# Composition of Object-Oriented Software Design Models

Siobhán Clarke, B.Sc. in Computer Applications (Hons)

Thesis Submitted for the Degree of

Doctor of Philosophy in Computer Applications

School of Computer Applications
Dublin City University

Supervisor
Dr. John Murphy

January 2001

# Table of Contents

# List of Figures

# Abstract

In practice, object-oriented design models have been less useful throughout the lifetime of software systems than they should be. Design models are often large and monolithic, and the structure of designs is generally quite different from that of requirements. As a result, developers tend to discard the design, especially as the system evolves, since it is too difficult to keep its relationship to requirements and code accurate, especially when both are changing. This thesis identifies a number of key, well-defined problems with current object-oriented design methods and proposes new techniques to solve them.

The new techniques present a different approach to designing systems, based on flexible decomposition and composition. The existing decomposition mechanisms of object-oriented designs (based on class, object, interface and method) are extended to include decomposing designs in a manner directly aligning design with requirements specifications. Composition mechanisms for designs are extended to support the additional decomposition mechanisms. The approach closely aligns designs with both requirements specifications and with code. It is illustrated how this approach permits the benefits of designs to be maintained throughout a system's lifetime.

# Preface

## Statement of Contribution

The author based the ideas relating to extending the decomposition and composition capabilities of the UML on the previously published work on subject-oriented programming from IBM Research. Having worked on the application of the ideas to the design phase for a time without contact with the subject-oriented programming team, the foundations of the work took notable shape when worked on collaboratively with the IBM Research software composition group, led by Harold Ossher, at the IBM T. J. Watson Research Center in Hawthorne, New York. In particular, the author worked most closely with Peri Tarr in moulding the work, and defining its shape, at a high level. This collaborative work culminated in a number of publications, in particular [Clarke et al. 1999a]. Participation in a number of workshops in that year explored subject-oriented design's application to the problems of multi-dimensional separation of concerns [Clarke et al. 1999b], software evolution [Clarke et al. 1999c], [Clarke et al. 1999e] and separation of cross-cutting concerns [Clarke et al. 1999d]. The author benefited greatly from discussions with many different people at these workshops.

In addition to those publications mentioned above, the author produced the following publications prior to this thesis. Introductions to the changes made to the UML metamodel to support composition relationships are contained in [Clarke 2000a] and [Clarke 2000b]. A description of the composition patterns model is contained in [Clarke 2000c]. Early ideas on how to resolve conflicts between corresponding elements are described in [Clarke & Murphy 1998a]. Early ideas on composing design models were also presented at a number of workshops, where again, the author benefited from discussions with many different people. Position papers for these workshops are contained in [Clarke & Murphy 1998b], [Clarke & Murphy 1998c] and [Clarke & Murphy 1997]. In all cases, this thesis should be regarded as the definitive account of the work.

# Acknowledgements

This thesis could not have happened without the support of many people. First, John Murphy. As my supervisor, he provided constant encouragement with his unwavering belief in me, and his willingness to provide me with the freedom and guidance to pursue my ideas. As my friend, he has been unfailingly supportive from the moment he put the idea of a PhD into my head.

I am indebted to Rob Walker (University of British Columbia) for his in-depth reviews of early drafts of this thesis, and numerous detailed discussions and suggestions about the finer points of the “subject” approach. I have no doubt that this is a better thesis because of him. Thanks also to Renaat Verbruggen (Dublin City University) for reviewing early drafts, and lending a sense of reality to the approach.

I believe that the most pivotal and influential period was the three months I was privileged to spend in the IBM T.J. Watson Research Center. I am grateful to Harold Ossher for the opportunity (initiated by Stuart Kent from the University of Kent) of working in his software composition group. While there, I worked most closely with Peri Tarr in defining the “subject” ideas for design. And I had a great time too! Thanks, Peri! I am also grateful to Bill Harrison for igniting my early interest in subject-oriented programming, and for numerous interesting discussions.

I received so much from many friends in IBM Ireland Ltd. Paul Murphy and Greg Scollan provided advice and encouragement in the early stages. Pat O’ Connor provided funding for conference travel. Emer MacDowell, Paul McDaid, Carol Smith and Donal Sullivan embroiled themselves in interesting discussions about my work. John O’Sullivan and James Rush gave invaluable information development tips. Thanks also to Regina, Catherine and Paula.

I am grateful to Andrew Butterfield from Trinity College, and Mel Ó Cinnéide from University College Dublin, for organising seminars for me that gave me the opportunity to discuss the ideas.

I was funded throughout by Dublin City University, and Padraic Moran.

Thanks to my family and other friends for putting up with me, and giving lots of very useful advice, especially my parents, Ursula, Regina, Aidan, Niamh, Anne, Gráinne, Paula, Winnie and Ian.

Finally, there is one person who was a constant support in every conceivable way. I couldn’t even begin to itemise them. This thesis is dedicated to my husband, Padraic, with all my love.

# Chapter 1: Introduction

Software design is an important activity within the software lifecycle and its benefits are well documented ([Booch 1994], [Coleman et al. 1994], [Cook & Daniels 1994], [Jacobson et al. 1992], [Rumbaugh et al. 1991], [Shlaer & Mellor 1988]). The benefits include early assessment of technical feasibility, correctness and completeness of requirements; management of complexity and enhanced comprehension; greater opportunities for reuse; and improved extensibility. The object-oriented design paradigm has become the standard approach throughout the software development process, but many issues remain open for research into improving its effectiveness against these benefits [Engels & Groenewegen 2000].

## *Current Issues with Object-Oriented Modelling*

In [Engels & Groenewegen 2000], a broad range of issues associated with current object-oriented modelling techniques are discussed. This work represents the most up-to-date view of areas requiring research. The issues are dealt with in the context of the Unified Modeling Language (UML) as it is the current standard language for object-oriented modelling, as defined by the OMG [UML 1999]. Currently open issues range across a number of different categories: 1) issues associated with the UML as a language, with assessments on its architecture, notation, completeness and semantics; 2) issues with the modelling units of the UML and their interdependencies; 3) issues with model composition techniques; 4) issues with the modelling process, with consideration for consistency, coordination and communication; 5) issues with the reviewing techniques available, for example, animation, simulation and analytical techniques; and 6) issues with embedding object-oriented modelling into the full software development process, with round-trip engineering and support tools among the cited concerns.

## *The Problems Addressed in this Thesis*

This thesis addresses a very important subset of the issues raised in the aforementioned paper. In particular, the **modularisation** (or *decomposition*) capabilities of object-oriented modelling units, and object-oriented model

**composition** capabilities, are addressed. As can be seen by the list of issues raised, many of the benefits of software design are not being realised within the object-oriented paradigm. Within this thesis, the need to realise more of the benefits of software design is an ultimate goal. Problems with current techniques are assessed based on their capabilities relating to management of complexity and enhanced comprehension, greater opportunities for reuse, and improved evolvability. As illustrated in this thesis, modularisation and composition capabilities are key to realising these benefits, and therefore become the focus for the research described in this thesis.

First, let us consider modularisation. Object-oriented modelling modularisation is based on the notion of class and object, which encapsulate structural properties defined by attributes and behavioural properties defined by operations and methods. This thesis illustrates that the limited modularisation catered for by the object-oriented paradigm is insufficient to support readily understandable models. This insufficiency impacts the ease with which models may change as the design evolves, and also impacts the opportunities for reuse.

## *Failure of Existing Approaches*

For example, a brief look at the limited modularisation capabilities of the object-oriented paradigm shows that the units of modularisation are structurally different from the units of modularisation of requirements specifications (see “Chapter 2: Motivation” on page 11 for more details). Requirements are specified based on the features and capabilities required of the software system. Evidence of the structural difference between this kind of modularisation and of object-oriented classes and methods is manifested in how the design of a single requirement generally needs multiple classes and methods to support that requirement, and also, how an examination of most object-oriented classes demonstrates that they support multiple different requirements. From a comprehension point of view (one of the key goals for software design), this means that understanding a single requirement needs an understanding of multiple classes across a design, and understanding a single class needs comprehension of multiple requirements.

So, what about extensibility, another of the key goals? Consider a situation where a new requirement is received. Adding the design of this new requirement may be as simple as adding a new class, with no impact on any existing class, but it is easy to imagine that this is often not the case (examples are illustrated in this thesis). In many cases, designing support for a new requirement will involve changing many of the existing classes and methods. This means that the details of all the existing classes, and the impact of all those changes must be clearly understood. This level of invasive change to the existing design is not compatible with the goal of a design that is easily changeable.

Finally, how does standard object-oriented modularisation fare when it comes to re-use? The structural mismatch previously discussed between units of modularisation in requirements specifications and in object-oriented specifications noted that an examination of a class demonstrates support for multiple different requirements. Classes, therefore, often include much more functionality than any given client would use, which decreases comprehensibility and, potentially, usability.

Other approaches exist that improve the modularity of object-oriented designs. For example, design patterns attempt to isolate different parts of a design into separate units, thereby attempting to improve understandability and extensibility [Gamma et al. 1994]. However, as illustrated in "Chapter 2: Motivation" on page 11, and indeed, discussed for each of the patterns in [Gamma et al. 1994], design patterns have their own difficulties. For example, usage of each pattern must be pre-planned and included in the design, as retrofitting any pattern once the design is complete may require multiple changes across the existing design. This is a problem, as it is not possible to anticipate all the changes that may be required of a system, and therefore to anticipate the best patterns to be included in a design.

