

# Extending standard UML with model composition semantics

Siobhán Clarke\*

*Department of Computer Science, Trinity College, Dublin 2, Republic of Ireland*

---

## **Abstract**

There is a well documented problem in the software engineering field relating to a structural mismatch between the specification of requirements for software systems and the specification of object-oriented software systems. The structural mismatch happens because the units of interest during the requirements phase (for example, feature, service, capability, function etc.) are different to the units of interest during object-oriented design and implementation (for example, object, class, method, etc.). The structural mismatch results in support for a single requirement being scattered across the design units and a single design unit supporting multiple requirements – this in turn results in reduced comprehensibility, traceability and reuse of design models.

This paper presents an approach to designing systems based on the object-oriented model, but extending this model by adding new decomposition capabilities. The new decomposition capabilities support a way of directly aligning design models with individual requirements. Each model contains a design of an individual requirement, with concepts from the domain (which may appear in multiple requirements) designed from the perspective of that requirement. Standard UML is used to design the models decomposed in this way. Composition of design models is supported, and it is specified with a *composition relationship*. A composition relationship specifies how models are to be composed by identifying overlapping concepts in different models and specifying how models should be integrated. This paper describes changes required to the UML metamodel to support composition relationships. Two kinds of integration strategies are discussed – *merge* and *override*. © 2001 Elsevier Science B.V. All rights reserved.

---

## **1. Introduction**

Decomposition with conventional object-oriented design models is by class, interface and method. This kind of decomposition matches well with object-oriented code, providing a measure of traceability between object-oriented designs and code. However, it does not align well with the structure of requirements specifications, which are generally described by feature and capability. There is a negative impact to this structural mismatch – support for individual requirements is scattered across the design, and support for multiple

---

\* *E-mail address:* siobhan.clarke@cs.tcd.ie

requirements is tangled in individual design units. This reduces comprehensibility and traceability, making designs difficult to develop, re-use and extend.

To support the direct alignment of design models with requirements, we extend conventional object-oriented design by adding additional decomposition capabilities. The model, called *subject-oriented design*, supports the separation of the design of different requirements into different, potentially overlapping, design models. Subsequent composition of the separated design models is specified with *composition relationships*. A composition relationship identifies overlapping elements in different design models (called corresponding elements), and specifies how they are to be integrated. For example, a composition relationship with merge integration might be specified to compose models that may have been designed concurrently by different teams to support different requirements of the system, or to compose different optional features of a system. Composition relationships with override integration might be specified to support a situation where a design model is intended to extend or change the behaviour of an existing design model because a change to requirements makes (part of) the existing design model's behaviour obsolete.

Decomposition based on requirements, with corresponding composition specification using composition relationships, is not part of the UML. The primary contribution of this paper is the presentation of an overview of this model with the extensions required to the UML semantics and metamodel to support the model. First though, there is a brief section describing the motivation for this work in Section 2, followed by the description of the model in Section 3. Section 4 describes the extensions to the UML, and Sections 5 and 6 provide a look at related work, and some conclusions.

## **2. Motivation**

To illustrate the problems that motivate this work, an example is presented involving the construction and evolution of a simple university library management system (LMS). An assessment of the design is presented based on the criteria used by Parnas in [12]. They are: 1) *Product flexibility*; 2) *Comprehensibility*; and 3) *Managerial*. Parnas considered that these criteria were the benefits to be “expected of modular programming”. These benefits remain the goal of high-quality software engineering. We also consider the *reuse* properties of the design.

The functional requirements for the LMS revolve around the management of multiple copies of books and periodicals. The system handles addition, removal, ordering and search of these resources, and also the borrowing and return of books. Fines may be imposed for late return. A technical requirement is also included, imposing that the services for managing resources are available concurrently. Those services that change entities (e.g. add resource and remove resource) should only run one at a time, and should also lock out the query services (search for resource). On the other hand, multiple query services should be allowed run concurrently, but only when there are no changing services running.

Based on these requirements, a UML structural design for this LMS is illustrated in Fig. 1. A `ResourceManager` class handles the requests for managing resources such as add, remove, order and search for resources. This class coordinates with a `CopyNoGenerator` class for finding a unique number for each copy of a resource being added to the system. Each resource is on order for or has been added for a particular course, and so is related to a `Course` class. Borrowers' fines are calculated based on the maximum days they are allowed hold a book. Synchronisation of the services to manage resources is included in the `ResourceManager` class, and impacts the behavioural specification of adding, removing and searching for resources. One example of this is illustrated for the operation to add a book to the system, `addBook()`, in Fig. 2.

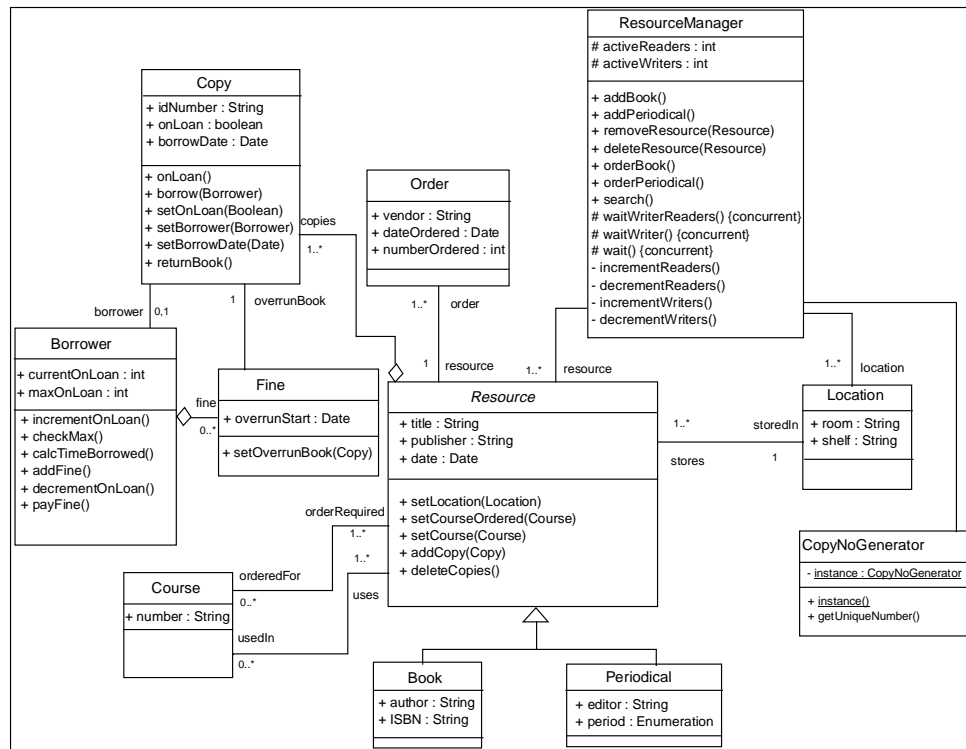


Fig. 1. LMS structural design in UML

### 2.1. Product Flexibility

Product flexibility is the “possibility of making drastic changes to one part of the system, without the need to change others” [12]. The structural differences in the specification paradigms between the requirements specification, and the object-oriented design of the LMS are central to the difficulties associated with changing the system. The natural outcome of the structural differences is a scattering and tangling affect across the object-oriented design.

*Scattering:* The structural difference results in the design of a single requirement to be scattered across multiple classes and operations in the object-oriented design. This property is apparent in the LMS, as the design of many of the requirements are scattered across the same core set of classes. This means that the impact of a change to a requirement is high, because a change will necessitate multiple changes across a class hierarchy.

*Tangling:* The structural difference also means that a single class or operation in the object-oriented design will contain design details of multiple requirements. This property is also present in the design of the LMS in, for example, the design of the synchronisation requirement. The protocol for synchronisation requires interaction with any method that needs to be synchronised, tangling its support with those methods (see Fig. 2 for one example). Dealing with synchronisation necessitates dealing with design details from multiple other requirements, making it difficult to change.

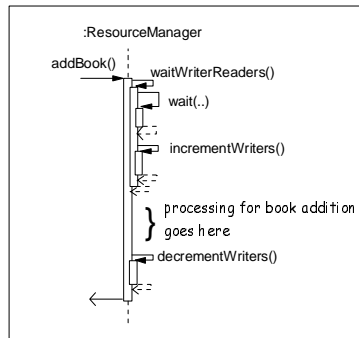


Fig. 2. Behavioural specification for adding a book resource

## 2.2. Comprehensibility

Comprehensibility is the “possibility of studying the system one part at a time. The whole system can therefore be better designed because it is better understood” [12]. The descriptions of the scattering and tangling problems as manifested in the LMS, and which are described in the previous section, also have a negative impact on the comprehensibility of the system. Any attempt at “studying the system one part at a time” will result in a required knowledge of the full design if the “one part” chosen is a requirement, or will result in a required knowledge of all the requirements if the “one part” chosen is a class in the design. This also has a negative impact on evolution.

