot be designed cleanly and independently as composition patterns allow.

The need for advanced separation of concerns across the development lifecycle is described in [7], motivated by agent-based product-line component based development for e-Commerce. A development process is proposed that draws together high-level analysis and design separation techniques and corresponding, supporting implementation techniques. Composition patterns provide a solution to designing cross-cutting features in a re-usable way that could be considered for inclusion in a development process such as this.

7 CONCLUSIONS

Software design is an important activity in the development lifecycle but its benefits are often not realized. Scattering and tangling of cross-cutting behaviour with other elements causes problems of comprehensibility, traceability, evolvability, and reusability. Attempts have been made to address this problem in the programming domain but the problem has not been addressed effectively at earlier stages in the lifecycle. This paper presents an approach to addressing this problem at the design stage with composition patterns. Composition patterns are based on the combination of merge integration from subject-oriented design and UML templates. Examples are presented that illustrate the flexible and reusable nature of composition patterns as a design approach for cross-cutting behaviour. The paper illustrates how both separation of aspects, and composition with other design models may be specified. The impact of possible composition at the design stage is demonstrated. The paper also illustrates how the separation may be maintained to the programming phase by mapping the composition patterns model constructs to AspectJ constructs.

While many of the ideas evolved from subject-oriented programming, the ease in which its concepts are mapped to aspect-oriented programming constructs for cross-cutting behaviour illustrates that it is not closely tied to a particular programming paradigm. Therefore, it is insulated from changes to programming environment constructs. In addition, extensions to the UML are minimal—use is made, where possible, of standard UML constructs (e.g., templates). This should make use of composition patterns intuitive to UML designers. However, tool support is essential for successful usage, and therefore one of our next primary foci is the inclusion of support for composition patterns in a design tool. Automation of the mapping to programming environments such as AspectJ would also be useful.

ACKNOWLEDGEMENTS

Many thanks to Don Batory and the anonymous reviewers for their comments on early drafts of this paper.

REFERENCES

1. S. Clarke. “Composing Design Models: An Extension to the UML”. In *Proc. of International Conference on the UML*, pp. 338–352, 2000.
2. S. Clarke. *Composition of Object-Oriented Software Design Models*. PhD Thesis, Dublin City University, January 2001.
3. S. Clarke, W. Harrison, H. Ossher, and P. Tarr. “Subject-Oriented Design: Towards Improved Alignment of Requirements, Design and Code”. In *Proc. of OOPSLA*, pp. 325–339, 1999.
4. S. Cook and J. Daniels. *Designing Object Systems: Object-Oriented Modelling with Syntropy*. Prentice Hall, 1993.
5. D. D’Souza and A. Wills. *Objects, Components and Frameworks with UML. The Catalysis Approach*. Addison-Wesley, 1998.
6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
7. M. Griss. “Implementing Product Line Features by Composing Component Aspects”. In *Proc. of First International Software Product Line Conference*, pp. 271–288, 2000.
8. W. Harrison and H. Ossher. “Subject-Oriented Programming (a critique of pure objects)”. In *Proc. of OOPSLA*, pp. 411–428, 1993.
9. J. Herrero, F. Sánchez, F. Lucio, and M. Toro. “Introducing Separation of Aspects at Design Time”. In *Proc. of Aspects and Dimensions of Concerns Workshop* at ECOOP, 2000.
10. W. Ho, F. Pennaneac’h, J. Jézéquel, and N. Plouzeau. “Aspect-Oriented Design with the UML”. In *Proc. of Multi-Dimensional Separation of Concerns Workshop* at ICSE, pp. 60–64, 2000.
11. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. “Aspect-Oriented Programming”. In *Proc. of ECOOP*, pp. 220–242, 1997.
12. C. Lopes and G. Kiczales. *D: A Language Framework for Distributed Programming*. Technical report SPL97-010. Xerox PARC, 1997. http://www.parc.xerox.com/csl/projects/aop/.
13. OMG. *The Unified Modeling Language Specification*. Version 1.3, June 1999.
14. 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.
15. T. Reenskaug, P. Wold, and O. Lehne. *Working with Objects. The OORam Software Engineering Method*. Manning Publications Co., 1995.
16. Y. Smaragdakis and D. Batory. “Implementing Layered Designs with Mixin Layers”. In *Proc. of ECOOP*, pp. 550–570, 1998.
17. J. Suzuki and Y. Yamamoto. “Extending UML with Aspects: Aspect Support in the Design Phase”. In *Proc. of Aspect-Oriented Programming Workshop* at ECOOP, 1999.
18. P. Tarr and H. Ossher. *Hyper/J™ User and Installation Manual*. http://www.research.ibm.com/ hyperspace.
19. M. VanHilst and D. Notkin. “Using Role Components to Implement Collaboration-Based Designs”. In *Proc. of OOPSLA*, pp. 359–369, 1996.
20. R. Walker. *Eliminating Cycles from Composed Class Hierarchies*. Technical Report TR-00-09, Dept. of Computer Science, Univ. of British Columbia, 2000.
21. Xerox Corporation. *AspectJ 0.7b3 Design Notes*. http://www.aspectj.org.