In "Chapter 3: Related Work" on page 37, other approaches to improving modularisation across the software development lifecycle are examined. There are some approaches discussed that yield ideas that are adapted for the research documented in this thesis, and other approaches which have limitations that influence the direction of this research.

In this thesis, composition is discussed in the context of the capabilities required to support new modularisation (or decomposition) approaches.

## *Proposed Solution*

This thesis proposes a new approach to object-oriented design that extends the modularisation capabilities currently available. Current object-oriented modelling techniques support decomposition of design elements by class, attribute, operation and interface. Groupings of classes into *packages* are currently available, where a package is simply a "grouping of model elements" [UML 1999]. As discussed previously, the structural difference between the way that requirements are specified/modularised and the way that object-oriented designs are specified/modularised causes difficulties in comprehension, reuse and extensibility. This thesis directly addresses this structural mismatch by adding **decomposition** capabilities that support structural matching of design models with individual requirements specifications. Corresponding **composition** capabilities are included in this new approach, where separate design models may also be integrated.

The approach to modularisation and composition described in this thesis is primarily based on a similar approach to modularisation and composition of object-oriented programming models, called *subject-oriented programming* [Ossher et al. 1996]. Throughout this thesis, the research described will be referred to as the *Subject-Oriented Design Model*, or subject-oriented design.

## Decomposition

The basis of the subject-oriented design approach to decomposition is that separate object-oriented design models may be specified for each individual requirement. This 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 in object-oriented models. That is, an examination of many classes in object-oriented models require an understanding of multiple different requirements in order to fully understand each class, and indeed, to understand multiple collaborating classes. The subject-oriented design model recognises and explicitly caters for this level of overlap in the different design models for each requirement. This is achieved by allowing each 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.

• *Cross-cutting 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 *cross-cutting* [Kiczales et al. 1997], since support for such requirements must be included across many different objects in a system. With the approach to decomposition proposed in this thesis, cross-cutting requirements may also be designed separately, with composition capabilities handling their integration with other system objects as appropriate.

Standard object-oriented design language constructs may be used within the individual design models modularised to support separate requirements. In other words, the new design approach proposed within this thesis does not require any new notations for the separate design models.

Composition

Corresponding composition capabilities are required to support the new kinds of decomposition proposed in this thesis. In order to verify the separated design models, and understand the implications of all the design models for the full system, composition of the design models is required. This thesis defines a new 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 should therefore 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 the 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. These two particular integration strategies are described in detail in this thesis (see “Chapter 6: Override Integration” on page 127 and “Chapter 7: Merge Integration” on page 155). However, other integration strategies are possible, and so this thesis discusses how new integration strategies may be added to this approach.

- *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.

Composition relationships are a new kind of design construct. This thesis uses the UML as the sample object-oriented design language on which to illustrate the decomposition and composition capabilities of the model described in this thesis. As such, extensions to the UML metamodel to incorporate this new design construct are included in “Chapter 5: Composition Relationship: An extension to the UML Metamodel” on page 109.

Composition Patterns

For design models that support *cross-cutting* 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 cross-cutting requirement has behaviour that will affect multiple classes in different design models in a uniform way. For these kinds of requirements, this thesis defines and discusses a mechanism whereby this common way of composing the cross-cutting design elements may be defined as a composition pattern.

### *Solving the Problems*

In [Engels & Groenewegen 2000], two of the issues with object-oriented modelling that are discussed relate to modularisation (or decomposition) of models, and composition of models. This thesis illustrates that limitations with current modularisation possibilities are the cause of difficulties with comprehensibility, extensibility and reuse of object-oriented designs. The limitations identified and illustrated in “Chapter 2: Motivation” on page 11 are directly associated with the structural mismatch between the modularisation of requirements specifications and the modularisation of object-oriented designs.

The subject-oriented design model described in this thesis removes this limitation by adding the capability of decomposing design models in a manner that supports the direct structuring of design models with requirements specifications. The approach is simple, as it means that standard object-oriented design techniques may be used for the resulting individual design models.

The primary extension to the standard is a new *composition relationship* that supports the composition of those models that contain the design of different requirements. So, how does this model solve the problems that current modularisation limitations cause?

Comprehensibility

As previously discussed, comprehensibility difficulties relating to the structural mismatch between modularisation in requirements specifications and modularisation in object-oriented models are two-fold. First, in order to understand how a particular requirement is designed, multiple design elements must be examined and understood in full. Second, in order to understand a particular object-oriented design element (for example, a class), multiple requirements must be examined and understood in full. This is illustrated in “Chapter 2: Motivation” on page 11. The subject-oriented design model proposed in this thesis eases this comprehensibility problem by having separate design models for each requirement. Understanding the design of one requirement in full requires an understanding of only those design elements that directly support that requirement. An examination of a single design element requires a detailed knowledge of only one requirement. This approach, as illustrated throughout this thesis, has a positive impact on the comprehensibility of design models.

Extensibility

As for extending and changing a system’s design, this thesis also illustrates how this can be achieved in a manner that does not require direct manipulation of existing designs, and therefore is simpler as a result. Each extension (for example, as a result of a new requirement) or change (for example, as a result of a change to business processes) may be designed in a separate design model, with its composition with existing designs specified with a composition relationship. In “Chapter 2: Motivation” on page 11, there is a discussion of the negative impact of having to change designs directly when new requirements are received. In “Chapter 9: Applying the Subject-Oriented Design Model” on page 213, there is an illustration of the improvements to extensibility with the new approach described in this thesis.

Reuse

As previously discussed, an important impediment to reusing design models is the tangling of the design for multiple requirements within design elements. This results from the structural mismatch of the modularisation approaches in requirements specifications and object-oriented models. If a need is identified for reusing the design of some particular requirement, unwanted design elements are part of the deal, impacting development and testing. With the approach described in this thesis, however, each requirement is supported by a single design model, and therefore the reuse potential of that design model is considerably enhanced.

## 1.1. Thesis Contributions

The previous section discusses the problems with current object-oriented design techniques addressed in this thesis, introduces the approach to solving these problems that is the basis of thesis, and summarises how this new approach to object-oriented design solves these problems. In this section, a succinct summary of the contributions of the research described in this thesis is provided. They are:

- **Extensions to Object-Oriented *Modularisation* Capabilities**

  The units of abstraction and decomposition in current object-oriented designs tend to focus on interfaces, classes and methods. This thesis describes an additional unit of decomposition designed to align object-oriented designs with requirements specifications. This approach to decomposition has been previously documented and implemented at the code level in the work on subject-oriented programming [Harrison & Ossher 1993], [Ossher et al. 1996]. This thesis applies the *subject* approach to the Unified Modeling Language [UML 1999], which has not previously been researched.

  Important implications of modularisation in this manner are that:

  - Overlapping specifications are supported
  - Cross-cutting specifications are supported

- **Extensions to Object-Oriented Model *Composition* Capabilities**

  The subject-oriented design model introduces *composition relationships* to UML which specify how designs should be composed. A composition relationship between design subjects (and component design elements) indicates correspondences between elements in subjects that describe overlapping concepts, specifies how mismatches between corresponding elements are to be resolved with *reconciliation specifications*, and how design subjects are to be understood as a whole with *integration specifications.* The full semantics of the subject-oriented design model are described in this thesis.

- **Extensions to the UML Metamodel to Support Design Model Composition**

  Composition specification requires key extensions to the UML that are described in this thesis. The semantics of the UML itself have been specified at the meta-level in [UML 1999], with the description of a *metamodel*. A metamodel “defines a language for specifying a model” [UML 1999] - that is, it defines all the design language constructs (for example, Class, Operation, Attribute etc.) that are available for specifying a design model. Since a *composition relationship* is an additional kind of design language construct required to support subject-oriented design, its semantics are defined within the UML metamodel as defined in [UML 1999] (see "Chapter 5: Composition Relationship: An extension to the UML Metamodel" on page 109). This is achieved with:

- meta-class models illustrating the details of composition relationships
- well-formedness rules specifying constraints for composition relationships
- detailed descriptions of the semantics of composition

- **Composition Patterns for Composing Collaborative Behaviour Supported**

Sophisticated specification of the behaviour of operations that are merged from different design models is possible. This is supported with the ability to attach collaborations to composition relationships with merge integration. In particular, patterns of collaborative behaviour may be identified and reused. A requirement that may have a behavioural impact across the full design may be encapsulated, with this impact specified as a pattern. Pattern composition relationships may be specified when the behaviour needs to be reused (see "Chapter 8: Composition Patterns" on page 198).

## 1.2. Thesis Structure

Chapter 2

Chapter 2 motivates the need for an approach such as subject-oriented design by describing problems associated with current approaches to object-oriented design. The focus is on problems with the use of UML, and UML with design patterns [Gamma et al. 1994].

Chapter 3

Chapter 3 discusses the current state of software engineering from the point of view of providing a context for subject-oriented design. Different approaches to requirements specifications, object-oriented design, object-oriented programming and database management systems are discussed.