## 2.3. Managerial

Managerial issues concern the “length of development time, based on whether different groups can work on different parts of the system with little need for communication” [12]. The abstraction units of the object-oriented paradigm (classes, interfaces, packages) are

inherently centralised, in that they each cleanly encapsulate (and own) all the structural and behavioural features relating to them. The monolithic nature of the classes has ramifications for the design process itself. For example, designers are limited in their ability to work concurrently on the design (and on the code), to a much greater degree than when producing a requirements specification. Specifically, it would be desirable to have a synchronisation expert work on the synchronisation requirement, a library resource expert work on the design of addition and removal of resources features, etc. However, scattering and tangling results in interdependencies across these features and across the classes, that hampers concurrent design and implementation.

#### *2.4. Reuse*

Scattering and tangling also has a negative impact on the reusability of the design artefacts. Consider the design for the synchronisation requirement. Synchronisation is not specific to the library domain, and therefore it is easy to imagine that its design might be useful in many different domains. However, it is not possible to simply extract either the structural or behavioural specification of synchronisation, as it is tangled within the `ResourceManager` class (see Fig. 1.), and its processing is specified within the interaction specification of all the methods to be synchronised (see Fig. 2).

Design patterns [5] prove useful in alleviating some of the problems described here in many cases, but their use results in the exchange of one set of problems for another. For example, the use of design patterns requires, in many cases, the need to pre-plan for change, since the designs and code must be pre-enabled with the pattern to avoid subsequent invasive changes to incorporate them. This, and other problems with the use of design patterns are more comprehensively dealt with in [1,2].

### **3. The Model**

The approach to addressing the structural mismatch described here is based on providing means of decomposing artefacts written in one paradigm so that they can align with those written in another. In other words, we want the designer to be able to design separate packages for each requirement, while also using object-oriented techniques within each individual package. This means that object-oriented designs must first decompose design packages by feature and capability, thereby encapsulating and separating their designs. Within each package, all the different kinds of UML design models may be included as required to support the requirement. Note that the semantics for *all* of the UML design models have not yet been defined, but in theory, all may be included. Since requirements are encapsulated, decomposition in this way removes the scattering of requirements across the full design. It also removes the tangling of multiple requirements in individual design

units, as requirements are separated into different design models. Decomposition in this manner has two important implications:

- *Overlapping specifications supported:* Different requirements may exist that have an impact on the same core concepts (for example, objects) of the system. It is this level of overlapping of requirements that is one of the causes of the problems with comprehensibility, extensibility and reuse discussed previously. The subject-oriented design model recognises and explicitly caters for overlap in the different design models for each requirement. This is achieved by allowing a separate design model to include the specification of any core concepts only as suits the requirement under design by that design model. Composition capabilities supported by this new approach cater for identifying overlapping concepts, integrating them, and handling any conflicts.
- *Crosscutting specifications supported:* There are also many kinds of requirements that will have an impact across the full design of a software system. For example, a requirement for distributed objects has an impact on a potentially large proportion of the objects of a computer system. Such requirements are referred to as crosscutting [9], since support for such requirements must be included across many different objects in a system. With the approach to decomposition described here, crosscutting requirements may also be designed separately, with composition capabilities handling their integration with other system objects as appropriate.

Decomposition in this manner also requires corresponding composition support, as object-oriented designs still must be understood together as a complete design. The subject-oriented design model supports a new kind of design construct, called a composition relationship that supports the specification of how design models should be composed. With composition relationships, a designer can:

- *Identify and specify overlaps:* Where decomposition allows overlaps in different design models, corresponding composition capabilities must support the identification of where those overlaps are. In order to integrate separate design models, overlapping design elements (or elements which correspond and therefore should be integrated into a single unit) are specified with composition relationships.
- *Specify how models should be integrated:* Design models may be integrated in different ways, depending on why they were modularised in a particular way. For example, if different design models were designed separately to support different requirements, a composed design where all requirements are to be included might be integrated with a merge strategy – that is, all design elements are relevant to the composed design. Alternatively, if a design model contains the design of a requirement that is a change to a requirement previously designed (for example, a business process has changed), then that design model might replace the previous design. In this case, integration with an override strategy is appropriate, where existing design elements are replaced by new design elements.
- *Specify how conflicts in corresponding elements are reconciled:* For some integration strategies, where some corresponding elements are integrated into a single design

element, (merge integration is an example of such a strategy) conflicts between the specifications of those corresponding elements must be reconciled. Composition relationships support the specification of different kinds of reconciliation possibilities – for example, one design model may take precedence over another, or default values should be used.

In addition, for design models that support crosscutting requirements, (i.e., those requirements that have an impact on potentially multiple classes in the design), composition of those models with other models is likely to follow a pattern. In other words, a crosscutting requirement has behaviour that will affect multiple classes in different design models in a uniform way. For these kinds of requirements, the subject-oriented design model defines a mechanism whereby this common way of composing the crosscutting requirement may be defined as a *composition pattern*. The synchronisation requirement for the LMS will be designed in this manner.

Decomposition and composition of designs as described in this paper is intended to support and extend the existing decomposition and composition mechanisms within standard UML. Some significant differences exist between the packages described here and UML packages. First, elements describing different views of the same core concept may appear in different packages with different specifications. With standard UML, such specifications will remain separate throughout the design cycle. With this approach, they may be composed, with differences resolved. In addition, crosscutting behavioural elements may be modularised with generic composition with other models supported (i.e., composition patterns). Approaches to component-based modelling (e.g., Catalysis [4]) are also enhanced with this approach, primarily with the modularisation and generic composition of crosscutting behaviours.

See Section 3.1. “Decomposing Design Models” for the application of the decomposition approach to the LMS. Section 3.2. “Composing Design Models” illustrates the use of composition relationships to specify the composed LMS design. Section 3.3. “Using Subject-Oriented Design” discusses possible areas of usefulness of the approach within the software development process. Section 4 “UML Metamodel Extensions” places the composition relationship in the context of the UML metamodel, and introduces the required extensions.

### *3.1. Decomposing Design Models*

For object-oriented design models, matching the structure of requirements means that design models must be divided up into separate packages that match that structure. These packages are called *design subjects*. Each design subject separately describes that part of a system or component that relates to a particular requirement, encapsulating its design and separating it from the design of the rest of the system. The kinds of requirements whose designs can be described in design subjects are varied. They include units of requirements like features, and crosscutting requirements (like persistence or distribution), that affect

multiple units of functionality. Where a requirement is a change to the existing design, a design subject may also encapsulate that change, making evolution of software additive rather than invasive.

For the LMS example, decomposing the design into design subjects that match the requirements yields a subject for each of the features in the requirements statement – a subject for adding resources, one for removing resources, one for ordering resources, one for searching for a resource, one for borrowing a book, one for returning a book, and one for paying a fine. Fig. 3 illustrates the structural designs for the subjects that handle the functional requirements for managing resources – AddResource, RemoveResource, OrderResource and SearchResource.

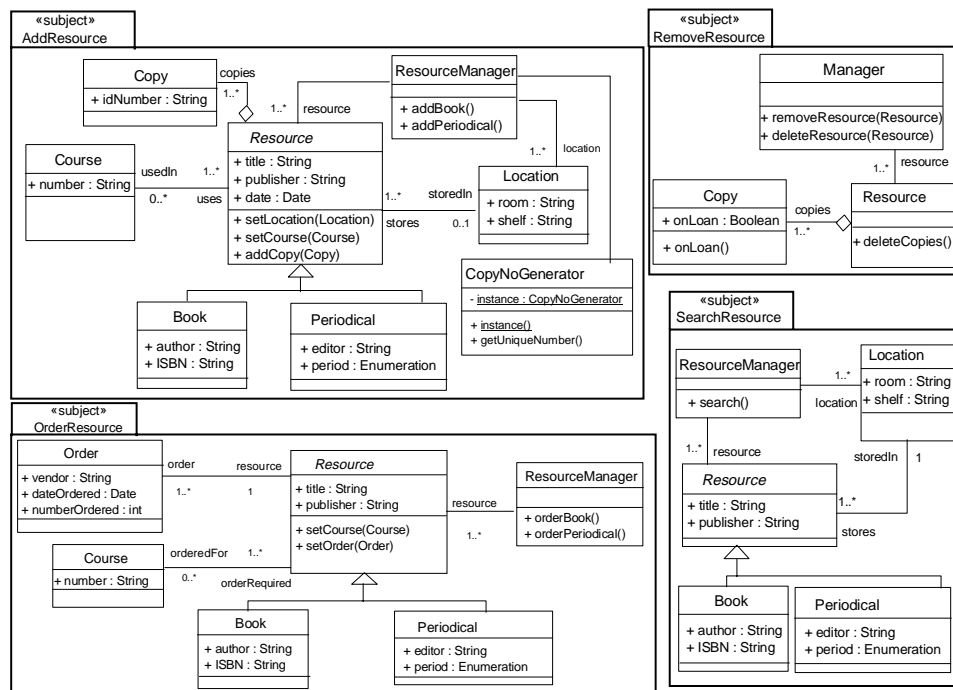


Fig. 3. Decomposing management of resources into different design models

Similarly, the designs for the subjects that handle the functional requirements for borrowing books may also be separated, such as `BorrowBook`, `ReturnBook` and `PayFine`. Neither the full class models, nor the supporting collaboration diagrams, are illustrated for space reasons. This example illustrates how designs for particular features can be encapsulated, and contain the design only from the perspective of that feature, regardless of the overlap in concepts across the subjects. This is one of the strengths of the subject-oriented design model – each of the different parts of a system under design may model the same concepts in whatever way is most appropriate for that subject’s view or purpose. Even in the small example of the LMS illustrated in Fig. 3, there are differences in the hierarchies specified in different subjects (`RemoveResource` has defined no subclasses for

Resource), and differences in naming of elements that support the same concept (the class managing resources is called `ResourceManager` in every subject except `RemoveResource`, where it was called `Manager`).

In Fig. 4, we see how a crosscutting requirement such as synchronisation can be designed separately from the operations to be synchronised – in this case, the operations to add, remove and search for resources. In this example, (an extension of) UML templates are used in the collaboration diagrams to design the synchronisation behaviour in relation to any operation to be synchronised. Semantics relating to merge integration of operations also influence this design, where a new operation is created to support the specification of merged behaviour, which delegates to the “real” operation. In this case, `write()` is created to define the additional synchronisation behaviour for operations that change state, while `_write()` represents the re-named actual writing operation from a subject merged with this pattern. Similarly, `read()` is created to define the additional synchronisation behaviour for reading operations, while `_read()` represents the actual reading operation from a subject to be merged with this pattern.

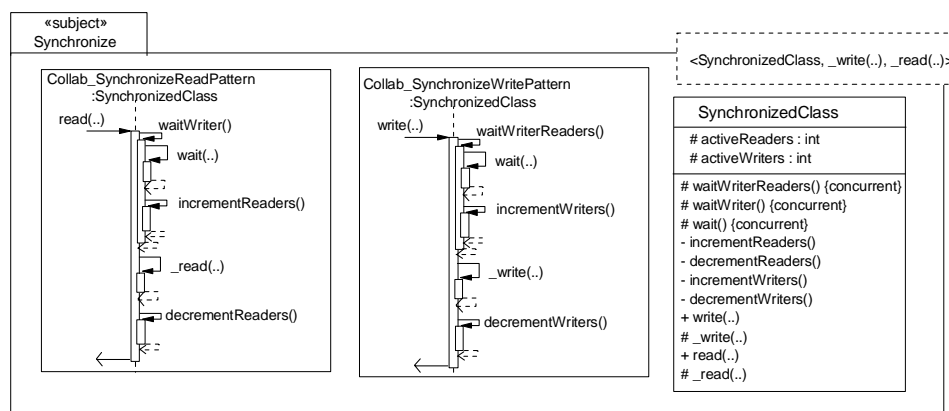


Fig. 4. Decomposing crosscutting requirements into a separate subject

The removal of scattering and tangling of the design specifications for different requirements has yielded a design that allows us work with different parts of the design independently. Understanding the design of any single requirement necessitates studying only the design subject for that requirement, where every element is relevant, and no element is present that does not directly support the requirement. This makes changing the design as a result of a change to any requirement more isolated to a single design subject, reducing the potential for side-effects. Reuse of individual design models is also better supported. For example, the design for synchronisation behaviour has been achieved without reference to any LMS domain operations, and therefore may be reused in any domain requiring such synchronisation.

### 3.2. Composing Design Models

Decomposing design models brings many benefits relating to comprehensibility, traceability, evolution and reuse. However, designs that have been decomposed must also be integrated at some later stage in order to understand the design of the system as a whole. This is required for reasons such as verification, or to support a developer's full understanding of the design semantics and the impact of composition.

A composition relationship between design models specifies how those design models are to be composed. The relationship identifies corresponding design elements in the related models, and specifies how the models are to be integrated. Different kinds of integration strategies may be attached to a composition relationship – for example, the models could be merged, or one model could override another. In the following examples, composition relationships are denoted with a dotted arc between the elements to be composed. The relationship arc indicates:

- Elements at every end of the arc correspond, and should be composed. An annotation to the arc such as `match[name]`, indicates that identification of correspondence between the components of related elements is by matching on the name property of the elements.
- The kind of integration strategy appropriate for this composition is denoted by the arrowheads at the end of the arc. For example, override integration is denoted with a single arrowhead at the end of the arc relating the element to be overridden. Merge integration is denoted with an arrowhead at every end of the arc.

Composition of design subjects results in a new design subject containing the integrated elements of the “input” design subjects. The input design subjects remain unchanged.

#### 3.2.1. Override integration

Override integration is used when an existing design subject needs to be changed. New requirements indicate that the behaviour specified in the existing design subject is no longer appropriate to the needs of end-users of the computer system. Therefore the behaviour as specified in the existing design subject needs to be updated to reflect the new requirements. An existing design subject is changed by creating a new design subject that contains the design of the appropriate behaviour to support the new requirements, and *overriding* the existing design subject with this new design subject. Overriding an existing design subject is specified with a composition relationship between the existing design subject and a new design subject, with an arrowhead at the end of the subject to be overridden.

In the first version of our LMS system, the rules for allowing a borrower take a book are based on a simple check that he has not already reached his maximum allowed (see the `BorrowBook` subject in Fig. 5.). However, for a later version of the system, these rules are extended to also check that the borrower does not have a fine outstanding. In order to make this change, the designer may design this new rule in a separate design subject, and override the existing design with the changes.

With subject-oriented design, it is possible to re-use those parts of a design that are not obsolete, and only override a partial design, if appropriate. In this case, only the rules checking whether a borrower may take a book have been changed to take fines into account. So, the new subject, `CheckBorrow`, need only include in this design the elements which are relevant for changing the borrowing rules – and need *not* include the entire borrowing processing. In Fig. 5, a composition relationship with override integration is specified between the new subject `CheckBorrow`, and `BorrowBook`. The identification of corresponding elements was based on the `match[name]` annotation on the composition relationship. This specifies that component elements within the related subjects that have the same name, correspond. The one exception to this general matching strategy is the name of the operation that handles the checking processing. In `CheckBorrow`, the designer has called this operation `check()`, where the operation to be overridden in `BorrowBook` was called `checkMax()`. This is separately catered for by an additional composition relationship between the two. The result of this composition will include all elements not overridden from `BorrowBook`, all elements from `CheckBorrow`, and the operation `check()` instead of the operation `checkMax()`. Composition semantics ensures that wherever `checkMax()` may have been called previously, `check()` is called in the composed design. Finally, any design elements in the overriding subject that do not have corresponding elements in the overridden subject are added to the composed design. In this case, the `Fine` class with its relationship to the `Borrower` class appear in the result.

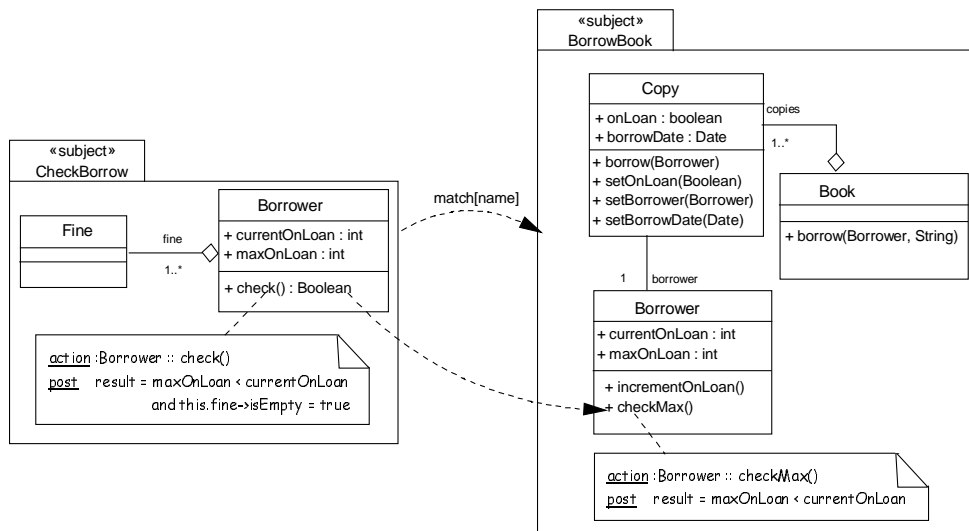


Fig. 5. Example of override integration

### 3.2.2. Merge integration

Merge integration is used when different design models (subjects) contain specifications for different requirements of a computer system. This may have occurred for different reasons. For example, within a system development effort, separate design teams may have

worked on different requirements concurrently. Alternatively, designs may exist for requirements from a previous version of the system. Or, designs may be reused from sources outside a development effort. The full system design is obtained by merging the designs of the separate design subjects.