Chapter 4

Chapter 4 defines the foundation for the subject-oriented design model. There is a discussion of the approach to decomposing design models and the approach to specifying how design models may be composed using composition relationships, with an introduction to the rules associated with their usage. There is also an analysis of the output of a composition process - the

| | |
|---|---|
| | composed design models, and a discussion on the usage of the subject-oriented design model. |
| Chapter 5 | Chapter 5 defines the syntax and semantics of the subject-oriented design model against the UML metamodel. This includes meta-class diagrams of the constructs for composition relationships, well-formedness rules covering constraints on composition relationships, and descriptions of the semantics of composition. This chapter includes an abstract specification of how integration may be specified with composition relationships, but excludes details of any specific integration strategies. |
| Chapter 6 | Chapter 6 defines the syntax and semantics of override integration. This includes a meta-class diagram illustrating the constructs of override integration in the context of the composition relationship constructs in Chapter 5, additional well-formedness rules for composition relationships with override integration specified, and a detailed description of the impact of override composition on each of the design constructs supported in this thesis. |
| Chapter 7 | Chapter 7 defines the syntax and semantics of merge integration. This includes meta-class diagrams illustrating the constructs of merge integration in the context of the composition relationship constructs in Chapter 5, additional well-formedness rules for composition relationships with merge integration specified, and a detailed description of the impact of merge composition on each of the design constructs supported in this thesis. |
| Chapter 8 | Chapter 8 discusses how patterns of composition may occur, and presents a solution for specifying patterns of cross-cutting behaviour based on a combination of the subject-oriented design merge integration model, and UML templates. These patterns are called *composition patterns*. |
| Chapter 9 | Chapter 9 describes the application of subject-oriented design to the examples in Chapter 2, showing how those problems are ameliorated with subject-oriented design. |
| Chapter 10 | Chapter 10 demonstrates the use of the subject-oriented design model using a Library Management System case study. |
| Chapter 11 | Chapter 11 concludes and suggests possibilities for future work. |

# Chapter 2: Motivation

This chapter motivates the need for a new approach to object-oriented design. With current software engineering techniques, a structural mismatch exists between the specification paradigms across the software development lifecycle. This structural mismatch is the root of the problems described in this chapter. The problems exist because of a *scattering* and *tangling* effect that is the mismatch's natural outcome. That is, support for a single requirement touches multiple classes in the object-oriented design and code (scattering), and a single class in the object-oriented design and code may support multiple different requirements (tangling). The new approach to object-oriented design proposed by this thesis adds decomposition capabilities to the object-oriented design model that support structural matching to requirements, thereby reducing scattering and tangling.

First, this chapter examines the specification paradigms of the requirements, analysis/design and implementation phases of the development lifecycle. The different paradigms are compared and a structural mismatch is found.

It is then illustrated how the structural mismatch causes scattering and tangling properties. It is shown that these properties result in a negative impact on the initial development and evolution phases of software development. The illustration is based on working with a small example and uses the current OMG standard language for object-oriented design (UML), together with design patterns (design improvement techniques, [Gamma et al. 1994]). The impact of the structural mismatch is assessed based on criteria used by Parnas in [Parnas 1974].[1] These criteria are:

*Evaluation Criteria*

- *Product flexibility:* The possibility of making drastic changes to one part of the system, without a need to change others.

1. Parnas considered that these criteria were the benefits to be "expected of modular programming". These benefits remain good goals for high-quality software engineering.

- *Comprehensibility:* The possibility of studying the system one part at a time. The whole system can therefore be better designed because it is better understood.
- *Managerial:* The length of development time, based on whether different groups can work on different parts of the system with reduced need for communication.

The expected benefits to software design discussed in “Chapter 1: Introduction” on page 1 (comprehensibility, extensibility and reuse) are subsumed and extended by Parnas’ criteria. Extensibility is discussed within “product flexibility” and reuse is discussed within “comprehensibility”.

The problems found motivate the need for a different design approach. A new design approach is proposed that diminishes the difficulties described and is the central tenet of this thesis.

## 2.1. Specification Paradigms Across Lifecycle

This section compares the specifications of requirements, object-oriented analysis/designs, and object-oriented implementations for software systems. The comparison is made based on one central theme - *how the problem is divided into smaller parts*.

As with any large, complex problem, breaking the problem into smaller parts makes it easier to understand [Pólya 1957]. Software engineering is no different in this respect, and so the specifications from each phase divide the whole problem into smaller parts. This section examines the selection of the parts for division in each phase, and the motivations for those selections. It is illustrated that since the motivations for selection are different, the resulting divisions are different, causing a structural mismatch in the specifications.

The process of developing software, and of changing software over its lifetime, has a number of different basic phases. These are:

*Software Phases*

- *Requirements Specification:* The output of this phase is a documentation of what the software system is expected to do [Jacobson et al. 1999]. The needs and requirements of the potential end-users of the software system are elicited and documented. The business processes the software system must support are examined, and the requirements to support those business processes are documented. The technical environment and technical constraints within which the software system must run are assessed and documented. All existing software systems with which the new software system must interact are identified, and the requirements for their interaction documented. Requirements specifications tend to be documented in a language which can be understood by the eventual users of the system. This is generally a natural language.

- *Analysis and Design:* The *analysis phase* refines and structures the requirements, providing a better understanding of those requirements [Jacobson et al. 1999]. By refining the requirements into more detail, the analysis process attempts to tease out any ambiguities and inconsistencies associated with the requirements specifications, and attempts to ensure that the complete set of requirements for the computer system has been defined[2]. This process is performed with the involvement of the business domain experts, and the people who define the technical requirements, in cooperation with the software analysts. The requirements are structured and documented in the language of the developer. The *design phase* shapes the system, providing sound and stable architectures and creates a blueprint for the implementation [Jacobson et al. 1999]. Detailed design decisions are made and documented (for example, class structure/behaviour; how the system should handle performance, distribution, concurrency - indeed, all technical concerns; subsystem separation for implementation; etc.).
- *Implementation:* Starting from the design specifications, the system is implemented in terms of source code, scripts, binaries and executables [Jacobson et al. 1999].
- *Test*: The result from the implementation is verified against the requirements. A test team develops a set of test cases that are based on the requirements specifications. The test cases are run against the software to verify that *all* the requirements are met by the software.

**Requirements**

The usage of software systems in society is ever increasing. Individuals, and groups of individuals (for example, clubs or businesses), have different needs for software systems from both a business and personal perspective. The vocabularies and processes used to describe these needs are wide and varied. This section examines:

2. Without the use of a formal description technique, it is difficult to test or measure the completeness and lack of ambiguity/inconsistency of analysis specifications. Without the ability to test and measure these properties, informal analysis techniques are therefore assumed to be, to some extent, ambiguous, inconsistent and incomplete.

1. the differences in the vocabularies used by various approaches to label individual requirement "units" (e.g. feature, functionality, service), and also,
2. the different approaches to dividing up a requirements specification into smaller parts. There are many terms associated with multiple users of a software system using it in different ways (e.g. role, view, perspective, responsibility). Each of these have an influence on the decision-making process associated with dividing the requirements specification into smaller parts, and so these factors are considered.

*"Units" in Requirements Specification*

First, a look at how individual units in a requirements specification are labelled. There are many words used to describe what a computer system is supposed to do: "requirement", "feature", "functionality", "facility" and "service". In order to give a context for the vocabulary, the dictionary [OED 1989] definitions for each of these terms are as follows:

| | |
|---|---|
| *Requirement:* | "need; depend for success, fulfilment, etc. on; wish to have" |
| *Feature:* | "distinctive or characteristic part of a thing; part that arrests attention; important participant in" |
| *Function:* | "mode of action or activity by which a thing fulfils its purpose" |
| *Service:* | "provision of what is necessary for due maintenance of a thing or property" |
| *Facility:* | "equipment or physical means for doing something" |

Different requirements engineering processes use different vocabularies to describe units of interest to the requirements gatherer. For example, the Unified Software Development Process, described in [Jacobson et al. 1999], refers to requirements, features and functionality, but in essence, describes the process of capturing *requirements* as "*Use Cases*". A use case delimits the system from its environment; outlines who and what will interact with the system, and what functionality is expected from the system; and captures and defines in a glossary common terms that are essential for creating detailed descriptions of the system's functionality.

Modelling domains in a *feature*-oriented way, integrated with a use case approach is described in [Griss et al. 1998]. The purpose of feature-oriented domain analysis (FODA) is "... to capture in a model the end-user's (and customer's) understanding of the general capabilities of applications in a domain", which, the point is made, "sounds like use-case modelling". However, the integration of the two approaches is motivated by the difference of use-case modelling and feature modelling serving different purposes. The use case model is *user-oriented*, providing the “what” of a domain: a complete description of what systems in the domain do. The feature model is *reuser* oriented, providing the “which” of the domain: which functionality can be selected when engineering new systems in the domain.

Features, described as “an optional unit or increment of functionality” in [Zave 1999], are also at the core of the Distributed Feature Composition (DFC) architecture described in [Jackson & Zave 1998]. The fundamental idea of the DFC architecture for the telecommunications domain is to treat features as independent components through which calls are routed from caller to callee. Examples of features in the telecommunications environment are “call-waiting”, or “3rd-party conference”.

*Services* and *facilities* are part of the specification of the OMG work on CORBA [Mowbray & Zahavi 1995], [Siegel 1996]. Examples of services a system supporting distributed objects, and conforming to the CORBA standard, should provide are an object naming service and an object event service. Examples of common facilities provided for by CORBA are user interface facilities, and data interchange facilities.

*Motivation for Choosing Units*

From these definitions, and the approach of different requirements specification techniques, requirements for computer systems can be seen to be *statements of what the computer system should do*. The opinions of what computer systems should do, even opinions of the same computer system, are dependent on the people who will use the system, and what they will use the system for. Different kinds of people have different needs - and again many different terms are used to describe the different motivations, for example: view; perspective; role. As before, in order to give a context for the vocabulary, the dictionary [OED 1989] definitions for each of these terms are as follows:

| | |
|---|---|
| *View:* | “manner of considering a subject, opinion, mental attitude; intention, design” |
| *Perspective:* | “aspect of a subject and its parts as viewed by the mind; view” |
| *Role:* | “one’s function, what person or thing is appointed or expected to do” |

Processes for requirements gathering take different approaches that are based on the motivations of the end-users of the computer system. Those motivations depend on the views, the perspectives, the roles or the responsibilities of the end-users. Views in requirements engineering are the focus in [Nuseibeh et al. 1994], where views are described as allowing "development participants to address only those concerns or criteria that are of interest, ignoring others that are unrelated". A framework for requirements elicitation based on the capture of multiple perspectives is described in [Easterbrook 1991], while the roles end-users play under different domain-dependent circumstances are the motivation behind role-modelling from [Reenskaug et al. 1995].

*Output of Requirements Phase*

A requirements specification, therefore, contains descriptions of required features, services, functions and facilities. Potentially, each individual unit may be described from different views and perspectives, and to support multiple roles.

## Object-Oriented Analysis and Design

In this section, the units of the object-oriented analysis and design paradigm are examined, together with the typical motivations for their specification.

From the early to the mid 1990's, there was a so-called "methods war" [Jacobson 1994], which resulted in "26 different object-oriented methods described by OMG's special interest group on analysis and design (SIGAD)". The proliferation of multiple methods prompted numerous studies into the differences between them, for example [deChampeaux & Faure 1992], [Carmichael 1994], [Graham 1993], [Hutt 1994]. These studies illustrate differences between methods, but for the purposes of comparison of the basic units of decomposition common to the object-oriented paradigm, it is sufficient to consider them collectively, as the methods generally agree in this regard. The most basic units of decomposition in object-oriented analysis and design methods in general are classes and objects [Wirfs-Brock et al. 1990]. Classes and objects encapsulate further units describing structural and behavioural elements of the system, namely attributes, operations, interfaces and methods. Many different methods have slightly different definitions of these terms, but essentially, the notions are the same.

*"Units" in Object-Oriented Specification*

Some examples of how each of the units are described in some of the different methods are:

**Structural Units**

| | |
|---|---|
| *Class:* | A description of a set of objects that share the same attributes, operations, relationships and semantics [Booch et al. 1998] |
| | Objects which share the same behaviour are said to belong to the same class. A class is a generic specification for an arbitrary number of similar objects [Wirfs-Brock et al. 1990]. |
| | A description of a group of objects with similar properties, common behaviour, common relationships and common semantics [Coleman et al. 1994] |
| *Object:* | A concrete manifestation of an abstraction; an entity with a well-defined boundary and identity that encapsulates state and behaviour; an instance of a class [Booch et al. 1998]. |
| | A concept, abstraction or thing with crisp boundaries and meaning for the problem at hand [Coleman et al. 1994]. |
| | The "is a" abstraction, representing a part of a system. An object has identity and attributes and is encapsulated so that the messages it sends and receives constitute all its externally observable properties [Reenskaug et al. 1995] |
| *Attribute:* | A named property of a class that describes a range of values that instances of the property may hold [Booch et al. 1998]. |
| | A data value held by the objects in a class [Coleman et al. 1994]. |
| | The information an object may store [Reenskaug et al. 1995]. |

**Behavioural Units**

| | |
|---|---|
| *Operation:* | The implementation of a service that can be requested from any object of the class in order to affect behaviour [Booch et al. 1998]. |
| | A function or transformation that may be applied to or by objects in a class [Coleman et al. 1994]. |
| | A piece of code triggered by a message [Cook & Daniels 1994]. |
| *Interface:* | A collection of operations that are used to specify a service of a class or a component [Booch et al. 1998]. |
| *Method:* | The implementation of an operation [Booch et al. 1998], [Coleman et al. 1994]. |

*Motivation for Choosing Units*

The motivations associated with the choice of "object" as the basic decomposition unit in the object-oriented software paradigm was to model "real world" objects, thereby making software systems easier to develop and understand. Since everyday living involves dealing with all kinds of objects, the concept of working with objects at the software specification level is therefore familiar and intuitive.

**note:** Following the methods war of the early 1990's, a collaborative effort started which resulted in a consortium of companies agreeing on a single submission to the OMG for an object-oriented analysis and design modelling language - the Unified Modelling Language (UML) [UML 1999]. Given the general consensus associated with the usage of UML as the standard object-oriented modelling language (and endorsed as a standard by the OMG), this thesis will hereafter refer to the semantics definition of the UML only. Analysis and design are considered throughout the thesis as object-oriented *modelling*. Though often referred to within the thesis as *designs*, the models considered are any that are written using the UML.

## Object-Oriented Implementation

The units of decomposition in object-oriented programming technologies such as C++ [Stroustrup 1991] and Java™ [Gosling et al. 1996], directly and deliberately match the the units at the design level described in the previous section. The direct matching is clear from each of the object-oriented programming languages' construct support for the notions of: class; the encapsulation of attributes and methods with class; interface; and the instantiation of classes to produce runtime objects. The deliberate matching is natural for the purposes of structuring object-oriented code with the same decomposition units as object-oriented designs, thereby providing direct traceability between the two phases.

For the purposes of this examination of specification paradigms across the software development lifecycle, the specification paradigms of the object-oriented design and object-oriented implementation phases are therefore considered as the same.

## Comparison

The requirements specification paradigm contains the notions of features, capabilities, services, functions etc. - with generally no mention of objects and interfaces or any of the units of interest in the object-oriented design domain. The object-oriented paradigm contains the notions of objects and interfaces etc. - with no mention of features, or requirements, or any of the units of interest in the requirements domain. That is the mismatch.

The units of interest in the requirements domain are structurally fundamentally different to the units of interest in object-oriented designs. Thus, requirements units of interest generally are not, and cannot readily be, encapsulated in the design. This is illustrated in "2.3. SEE System Design, Version 1.0" on page 22.

In the previous section, there is a discussion about how object-oriented designs structurally match object-oriented code, providing a measure of traceability between the two phases. This necessitates a transition from “feature” (or function or....) concerns in the requirements phase to the object/class concerns of the object-oriented paradigm at the design phase. In achieving a close tie to code, object-oriented design loses potential for a close tie with requirements.

This point is particularly important. *In general, most design paradigms are not sufficiently powerful to permit designs to match both requirements and code* - they allow designs to align with either the requirements or the code, but not both.

The evidence of the negative impact of the structural mismatch between requirements specifications and object-oriented designs can now be presented. The next section introduces the example to be used that will show this evidence. The following section illustrates how the mismatch affects the initial development of the system (“2.3. SEE System Design, Version 1.0” on page 22). The negative impact on the evolution of that system is described in “2.4. Evolving the SEE System Design” on page 29.

## 2.2. Example: Software Engineering Environment

This section presents a running example that is used to illustrate the problems that motivates this research. The example involves the construction and evolution of a simple software engineering environment (SEE) for programs consisting of expressions. A simplified software development process is assumed, consisting of informal requirements specification in natural language, design in UML, and implementation in Java.

**Requirements Specification**

The required SEE supports the specification of simple expression programs. The following initial set of tools are needed to work with expressions:

- an *evaluation* capability, which determines the result of evaluating expressions;
- a *display* capability, which depicts expressions textually; and
- a *check* capability, which optionally determines whether expressions are syntactically and semantically correct.

The SEE should also permit optional logging of operations.

**Supported Grammar for Expressions**

The initial software system supports a small grammar for expressions as follows:

```
Expression := VariableExpression | NumberExpression | Plus-
Operator | MinusOperator | UnaryPlusOp | UnaryMinusOp
PlusOperator := Expression '+' Expression
MinusOperator := Expression '-' Expression
UnaryPlusOp := '+' Expression
UnaryMinusOp := '-' Expression
VariableExpression := ('A'| 'B' | 'C' | ... | 'Z') +
NumberExpression := ('0' | '1' | '2' | ... | '9') +
```

This grammar is very simple and small to effectively illustrate two problems: first, even with a small grammar, the design of a supporting SEE gets unwieldy and second, adding new constructs to the grammar, for example a product or assignment operator, requires invasive changes to the design.

**Expressions as Abstract Syntax Trees**

In this thesis, the SEE design in all examples represents expressions as abstract syntax trees (AST). Each type of AST node is represented as a class as shown in Figure 1.

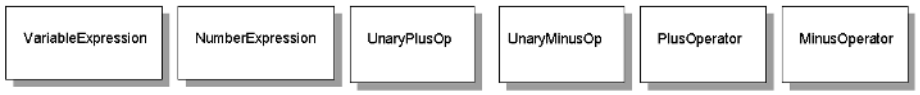


**Figure 1: AST Nodes as Classes**

Further examination of the nodes of the tree for this grammar show that there may be common properties between different nodes which could be abstracted to superclasses. For this example, the `PlusOperator` and the `MinusOperator` have similar properties in that they both have left and right operands, which could be abstracted to a class called `BinaryOperator`. Also, the `UnaryPlusOp` and the `UnaryMinusOp` are similar in that they both only have one operand, which could be abstracted to a class called `UnaryOperator`. Finally, `NumberExpression` and `VariableExpression` are literals, and so could be abstracted to a class called `Literal`. These classes are illustrated in Figure 2.