Looking at some design language constructs from UML, for *classifiers* and *attributes*, merge integration indicates that the composed design contains a single element, whose property values are obtained from the subjects connected by the composition relationship. Where a conflict exists between the property values of corresponding elements, reconciliation strategies may be attached to the composition relationship (see section 3.2.2.1 “Merge with reconciliation required” for a discussion). Merging corresponding *operations* indicates that the specification of the (unified) operation results from the aggregation of the specifications of those operations in all of the related subjects (see section 3.2.2.2 “Merging corresponding operations” for a discussion).

In this section, we look at examples from the LMS of merging subjects that require reconciliation, the impact of merging corresponding operations, and the use of composition patterns for crosscutting functionality.

#### 3.2.2.1. Merge with reconciliation required

For the LMS, merging, for example, the separately designed `AddResource` and `RemoveResource` subjects is specified with a composition relationship between the two. In Fig. 6, this is illustrated with the two-way arc between them. The `match[name]` criteria annotating the composition relationship indicates that those elements within the two subjects which have the same name are deemed to be corresponding. Further specification of corresponding elements is required in this example, as the `AddResource` subject named its class to handle resource management `ResourceManager`, but the `RemoveResource` subject named it `Manager`. This is handled with an additional composition relationship between those elements, denoting that they correspond and should be considered the same class in the result, with their elements merged.

A further look at how the designers of the subjects specified the class `Resource` indicates that the values of their properties are not the same. The impact of merge on classifiers is that a single classifier of the corresponding set of classifiers appears in the result. Since the specifications are not identical, a conflict is encountered which must be resolved. The design process in use is likely to dictate the development team’s approach to choosing a reconciliation strategy in the event of a conflict. For example, it is likely that there will be a level of negotiation and discussion between the designers of different subjects to resolve this conflict. A dissertation on a design process to support this work is outside the scope of this paper. In this case, it has been agreed that precedence will be given to the `AddResource` subject (indicated with the `prec` keyword at the `Kernel` end of the composition relationship), and so the values of the properties of `Resource` in the result are those of the specification in the `Kernel` subject. This reconciliation strategy also applies to the components of the related subjects, and therefore, the name of the class handling resource management is `ResourceManager` in the result. The composition relationship

between Manager and ResourceManager could have defined its own reconciliation strategy to override the one defined by their owners if that was required. Other reconciliation strategies are also possible, and are discussed in Section 4 “UML Metamodel Extensions”.

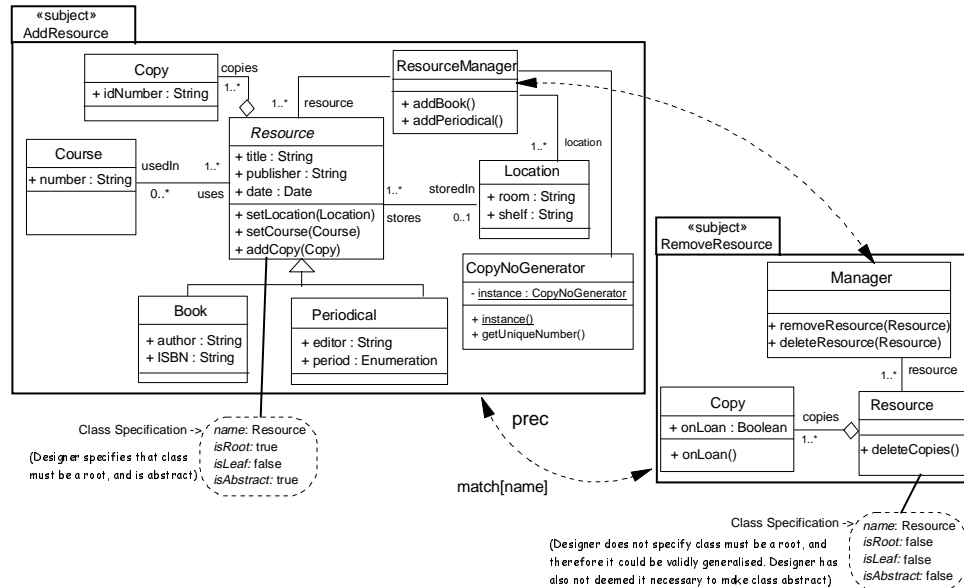


Fig. 6. Example of merge integration with reconciliation

### 3.2.2.2. Merging corresponding operations

On further examination, the requirement that extended the rules for checking whether a borrower may take a book can also be looked on as an addition to the rules, as opposed to a full replacement of the rules. In addition to checking whether the borrower has reached his limit, we also want to check whether he has fines outstanding. The designer may therefore have designed a subject just catering for checking for outstanding fines. Fig. 7. illustrates a CheckBorrow subject that is merged with the BorrowBook subject, denoted by the merge composition relationship between the two. This relationship has a general match[name] criteria for defining corresponding elements, but since the checking operations in both subjects have different names, and additional composition relationship is defined between the them, denoting that they are corresponding.

Merging the specifications of operations in the result implies that all corresponding operations are added to the merged subject. Merge integration means that an invocation of one of the corresponding operations results in the invocation of all corresponding operations. In the example as illustrated in Fig. 8, both checks are executed on invocation of either one. This example illustrates where delegation is used within the composition semantics, where renaming of the actual operations occurs, together with the creation of new operations, check() and checkMax(), used to specify the merged, delegated behaviour.

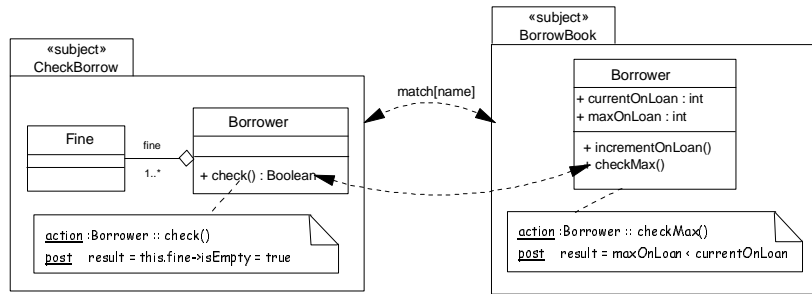


Fig. 7. Merging corresponding operations

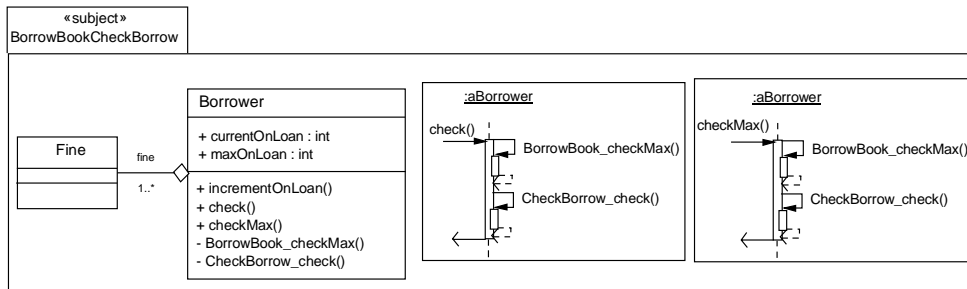


Fig. 8. Output of merging operations specification

Where operations have a return type, the default behaviour is that the value returned by the *last* operation run is returned. Of course, this approach is not sufficient for this checking example, as failure should occur if either of the checking operations fails. Therefore, we need something more. The subject-oriented programming domain supports *summary functions* [15], which synthesise the return values of each of the operations to return a value appropriate for the collaborating operations. This approach is under design for the subject-oriented design domain.

In this example, the collaborations indicating the merged behaviour of the operations is generated automatically, with the ordering of the execution of the operations set arbitrarily. Where the order of execution is important, a collaboration may be attached to the composition relationship indicating that order. If at least one collaboration is attached to a composition relationship, then no additional collaborations are automatically generated. This can be used to restrict the extent to which operations are merged. For example, if it is not the desired behaviour that `CheckBorrow_check()` should be executed every time `BorrowBook_checkMax()` is invoked, then such a collaboration could be excluded from the attachments if at least one other collaboration is defined.