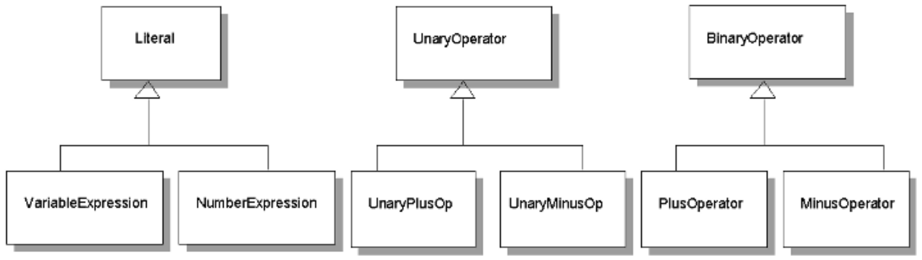


**Figure 2: AST Classes with Superclasses**

The tree structure nature of the AST is supported using the Composite pattern from [Gamma et al. 1994]. The intent of the Composite pattern is to "compose objects into tree structures to represent part-whole interactions". The idea is to provide a uniform interface to the objects within such a tree structure, be it a leaf or a composite object. Composite is centred around an abstract class that represents both primitives (in the SEE case, literals) and their containers (in the SEE case, operators, which "contain" one or two expressions). From the pattern, a container object maintains an aggregation relationship [Booch et al. 1998] with its parts. As shown in Figure 3, the abstract class that is used to represent literals and operators is called `Expression`. Since both `UnaryOperator` and `BinaryOperator` are containers of expressions, they maintain aggregation relationships with `Expression`.

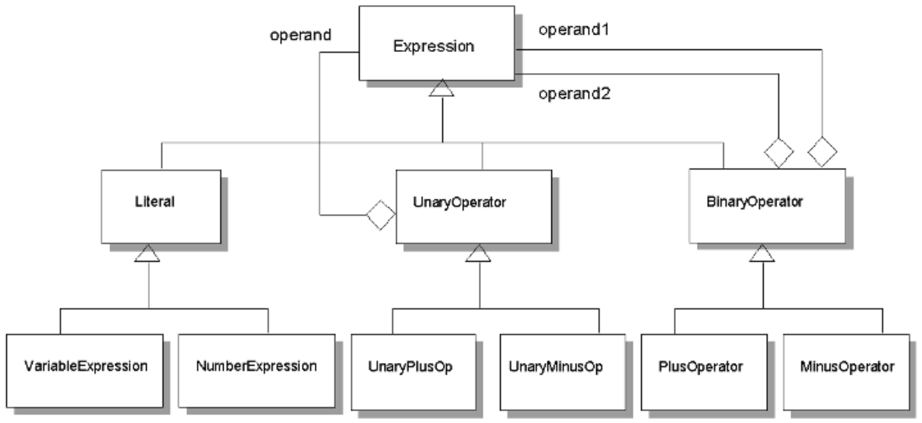


**Figure 3: Composite Pattern for AST**

The basic structure of this design recurs in all examples of designs for a software engineering environment supporting expressions.

The next two sections show evidence of the negative impact of that structural difference on a small example object-oriented system design, affecting first the initial development and then the evolution of that system.

## 2.3. SEE System Design, Version 1.0

In this section, the design is considered as “Version 1.0” (the “first release”) of the SEE system. In later sections, the impact of evolving the system as a result of adding new requirements is assessed.

The requirements specification in “Requirements Specification” on page 19 identifies several requirements that must be realised in the design: expression support, the evaluation tool, display tool, check tool, and a logging utility that can be included or excluded from the environment.

There may, of course, be many approaches to the design and implementation of such a system, from both a management and technical point of view. Technically, a simple design is illustrated here. In “Evolving the SEE System Design” on page 29, some general kinds of problems that other approaches produce (notably, those that use design patterns) are discussed. From a management perspective, let us assume that the project manager recognises that a team member is knowledgeable in the area of expressions, and design patterns, and gives him the task of designing the core expression environment. This designer designs an expression as an abstract syntax tree, as described in “Expressions as Abstract Syntax Trees” on page 20, which, with its structural and accessor properties, is illustrated in Figure 4.

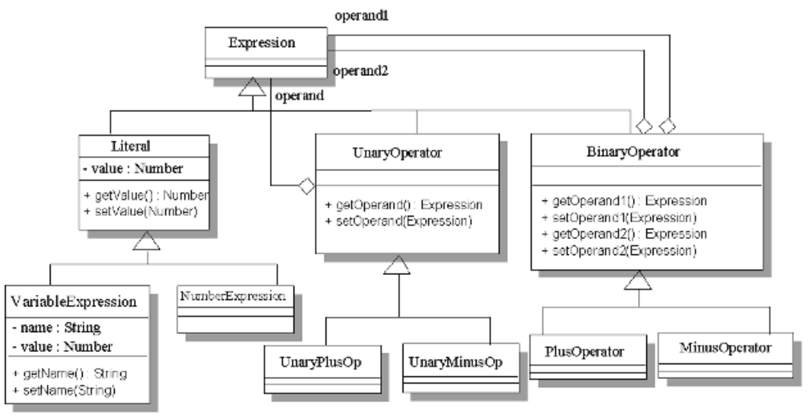


**Figure 4: Core Expression Design in UML**

The project manager also has an expert in the syntax checking of expressions, who is given the task of designing the check requirement.

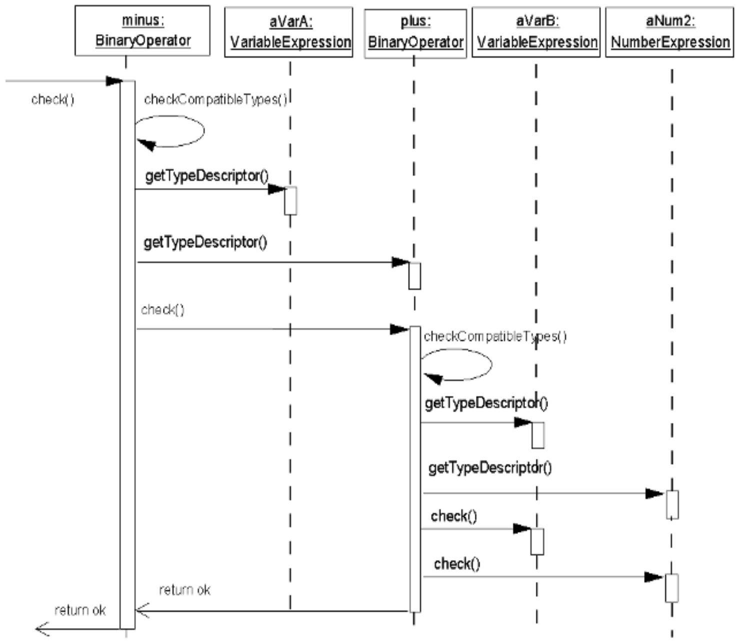


**Figure 5: Sequence Diagram for Checking the Syntax of an Expression: `A−B+2`**

This designer, however, must wait until the core structure of the expression classes is decided, before working on a design for the checking behaviour. He works with a number of scenarios for sequence diagrams to determine the required operations, determining that recursive operations are appropriate for the tree nature of expressions. One example of a scenario is one to support the checking of the expression `A−B+2` as illustrated in Figure 5.

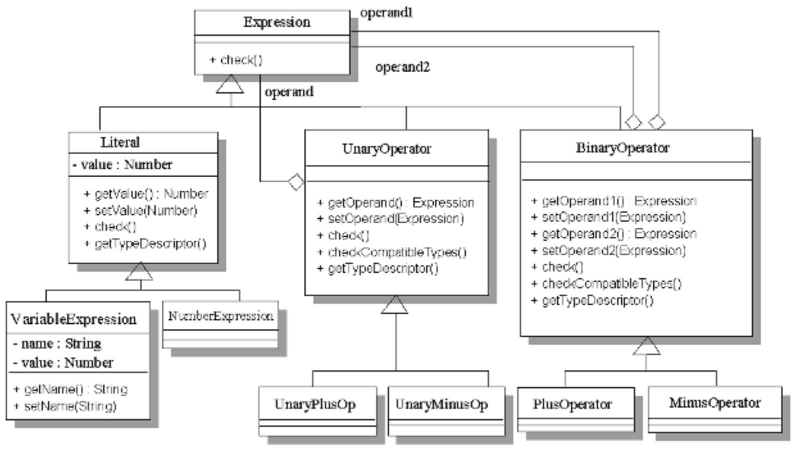


**Figure 6: Support for Check added to Class Diagram**

Once the check designer is comfortable with the design elements (attributes and operations) that are to be added to the class, and therefore appear on class diagram, he must ensure exclusive access to the class diagram in order to update it with the additional properties to support checking expressions. Of course, sophisticated CASE tool support may reduce the “wait-time” for the exclusive access to the class diagram. The impact of adding the checking design properties to the class diagram is illustrated in Figure 6.