### 3.2.2.3. Composition Patterns

For crosscutting functionality such as synchronising operations, we saw how the use of templates as placeholders for elements with which the crosscutting functionality will interact supports the independent design of such functions. In the case of synchronisation,

Fig. 4 showed write and read operations to be synchronised as separate templates, with the synchronisation behaviour designed around those templates. The subject-oriented design model combines its composition relationship that supports the specification of how different subjects may be integrated to a composed output, with an extension to the notion of the UML's 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. In Figure 9, this is illustrated for the synchronisation functionality of the LMS.

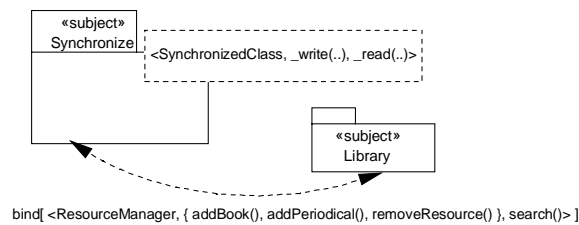


Fig. 9. Specifying the merge of crosscutting behaviour

In this example, a previously composed subject (called `Library`) now containing elements to add, remove and search for resources is merged with the `Synchronize` subject containing operation templates. Specifying how to compose the `Library` 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 synchronization behaviour, and which read and write operations are to be synchronized. In this case, the library's `ResourceManager` class replaces the pattern class in the output, `addBook()`, `addPeriodical()` and `removeResource()` are defined as write operations, and the `search()` operation is defined as read (see Fig. 9).

Pattern specification and binding, as previously illustrated, is all the designer has to do to define truly reusable patterns of crosscutting behaviour, and specify how they are to be composed with designs requiring that behaviour. The composition process, utilising UML template semantics and integration semantics for merging corresponding operations, produces the design illustrated in Fig. 10, where `ResourceManager` now has synchronising behaviour. Note also that the role names in the interactions have been renamed as appropriate.

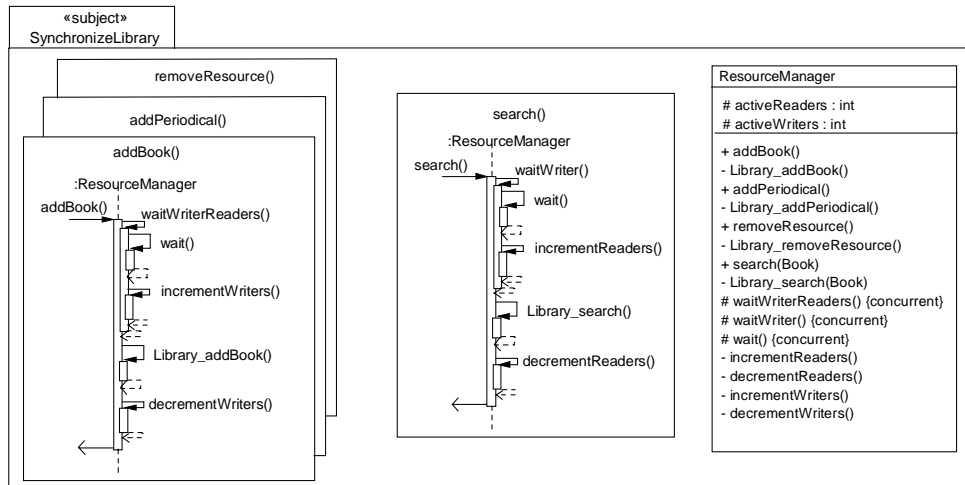


Fig. 10. Output of merged crosscutting behaviour

More examples of the applicability of composition patterns to crosscutting behaviour are demonstrated in [1,3].

### 3.3. Using Subject-Oriented Design

In this section, we look at how subject-oriented design might be used in some different phases within a development process, and discuss the possible complexity of composition specification.

#### 3.3.1. Usefulness throughout the development process

Different phases of software development cycles may gain different kinds of benefits from decomposing design models based on requirements specifications. For example:

- *A new system is under design, and the initial design phase is being planned.* A primary goal from a planning perspective may be to reduce the critical paths of parts of the system. This maximises designer effort by minimising idle time generated by waiting for artefacts on critical paths. By decomposing based on requirements, different requirements may be designed concurrently by different teams. In this situation, the composition requirement is to amalgamate (i.e., merge) all the different designs to build the complete design. The designers may also search for reusable artefacts previously design elsewhere, which might be integrated into the new design effort.
- *New versions of existing systems are required, based on adding new features.* New requirements for additional features are received. As per the initial design effort for previous versions, separating each new requirement into different subjects supports concurrent development, with the composition requirement being to merge the new designs with the previous version.

- *Existing system needs to be ported to different technologies.* For example, a fat client implementation is to be changed to work in a distributed environment. Here, it is likely that the whole design is affected. Even so, the design of the support for the new environment may be separated into a design subject and merged with the existing subjects. Or, if explicit support for a different environment exists in a previous design, then this support may need to be overridden.
- *System change requires are received from test teams (or any interested party).* Here, it has been determined that the behaviour as specified in a design subject does not adequately or correctly support the requirement. A design subject may be designed to correct the inadequacies, with composition required to override the previous effort.

It is not the intent of this paper to impose any particular development process for use with the composition model. This list of possible areas of usefulness throughout the development process is not exhaustive. Different development processes may have different needs in different situations. Since it is not possible to anticipate all the kinds of processes a software development effort may employ, it is the approach of this composition model to support the composition of design models in the most flexible way possible. This is achieved by allowing the sub-division of design models into whatever is most appropriate for the particular development effort, and supporting composition of those models.

### 3.3.2. *Complexity of composition specification*

Composition specification with composition relationships is flexible in the kinds of compositions allowed. Within the context of a single composition specification, multiple composition relationships may be specified between elements within the design subjects, with the same elements possibly participating in multiple different relationships. This flexibility means that the suite of composition relationships within the context of a composition to a single output could get quite complex. Where some cooperation exists between the design teams of subjects with potentially considerable overlap, composition specification could be as simple as a single composition relationship between input subjects. In this case, with some communication, there may be few differences in the overlapping areas. On the other hand, one of the benefits of this approach is the support for design teams working concurrently with, potentially, little or no contact between them. Taken to the extreme, this might result in considerable differences in the specifications of overlapping concepts. This situation would require multiple composition relationships to specify the overlapping concepts' resolution and integration. In this case, composition specification becomes more complex. For each software development project using the subject-oriented design model, a balance should be found between increasing the level of communication between different design teams and thereby decreasing the level of complexity of composition specification versus totally isolating the design teams, thereby increasing the likelihood of more complex specifications. Depending on the personnel make-up of the team in terms of levels of experience and knowledge, and on the physical locations of the different teams, different choices may be appropriate. In addition,

experience with using the model will provide assistance in both determining an appropriate extent of isolation of teams.

#### 4. UML metamodel extensions

The UML is the OMG's standard language for object-oriented analysis and design specifications [10]. The OMG currently defines the language using a *metamodel*. The metamodel defines the syntax and semantics of the UML, and is itself described using the UML [10]. The metamodel is described in a semi-formal manner, using the views:

- **Abstract syntax:** This view is a UML class diagram showing the metaclasses defining the language constructs (e.g. Class, Attribute, Operation, Association etc.), and their relationships. An informal description in natural language describes each of these constructs and their relationships. The class diagrams include multiplicity and ordering constraints.
- **Well-formedness rules:** A set of well-formedness rules, each of which has an informal description and an OCL definition, describes the static semantics of the metamodel, specifying constraints on instances of the metaclasses – i.e. the usage of the UML language constructs.
- **Semantics:** The meanings of the constructs in the language are described using natural language.

In this paper, we use a similar style to that of the UML to describe the syntax and semantics of composition relationships. This section contains a brief flavour of the semantics of the model with:

- UML class diagrams describing the constructs for the composition relationships.
- Well-formedness rules for composition relationships.
- Descriptions of the semantics of composition relationships with merge and override integration.

##### 4.1. Composable elements

First, it is necessary to define the kinds of design elements that may participate in a composition relationship. Currently, the scope of this work is the structural and collaboration diagrams. The style for restricting the kinds of model elements that may participate in compositions is similar to the way that the UML defines the model elements that may participate in generalization relationships. In the UML, an abstract construct called `GeneralizableElement` exists, from which any model elements that may participate in a generalization inherits. Similarly, a new abstract construct, called `ComposableElement` is created here to define which model elements may participate in a composition relationship (see Figure 11).

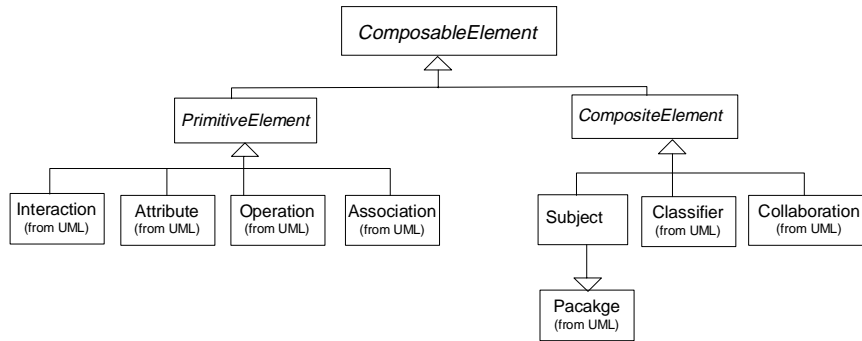


Fig. 11. Composable elements

Composable elements are divided into two types: primitives and composites. The impact of integration on design elements depends on the kind of elements they are. For example, the following operation specification is of a protected operation named `op1` with two parameters:

```
# op1(p1 : Integer, p2 : String)
```

Another subject has a specification for `op1` as a public operation with three parameters:

```
+ op1(p1 : Integer, p2 : String, p3 : Boolean)
```

Overriding the first `op1` specification with the second results in an operation specified as public, with three parameters as defined by the second specification. Some rules have been defined as to the kinds of operations that may be merged (see section 4.4.1 Semantics). It is the subject of further research as to the need for similar restrictions for override integration. Notwithstanding, this example illustrates how the full specification of an operation is overridden, and so, in this sense, operations are *primitives*. Primitives are defined as elements whose full specifications are composed with other primitives. For the purposes of composition, the following elements are considered to be primitives: Attributes, Operations, Associations and Interactions.

There are, however, some elements that contain other elements, and cannot be considered as primitive. For example, a class contains attributes and operations, and those attributes and operations must be examined individually for correspondence matching and integration. Such elements are called *composites*. Composites are defined as elements whose components are not considered part of the full specification of the composite and therefore are considered separately for composition. Composites may contain composites and primitives, and therefore may be considered as a standard tree structure. For the purposes of composition, composites are: Subjects, Classifiers and Collaborations.

## 4.2. Composition Relationship

Composition relationships are defined between composable elements. Elements are composed with their *corresponding* elements. Elements are said to “correspond” when they “match” for the purposes of composition, where correspondence matching specification is part of the composition relationship. A composition relationship is a new kind of relationship for the UML, and is subclassed from the Relationship metaclass (see Figure 12). Composition entails synthesising two or more input subjects into an output subject. Identifying inputs to a composition must first involve identifying the input subjects, and specifying a composition relationship between those subjects – this composition relationship is considered to be the *contextual* composition relationship that defines the context for any additional composition relationships specified between elements inside the input subjects (i.e., levels lower in the tree structure mentioned in the previous section).

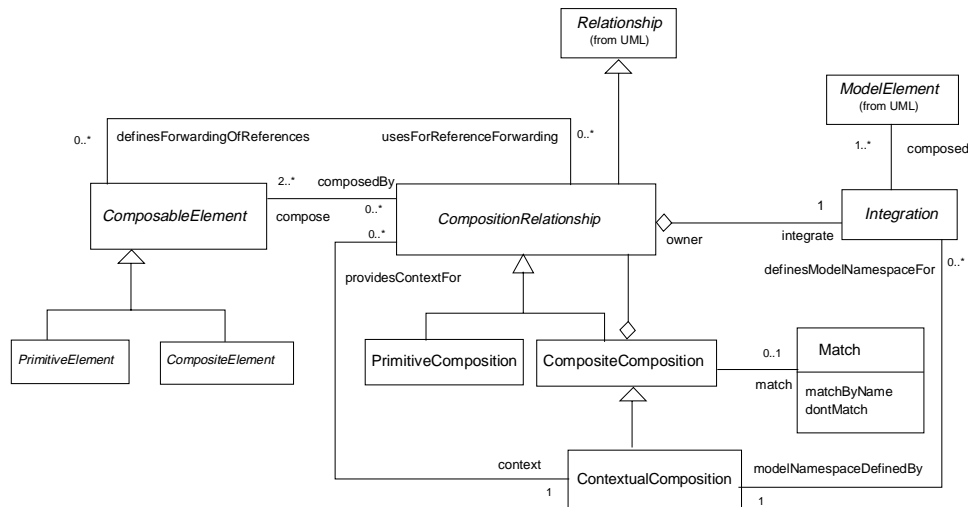


Fig. 12. Composition Relationship

A primitive composition relationship is a composition relationship between primitives, and a composite composition relationship is a composition relationship between composites. Composable elements explicitly related by a composition relationship are said to correspond, and are integrated based on the semantics of the particular integration strategy attached to the composition relationship. A more general approach to identifying corresponding elements is possible with the Match metaclass, which supports the specification of match criteria for components of composite elements related by a composition relationship. Integration as specified in Figure 12 is an abstract metaclass, where it is intended that it be specialised to cater for the particular integration specification required. The default semantics for integration strategies is that a composition process results in the composition of elements to new model elements, as defined by the “composed” relationship from the Integration metaclass.

In some cases, integration of design elements results in changes to an element in an output subject – for example, override integration changes the specification of the overridden element to that of the overriding element. Elements which reference such an element in an input subject may therefore, when themselves copied to the output, have a difficulty because their referenced element has changed. The composition model therefore supports the notion of “forwarding” of references to changed elements to resolve this difficulty. Where elements are involved in multiple compositions, the designer may explicitly state the composed element to which references will forward.

Some examples (not all are shown for space reasons) of well-formedness rules for composition relationships are:

- A contextual relationship is only defined between subjects.

```
context cr : CompositionRelationship
cr.ocIsKindOf(ContextualComposition) implies
    cr.compose.forAll( c | c.ocIsKindOf(Subject))
```

- All kinds of composition relationships other than the contextual composition relationship are defined with a context relationship to a contextual composition relationship

```
context cr : CompositionRelationship
cr.ocIsTypeOf(PrimitiveComposition) or
    cr.ocIsTypeOf(CompositeComposition) implies
    not cr.context.isEmpty
```

- Composition relationships may only be specified between design elements of the same type

```
context cr : CompositionRelationship
cr.compose->forAll (c1, c2 | c1.ocType = c2.ocType)
```

- For each of the input design elements to a composition relationship, the subject in which that design element is contained must participate in the contextual relationship that defines the context of the composition relationship

```
context cr : CompositionRelationship
cr.compose->forAll (c |
    cr.context.compose->exists( s |
        c.owningSubject = s))
```

- A composition relationship specified between input subjects defines the namespace for composed elements in an output subject.

```
context cr : CompositionRelationship
cr.integrate.composed->forAll (outEl |
    cr.context.compose->forAll(s |
        outEl.namespace =
            s.namespace.concat(outEl.namespace)))
```

- Within a single composition context, one composition relationship must be defined as the one specifying the result to which all referring elements forward.

```

context cc : ContextualComposition
cc.inputComposableElements->forall (cEl |
  exists(cr | CompositionRelationship |
    cr.definesForwardingOfReferences.includes(cEl)
    and cr.context = cc))

```

The additional operations required to support all well-formedness rules discussed in this paper (such as `compose`, which returns the set of related elements, etc.) may be found in [1].

#### 4.2.1. Semantics for Identifying Corresponding Elements

- Correspondence between elements is established either directly with a primitive composition relationship, or indirectly based on matching from the specification of the relationship between their owners. Correspondence between elements is not possible where the elements are components of non-corresponding composites.
- Elements that participate in a composition relationship with a `dontMatch` specification do not correspond.

Integration semantics are defined by subclasses to the Integration metaclass.

#### 4.3. Override Integration

Override integration is used when an existing design subject needs to be changed. A composition relationship identifies the subject to be overridden, and the overriding subject. Override integration is subclassed from the Integration metaclass (see Fig. 13).

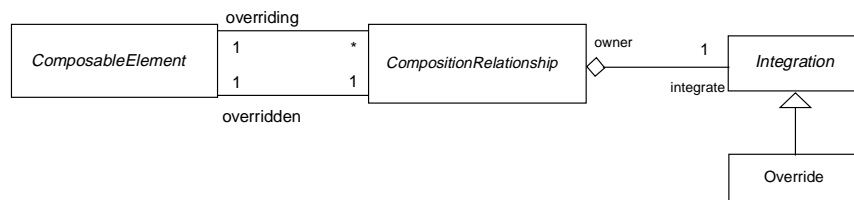


Fig. 13: Override integration

Some examples of well-formedness rules for override integration are:

- The composition relationship to which override is attached relates composable elements based on its overridden and overriding associations only.

```

context or : Override
or.owner.compose =
  or.owner.overriding->union(or.owner.overridden :
    ComposableElement) : Set(ComposableElement)

```

- The overriding and overridden elements are different.

```
context or : Override
or.owner.overridden <> or.owner.overriding
```

- Within the context of a single composition, a composable element may only participate in one composition relationship as the overridden element.

```
context or : Override
or.owner.compose.providesContextFor->forall(cr1,cr2 |
    cr1 <> cr2 implies cr1.overridden <> cr2.overridden
```

#### 4.3.1. Semantics

This section summarises the general semantics for overriding design elements that apply to each element type. The impact of override integration on all supported kinds of model elements is more fully defined in [1].