The experts on evaluating expressions and those on displaying expressions, design these tools as recursive functions over the abstract tree representation of expressions, in a standard object-oriented manner, using the UML [Booch et al. 1998], and in a manner similar to the behavioural design of the check tool illustrated in Figure 5. The behavioural diagrams may be worked on separately, but the additional structural and behavioural properties may only be added to the class diagram when it is available, after which the class diagram is as illustrated in Figure 7.

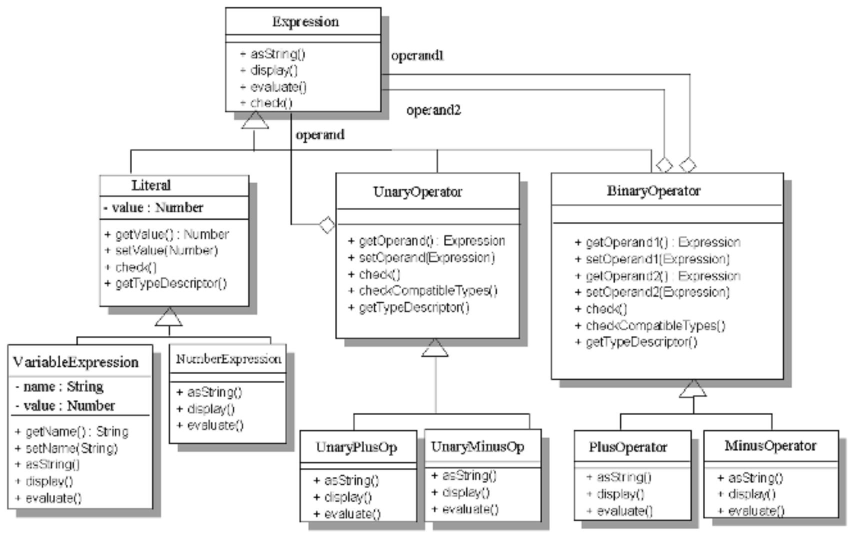


**Figure 7: Class Diagram with Expression AST and Check, Evaluate, and Display Tools**

The remaining requirement to be designed is the optional logging of operation execution. Figure 8 shows an example collaboration diagram for logging a `check()` operation. If the logging utility is turned on (modelled as a Boolean attribute `loggingOn`) each operation invokes `Logger.beforeInvoke()` prior to performing its action, then invokes `Logger.afterInvoke()` just before it terminates. The Logger permits applications to turn logging on and off with its `turnLoggingOn()` and `turnLoggingOff()` methods. This permits logging to be optional, as required.

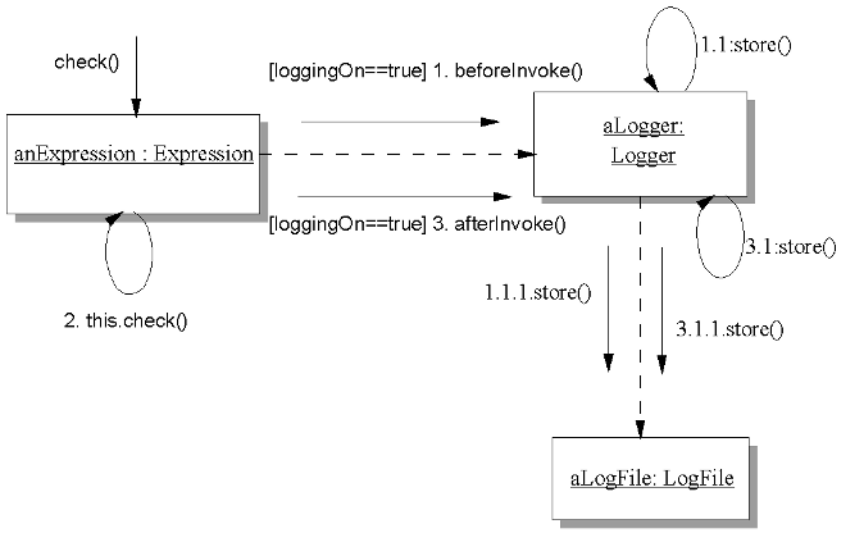


**Figure 8: Collaboration Diagram for Logging Utility: Example - `Check()`**

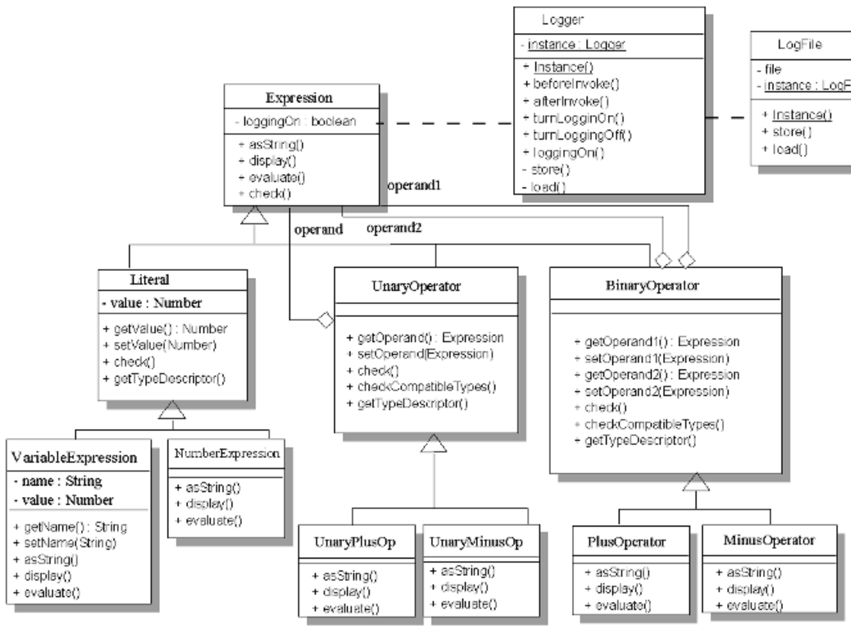


**Figure 9: UML Class Diagram for SEE, Version 1.0**

The impact of the logging requirement on the structure diagram of the SEE is illustrated on Figure 9. Logging is modelled as a separate, singleton class, `Logger`. "Singleton" is a design pattern from [Gamma et al. 1994] that ensures a class only has one instance, as is appropriate for a class performing a logging function that will always behave the same way regardless of what operation is being logged.

The design demonstrates some important features. The mapping from design to code is straightforward and quite direct - every unit of interest (i.e. class) in the UML class diagram will have a direct correspondent in the code. This is not unexpected, since both are object-oriented, and much of the reason for the trend toward object-oriented design is that it permits a direct mapping between design and object-oriented code.

The mapping between the SEE requirements specification and this design, on the other hand, is more complex. Even with a requirements specification for a small system, there is evidence of problems against each of our evaluation criteria from Parnas:

- product flexibility
- comprehensibility
- managerial.

## Product Flexibility

As described in [Parnas 1974], product flexibility is the "*possibility of making drastic changes to one part of the system, without the need to change others*". The structural differences in the specification paradigms between the requirements specification, and the object-oriented design (discussed in general in "2.1. Specification Paradigms Across Lifecycle" on page 12) for the SEE are central to the difficulties associated with changing the system.

The natural outcome of the structural differences is a *scattering* and *tangling* effect across the object-oriented design.

| | |
|---|---|
| *Scattering:* | The structural difference results in the design of a single requirement being *scattered* across multiple classes and operations in the object-oriented design. |
| *Tangling:* | The structural difference also means that a single class or operation in the object-oriented design will contain design details of multiple requirements. |

*Scattering* and *tangling* are apparent in the design for the SEE.

<u>*Scattering:*</u> The SEE requirements of expression evaluation, checking, and display, which are described as encapsulated concerns in the requirements specification, are not encapsulated in the design. In fact, these requirements are *scattered* across the AST classes - each class contains a method that implements these capabilities for its own instances. Scattering is negative from an evolutionary perspective: the impact of change to a single require-

ment, well localised at the requirements level, can nonetheless be extremely high, because that change necessitates multiple changes across a class hierarchy.

*Tangling:* The logging capability is realised as a first-class unit of interest in both the requirements and the design. Nonetheless, the protocol for logging requires co-operation from each method in each AST class, to appropriately invoke `Logger.beforeInvoke()` and `Logger.afterInvoke()`. This is *tangling* - satisfying a given requirement necessitates interleaving design details that address the requirement with details that address other requirements. Tangling is a serious impediment to software comprehension, reuse and evolution because it is impossible to deal with the design details relating to one requirement without constantly encountering and having to worry about intertwined details relating to other requirements.

*Traceability:* Scattering and tangling are also devastating from the point of view of *traceability*: the ability to determine readily how a piece of one software artefact (e.g. requirement, design, code) affects others. Traceability makes it possible to look at a change to a requirement, and to find those parts of the design and code details that are affected by the change. Traceability is essential to keeping requirement and design documents up-to-date with respect to evolving code. Without it, these documents are likely to become obsolete and useless, since, when it is difficult to determine how a proposed change to one will impact the other, changes may not be propagated across them consistently, or at all.

**Comprehensibility**

As described in [Parnas 1974], 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*". The descriptions of the scattering and tangling problems as manifested in the SEE, 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.