- For each element in the overridden subject, the existence of a corresponding element in the overriding subject results in the specification of that element to be changed to that of the corresponding element in the composed result.
- Elements in an overridden composite that are not involved in a correspondence match remain unchanged, and are added to the composed result.
- Elements that are components of an overriding composite and are not involved in a correspondence match are added to the composed result.
- Overriding elements may not result in name clashes in the resulting subject – renaming will occur.
- The resulting subject must conform to the well-formedness rules of the UML.

#### 4.4. Merge Integration

Merge integration is used when different design models (subjects) contain specifications for different requirements of a computer system. Merging design subjects is done by specifying composition relationships, with merge integration, between the subjects to be merged.

Composition relationships identify the subjects to be merged, and the corresponding design elements within those subjects that should be merged. For many elements (for example, classifiers and attributes) this means that the corresponding elements appear once in the merged result. In cases where differences in the specifications of corresponding design elements needs to be resolved, composition relationships with merge integration specify guidelines for the reconciliation. The metaclass class diagram in Figure 14 illustrates merge integration as a subclass of the Integration metaclass attached to the composition relationship metaclass, and also illustrates the metaclasses required to support the reconciliation of elements.

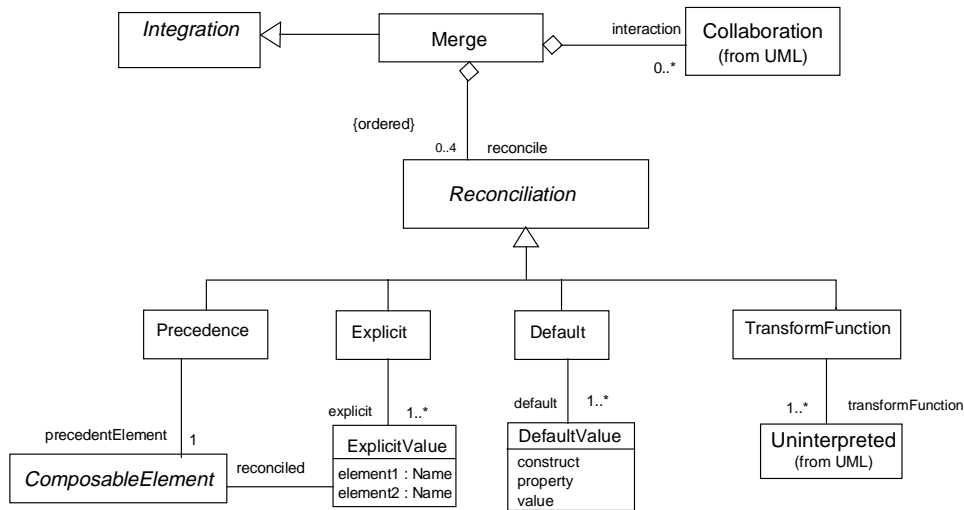


Fig. 14: Merge Integration

When corresponding operations are merged, all of the merged operations are executed on receipt of a message that may have activated one of the operations in an input subject. Collaborations may be attached to the merge relationship to determine the order of execution.

Examples of well-formedness rules are:

- Reconciliations attached to a composition relationship apply to all elements *except* operations, constraints and collaborations
- There may be only one of each of the kinds of reconciliation in the ordered set of reconciliations attached to a composition relationship with merge integration. For example, only one precedent element is possible.
- All operations in a corresponding set must be referenced in any collaboration specifying the order of execution for that corresponding set. Not all operations have to be realised by collaborations.
- Merging operations must have the same parameter list with one exception. Where one of the corresponding operations has parameters whose values may be used in other corresponding operations with a *subset* of the parameters in the called operation, these operations may be defined as corresponding. In this case, the designer must attach a collaboration to the composition relationship indicating how the operations are called.

#### 4.4.1. Semantics

This section summarises the general semantics for merging design elements. The impact of merge integration on all supported kinds of model elements is more fully defined in [1].

##### 4.4.1.1. Corresponding Operations:

- Corresponding operations are each added to the merged subject. Operations with conflicting properties (such as visibility) are deemed to be non-corresponding.

- Where no collaboration is attached to a merge integration specification, the behaviour of the merged subject in relation to the merged operations is specified with a new collaboration specification. This collaboration specifies that an invocation of one of the corresponding operations results in the invocation of all corresponding operations.
- When the order of execution of corresponding operations is important, a collaboration specifying this order should be attached to the merge integration. In this case, the attached collaboration is added to the merged subject as the specification of the behaviour of corresponding operations.

#### *4.4.1.2. Elements other than Operations:*

- For all elements other than operations, corresponding elements appear once in the merged subject.
- Any conflicts in the specifications of corresponding elements are reconciled prior to addition to the merged subject. Reconciliation strategies may be attached to the merge integration specification as follows:
  - One subject's specifications take precedence in the event of a conflict
  - A transformation function may be attached to a merge relationship that, when run against the conflicting elements, results in a reconciled element.
  - An explicit specification of the reconciled element may be attached to a merge relationship.
  - Default values may be specified which are to be used in the event of a conflict in specific properties of elements.

#### *4.4.1.3. General Semantics of Merge for All Elements:*

- Elements that are components of merging composites and are not involved in a correspondence match are added to the composed result.
- Merging elements may not result in name clashes in the resulting subject.
- The resulting subject must conform to the well-formedness rules of the UML.

### *4.5. Composition Patterns*

The composition patterns model, at the specification level, differs from the UML templates model in two primary ways, as illustrated in Fig. 15.

1. Templates within a composition pattern are centred around pattern classes (placeholder classes to be replaced by "real" class elements) within a subject first, which may have additional template parameters defined.
2. Binding actual classes and model elements from (an)other subject(s) is achieved with an extension to a composition relationship with merge integration defined. This composition relationship's arguments define which classes replace the pattern classes, and which elements within the replacing classes replace a pattern class's template parameters.

Examples of well-formedness rules are:

- Only one subject involved in a single contextual composition relationship may contain template elements
- Only a contextual composition relationship may have a pattern match integration defined – that is, when the composition relationship is between two or more subjects
- All templates must have at least one replacement defined.
- Replacements defined for pattern classes must be contained within the subject input to the composition.

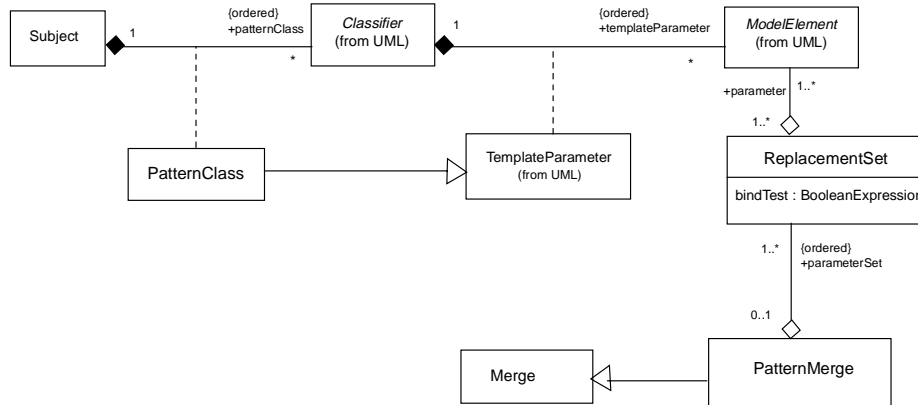


Fig. 15: Composition patterns metamodel

#### 4.5.1. Semantics

With integration when patterns are involved, corresponding operations within replacing classes are substituted for templates defined in pattern collaborations. For each instance of the collaborating pattern, a collaboration is added to the merged subject that defines the execution of the actual corresponding operations. Use of templates in collaborations, together with composition relationships defining the binding elements, support the merge of *crosscutting* operations – that is, operations designed to supplement the behaviour of multiple operations in a subject, and therefore may be considered as a pattern.

## 5. Related work

There is a rapidly growing realisation that decomposition of object-oriented systems by class is necessary, but not sufficient for good software engineering. Many approaches have provided improved decomposition capabilities in object-oriented software, and we look at approaches at the design level particularly, though many interesting approaches also exist at the implementation level.

The UML itself [10] contains a small number of mechanisms that could be used to separate different elements that support different requirements. For example, attributes and operations may be organised within classes using stereotypes to group them for particular

needs. In addition, multiple models of the same kind (e.g. multiple object models or multiple class models) may be defined within the same package that could be used to provide a limited measure of separation based on requirements. This support is limited for overlapping concepts (concepts that support multiple requirements) because, using UML, design elements that support the same concept, but have different views that necessitate different specifications, must be specified separately. Since there is no means of synthesising a complete design of incomplete pieces in UML, such elements will remain separate throughout the design cycle. Multiple generalization is another mechanism that could be used to combine multiple different structural and behavioural properties, designed to support different requirements. However, there are some difficulties with using this technique in an attempt to separate support for different requirements into different classes. First, separation based on multiple generalization is not possible when there are overlapping concepts that support multiple requirements. Another issue is the practicality of the approach based on the possibilities relating to an explosion of the class hierarchy for each new requirement added.

Conceptually, the subject-oriented design and composition patterns model has evolved from the work on subject-oriented programming [6,11]. 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).

The goals of subject-oriented design and those of the role modelling work from the OORam software engineering method [13] are similar. OORam shows how to apply role modelling by describing large systems through a number of distinct models. Derived models can be synthesised from base role models, as specified by *synthesis relations*. Synthesis relations can be specified both between models and between roles within the models, much like our composition relationships. The synthesis process is equivalent to the synthesis of subjects defined with a merge interaction specification. The subject-oriented design model distinguishes itself with its notion of *override* integration, and more particularly, with the potential provided by composition patterns to provide for more sophisticated, complex possibilities for combination patterns.

Separation of concerns with Catalysis [4] is based on UML, using *horizontal* and *vertical slices* to separate a package's contents according to concerns. Composition of artefacts is based on a definition of the UML *import* relationship, called *join*. The designer is instructed to form a new design containing the simple union of design elements, with re-naming in the event of unintended name clashes. This approach is similar to the meaning of a merge interaction specification with property matching by name. Catalysis encourages a design strategy in which an initial design is gradually modified to produce a completed one, which is a single, fully integrated design. The subject-oriented design model supports a design strategy in which pieces (subjects) are identified and designed separately, and may remain separate in the completed design, though related by composition relationships. This enhances the traceability of requirements that lead to various artefacts. For example,

Catalysis describes the rules and decisions a designer should (might) follow to form the result of joining two packages, while the original packages in subject-oriented design are retained. Instead, a way of specifying the rules and decisions as annotations on the composition relationship(s) relating them is defined. Reusable design components are supported in Catalysis with template frameworks, containing placeholders that may be imported, with appropriate substitutions, into model frameworks. This is similar to merging reusable design subjects that have no overlapping design elements, possibly using composition patterns.

From the perspective of crosscutting requirements, the subject-oriented design with composition patterns model also closely relates to the aspect-oriented programming model that separates crosscutting behaviour into separate “aspects” [9]. There are some interesting design approaches that are rooted in the aspect-oriented programming paradigm. 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. synchronization [7] or command pattern [8]). In such approaches, the constructs required by each particular aspect are stereotyped so that a weaver (like a composer) can determine which elements match the appropriate aspect construct. In both these examples, many of the behavioural details of synchronization 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 [14] 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 crosscutting behaviour in a manner that is independent of a particular programming environment.

## 6. Conclusions

Standard object-oriented designs do not align well with requirements. Requirements are typically decomposed by function, feature or other kind of user-level concern, whereas object-oriented designs are always decomposed by class. This misalignment results in a host of well-known problems, including weak traceability, poor comprehensibility, poor evolvability, low reuse and high impact of change. In this paper, *subject-oriented design* is described as a means of achieving alignment between requirements and object-oriented designs, and hence alleviating these problems. This alignment is possible because requirements criteria can be used to decompose subject-oriented designs into *subjects*, which can then be synthesised as specified by composition relationships. Subjects can be designed independently, even if they interact or cut across one another.

One of the strengths of the subject-oriented design model is its support for the design of overlapping concepts from different perspectives in different subjects. The model supports differences in the specifications of those overlapping parts, with techniques to handle

conflicts. In addition, references to elements that are composed with other elements (and therefore, potentially changed in some way) are changed to refer to the composed element. For example, if an attribute has a classifier type which is integrated with another type, then the attribute's type refers to the integrated type. However, with composition of this nature, there is potential for the designer to specify a result that may not be well-formed from a UML perspective. For example, a root class with subclasses defined in one subject may be overridden with a class which is defined as a leaf class. This clearly is incorrect, and the result is ill-formed. There are many such examples across all the UML constructs. The current subject-oriented design model allows considerable freedom to the designer to identify corresponding elements, and compose, as desired, with breakages to the well-formedness rules highlighted in the result. This was deemed to be the most flexible approach, as many of the potential difficulties will require domain knowledge to decide the appropriate course of action. Using this model, the designer can test different possibilities. This approach can be extended, though, to explicitly list restrictions to the possibilities for composition based on the values (or combination of values) of all properties of design language constructs. For example, such a rule might state that public and private operations may not be merged. There is also some scope for providing support to avoid inheritance cycles in the output model. There has been some work in the area of eliminating cycles in composed hierarchies [16], which could be incorporated here. The approach is based on separating a type hierarchy from the implementation hierarchy in the input subjects, while only maintaining the generalizations in the type hierarchy.

Remaining work includes defining the impact of composition on all UML design models, and providing tool support for decomposition in this way, together with support for composition specification. In addition, a formal algebra defining the rules associated with multiple compositions of multiple subjects may be useful. Automation of the link from subject-oriented design to subject-oriented programming is also an area worth pursuing. This includes automating the generation of subject code from subject designs, and generation of composition rules from composition relationships. Another possibility for environment support is the extent to the subject-oriented design model may be used as a design model for aspect-oriented programming. Similarities in goals between the approaches provide considerable encouragement for links between them.

To date, the subject-oriented design approach has been only tested on small examples. One of the most important priorities for further work is to test the approach against large-scale, industrial projects. We consider it likely that details of the subject-oriented design approach will evolve through experience with its usage. Such a test will also serve to assess the impact of the subject-oriented design model on the full software development process.

## **Acknowledgements**

Many thanks to Robert J. Walker, Peri Tarr, Harold Ossher, William Harrison, Stuart Kent and the anonymous reviewers.

## References

- [1] S. Clarke. Composition of Object-Oriented Software Design Models. PhD Thesis, Dublin City University, January 2001
- [2] S. Clarke, W. Harrison, H. Ossher, P. Tarr. Subject-Oriented Design: Towards Improved Alignment of Requirements, Design and Code. In Proc. OOPSLA 1999
- [3] S. Clarke, R.J. Walker. Composition Patterns: An Approach to Designing Reusable Aspects. In Proc. ICSE 2001
- [4] D. D'Souza, A. C. Wills. Objects, Components and Frameworks with UML. The Catalysis Approach. Addison-Wesley, 1998
- [5] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994
- [6] W. Harrison, H. Ossher. Subject-Oriented Programming (a critique of pure objects). In Proc. OOPSLA 1993
- [7] J.L. Herrero, F. Sánchez, F. Lucio, M. Toro. Introducing Separation of Aspects at Design Time. In Proc. Aspects and Dimensions of Concerns workshop at ECOOP 2000
- [8] W-M Ho, F. Pennaneac'h, J-M Jezequel, N. Plouzeau. Aspect-Oriented Design with the UML. In Proc. Multi-Dimensional Separation of Concerns workshop at ICSE 2000.
- [9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M Loingtier, J. Irwin. Aspect-Oriented Programming. In Proc. ECOOP 1997
- [10] OMG. The Unified Modeling Language Specification. Version 1.3. June 1999
- [11] H. Ossher, M. Kaplan, A. Katz, W. Harrison, V. Kruskal. Specifying Subject-Oriented Composition. In Theory and Practice of Object Systems, Vol. 2(3) 1996
- [12] D.L. Parnas. On the criteria to be used in decomposing systems into modules. In Communications of the ACM, Vol. 15(12):1053-1058 1972
- [13] T. Reenskaug, P. Wold, O.A. Lehne. Working with Objects. The OOram Software Engineering Method. Manning Publications Co. 1995
- [14] J. Suzuki, Y. Yamamoto. Extending UML with Aspects: Aspect Support in the Design Phase. In Proc. Aspect-Oriented Programming workshop at ECOOP 1999.
- [15] P. Tarr, H. Ossher. Hyper/J<sup>TM</sup> User and Installation Manual. Available from <http://www.research.ibm.com/hyperspace>.
- [16] R.J. Walker. Eliminating Cycles in Composed Class Hierarchies. Technical Report TR-00-07, Department of Computer Science, University of British Columbia, 2000