Comprehensibility is also an essential property to the successful reuse of any unit from a system design, as any unit to be reused must be understandable or it will not be reused correctly. "Reuse" is a much heralded benefit of the object-oriented approach to software engineering, but the properties of scattering, tangling and poor traceability also contribute to a design that is difficult to reuse. Poor traceability (resulting from scattering and tangling) makes it difficult to follow exactly what parts of the design relate to a particular feature of the system, and therefore what parts must be included in a reuse of that feature. Another important ingredient for successful reuse is clean boundaries - i.e. a design unit that does not have interdependencies with other design units, and which may therefore be easily incorporated into a different system, with limited impact on the system. Again, scattering and tangling properties in a design are the antithesis of such a clean incorporation into another system. Further, effective reuse requires powerful mechanisms for customisation and adaptation. With this design, designers are forced to make invasive, rather than additive changes to adapt design units. For example, adding a new feature to the SEE, like additional forms of checking of expressions, requires each of the AST nodes to be changed.

**Managerial**

As described in [Parnas 1974], 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"*. 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. As described, even in this small system, comprehension, maintainability and reusability are reduced as a result of the monolithic nature of the classes. This monolithic property also 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 compiler expert work on the AST representation itself, a user interface expert work on the design of the display feature, etc. The scattering and tangling of these features results, however, in interdependencies across these features and across the classes, that hampers concurrent design and implementation. Since classes encapsulate and own their own structural and behavioural properties, they are inherently centralised notions, so it is also often fairly difficult to permit concurrent development of the same classes. Further, while the logging capability can be designed independently of the AST classes, all the developers must be aware of its presence and must design with it in mind. For the same reasons, all of the SEE tool designers must wait for the "core" AST to be defined before they can work effectively even if designers could work in parallel on features. This opens the door to a variety of errors, and it can result in delays while designers wait for one another.

**Conclusion**

The core reason for these problems is because the concerns identified in the requirements, which are based on *requirements* of the SEE, are different from those used to modularise the design, which are the *objects* and *classes* that implement the SEE. Thus, the requirements units of interest generally are not, and cannot readily be, encapsulated in the design. This is different from the relationship between design and code, where the respective set of concerns are very similar. In the process of creating designs from requirements, UML and other object-oriented formalisms and languages necessitate a transition from feature concerns to object concerns. This transition essentially results in the discarding of the encapsulation of those units of interest identified during requirements specification in favour of units of interest mandated by the design and coding paradigms. In achieving a close tie to code, object-oriented design loses its close tie with requirements. Scattering and tangling are, in fact, symptomatic of this mismatch.

Thus, designs fail to achieve one of their primary purposes: to promote *traceability* by bridging the gap between requirements and code. Traceability is an important prerequisite to evolution, as is encapsulation, which aids in limiting the impact of any given change. For example, it is difficult both to determine how a change to the logging requirement will impact the design, and to affect such a change additively, rather than invasively. Limited traceability and encapsulation, as is present in the SEE design, result in reduced evolvability. Consequently, they also result in the eventual obsolescence of requirements, design or both, since changes may not be propagated consistently if it is difficult to determine how a proposed change to one will impact the other.

The next section looks at the process of evolving the SEE system as a result of new requirements. Different approaches to designing systems, based on Design Patterns [Gamma et al. 1994], are examined to assess whether they are sufficient to solve the problems illustrated in this section.

## 2.4. Evolving the SEE System Design

This section assesses the impact on the design of adding new requirements to the SEE requirements specification. The approaches to extensibility as recommended by design patterns [Gamma et al. 1994] are considered.

**New Requirements**

After using the SEE for some time, the clients request the inclusion of different forms of optional checking;

1. A check is required to ensure that all variables used are defined, and all variables defined are used (def/use)
2. A check is required to verify that expressions conform to local naming conventions.
3. The check feature is a “mix-and-match” capability - clients can choose a combination of syntax, def/use, and/or style checking to be run on their expression programs when they invoke the check tool.

**Extending Version 1.0 directly**

This change in requirements is additive - it need not affect any other requirement. At the design level, however, the change is not as straightforward, since the check feature is not encapsulated as a concern in the design. In fact, this change necessarily affects all AST classes in the design. One approach is to add new `defUseCheck()`, and `styleCheck()` operations to each of the AST classes, with conditional execution based on boolean attribute options. This approach requires each class in the design to be changed, with corresponding significant potential for error introduction even to Version 1.0 of the SEE system design. Another possible approach to designing the new forms of checking would be to create new subclasses of the AST classes, where a given subclass overrides the original (syntax) `check()` method with one intended to provide def/use or style checking for a particular kind of AST class. Clearly, while this approach is non-invasive, it is completely impractical, as it results in combinatorial explosion of classes with each new feature.

**Using Design Patterns**

A better approach is to use the Visitor design pattern [Gamma et al. 1994]. The Visitor pattern “represents an operation to be performed on elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates” [Gamma et al. 1994].This pattern definition with its corresponding description in [Gamma et al. 1994], makes it a good candidate for solving the problem of adding new check operations non-invasively. This is achieved by having a Visitor to represent checking, and to provide different visitors that correspond to the different kinds of checking. The Visitor approach, which is depicted in Figure 10, facilitates “mix-and-match” without combinatorial explosion of classes. It requires, however, an invasive change to all of the AST classes, to replace the `check()` methods with `accept(Visitor)` methods.

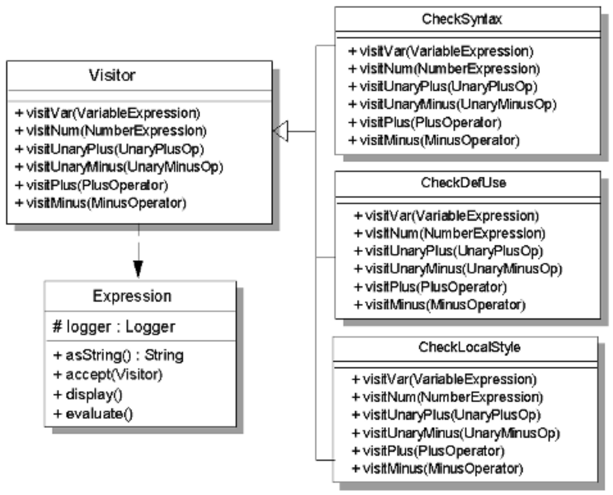

Visitor
+ visitVar(VariableExpression)
+ visitNum(NumberExpression)
+ visitUnaryPlus(UnaryPlusOp)
+ visitUnaryMinus(UnaryMinusOp)
+ visitPlus(PlusOperator)
+ visitMinus(MinusOperator)
Expression
# logger : Logger
+ asString() : String
+ accept(Visitor)
+ display()
+ evaluate()
CheckSyntax
CheckDefUse
CheckLocalStyle

**Figure 10: Using Visitor to Separate Check Functions**

The use of visitors also introduces a second complication. The logging feature requires the visitors to invoke `Logger.beforeInvoke()` and `Logger.afterInvoke()` appropriately, further increasing the scattering and tangling problems associated with this feature.

Another possibility for the use of design patterns is in the design of the logger functionality. For example, a mutation of the Observer pattern [Gamma et al. 1994] appears as if it might be useful in capturing operations for logging. The Observer pattern supports an object that has changed state notifying other objects that have expressed an interest in its state. In Figure 11, this approach is evolved to capture all operations on an object by the interested object which is the Logger.

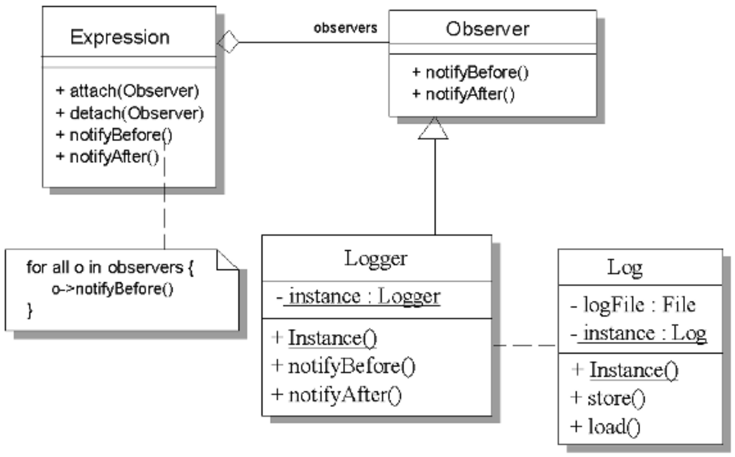


**Figure 11: Using Observer for Logging**

In this design, any operation call results in a call to `notifyBefore()` and `notifyAfter()`, before and after its execution. This approach has the advantage that any object other than an instance of a logger, may express an interest in operations within the expression, and attach itself as an observer to be notified before and after operation execution. For example, different kinds of audit trails may be attached with no change to the design of the expression AST.

Another approach to designing logging is to use the Decorator pattern [Gamma et al. 1994]. Decorator supports the attachment of additional responsibilities to an object dynamically. Decorators provide an alternative to subclassing for extending functionality, and reduces coupling by, for example in the logging case, separating the logging functionality into separate, decorator objects, as illustrated in Figure 12.

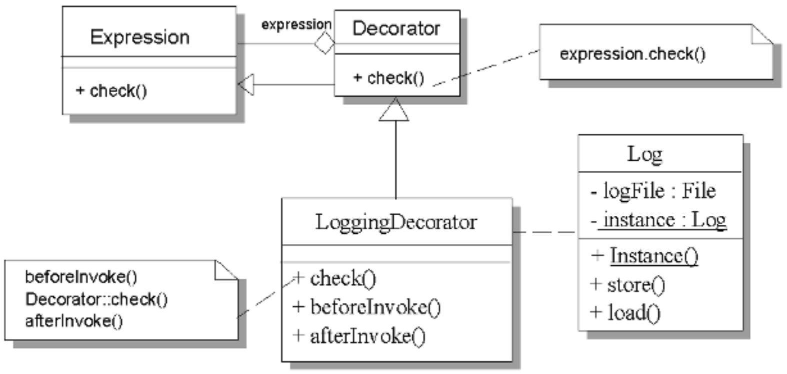


**Figure 12: Using Decorator for Logging**

**Assessing Design Patterns**

Many other design approaches are possible for the SEE, and some of them address some of the issues that have been raised. For example, the judicious application of design patterns might help solve some of these problems. While it is impossible to elaborate the possible design approaches (with or without design patterns) exhaustively, this section briefly explores some of the design pattern alternatives to illustrate why neither they, nor other approaches, address the whole problem.

*Visitor:* The initial use of the visitor pattern to model checking ("SEE System Design, Version 1.0" on page 22) would have facilitated greatly the addition of new checkers - this is the case precisely because visitors provide encapsulation of features, which results in better alignment of design with requirements. While visitors promote some forms of evolution, they hinder other forms. For example, adding a new type of expression, like assignment, is simple in the original design in Figure 9, but it would necessitate invasive changes to *all* visitors [Gamma et al. 1994].

_Observer:_ To reduce the coupling between the logger and the AST classes, logging could be performed by observers. This approach would achieve looser coupling. Observer is, however, an extremely heavyweight solution that incurs high overhead, in both complexity and performance. Further, it does not improve the scattering problem, as AST methods must notify any observers, thereby scattering the implementation of logging across all the AST classes. Used in conjunction with visitors for the AST tools (check, evaluate, display), the design for the SEE becomes significantly larger and more complex, with many more interrelationships among the classes to be represented and enforced.

_Decorator:_ As an alternative to observer, logging could be designed using the decorator pattern, where decorators optionally perform logging. Decorator, like observer, helps to reduce coupling, and unlike observer, it reduces tangling by segregating logger notification code into separate, decorator objects. Unfortunately, the decorator solution is significantly more problematic than the observer solution, because of the *object schizophrenia* problem. That is, to ensure that logging occurs consistently, it is necessary to ensure that *all* messages to all objects go through the decorator, *not* directly to the object itself. Once a method on an object is invoked, however, that method may invoke others, which, in turn, must go through the decorator. This means that the object must know about its decorator(s), which introduces a new form of coupling and tangling (i.e. each class must include code to implement interaction with the decorator).

This evolutionary change, which appeared to be straightforward and additive from the client's perspective and from its impact on the requirements, demonstrates, in a microcosm, the spectrum of problems resulting from the misalignment problem. Scattering and tangling lead to weak traceability and poor encapsulation of requirements-level concerns within the design, and subsequently, the code. They also make propagation of requirements changes to design and code very difficult and invasive. It is even difficult to determine which design elements are affected by a given requirements change. The level of effort needed to propagate changes from requirements to design is much greater than the effort to propagate the changes from design to code, precisely because of the misalignment.

**Summary**

Design patterns can help alleviate some, but not all, of the identified problems. Unfortunately, in diminishing some problems, they introduce other problems or restrictions [Gamma et al. 1994], [Vlissides 1998]. Designs and code must be pre-enabled with design patterns to avoid subsequent invasive changes to incorporate them. This need to pre-plan for change - which is present in the use of all design patterns - is especially problematic. It is impossible to anticipate every kind of change that might be required; even if it were possible, flexibility always comes at a cost in terms of conceptual complexity and/or performance overhead, as the visitor, observer and decorator patterns demonstrate. Enabling for some forms of change inhibits other kinds of change - for example, introducing visitors will promote the future addition of new types of checkers, but it greatly complicates the addition of new types of expressions.

Thus, while design patterns and other design approaches are very useful, they cannot address the issues raised here - their use results in the exchange of one set of problems for another. In some cases, the new set of problems is acceptable, but in others, it is not. As long as the misalignment problem exists, its consequences - weak traceability, low comprehensibility, scattering, tangling, coupling, poor evolvability (including high impact of change and invasive change), reduced concurrency in development, etc. - will be present.

Clearly, the need for a new approach to designing object-oriented software has been motivated. The next section proposes the solution that is the central theme of this research.

## 2.5. Drawing Conclusions for a Solution

As illustrated in this chapter, the structural misalignment of requirements, design and code is at the root of the problems associated with object-oriented designs. Two general approaches exist to addressing the misalignment problem. One is to impose the same development paradigm on *all* software artefacts. This is precisely the approach that has been used to provide close alignment between designs and code - both are written in the object-oriented paradigm. This approach is not appropriate when applied to requirements specifications, however, as requirements deal with concepts in the *user's* domain, while designs and code deal with concepts in the *programming* domain.

The other approach to addressing the misalignment problem is to provide additional means of further decomposing artefacts written in one paradigm so that they can align with those written in another. This approach suggests, for example, that it must be possible to cleanly encapsulate requirements within the object-oriented design paradigm - that is to have object-oriented design models encapsulating requirements units of interest only. This is the approach that is adopted in this thesis, in recognition of the fact that different paradigms are appropriate under different circumstances, so that homogeneity, while appealing, is likely to be inadequate. The approach proposed in this thesis is called *Subject-Oriented Design* and is related to the work on subject-oriented programming, which addressed misalignment and related problems at the code level [Harrison & Ossher 1993], [Ossher et al. 1996].

Like subject-oriented programming, subject-oriented design supports decomposition of object-oriented software into modules, called *subjects*, that cut across classes. For the SEE system, this means that there will be separate design modules for each of the requirements (see Figure 13).

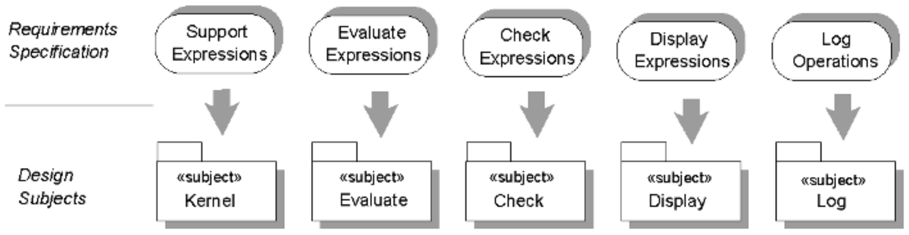


**Figure 13: Matching SEE Requirements with Design Models**

The complexity of understanding the combined impact of multiple requirements on the design of a system is not entirely removed, however, as these separated design models may also be integrated to form complete designs.

See “Proposed Solution” on page 3 for a brief introduction, and “Chapter 4: Composition of OO Designs: The Model” on page 64 for more details.

## 2.6. Chapter Summary

This chapter clearly illustrates that a new approach is needed for object-oriented design. This is because object-oriented designs are difficult to understand, extend and re-use. The chapter outlines and illustrates why this is the case. At the root of the problem is a significant structural mismatch between the units of interest that are the focus of requirements specifications and the units of interest that are the focus of object-oriented specifications.

First the chapter analyses how requirements are specified and how object-oriented designs are specified with the respective motivations for selection of the units of interest discussed. The two paradigms are compared, and a structural mismatch found.

This is followed up with an illustration of how this structural mismatch causes difficulties with the development and evolution of software systems because of the *scattering* and *tangling* effect that is its natural outcome. That is, software system support for a single requirement touches multiple classes in the object-oriented design and code, and a single class in the object-oriented design and code may support multiple different requirements. Even with a small example system, the impact of this mismatch is obvious, with the *scattering* and *tangling* of requirements in the designs reducing the flexibility and comprehensibility of the system, and causing managerial difficulties in the development process. Other design approaches based on Design Patterns are examined, but while some of the problems are solved, their use often involves the exchange of one set of problems for another.

Finally, a new approach to designing systems is proposed that is described in this thesis. This new approach extends the object-oriented design paradigm by adding additional decomposition capabilities that support the designer creating design models that directly encapsulate a single requirement, thereby aligning the designs directly with requirements, and removing the scattering and tangling properties that cause the outlined problems. In the remainder of this thesis, it is illustrated how this solution removes the scattering and tangling properties of standard object-oriented designs, thereby improving comprehensibility, extensibility and reusability. The SEE example is redesigned in “Chapter 9: Applying the Subject-Oriented Design Model” on page 213.

The new approach is called Subject-Oriented Design. The model supports both the new decomposition capabilities and the corresponding composition of design models capabilities, and is described in more detail in “Chapter 4: Composition of OO Designs: The Model” on page 64.

First though, let us examine work related to this thesis (“Chapter 3: Related Work” on page 37). Approaches throughout the software development lifecycle are considered, as the need to decompose large problems, together with the need to integrate them are common problems for each development phase.

# Chapter 3: Related Work

The approach to designing object-oriented software proposed in this thesis is based on providing a new way to decompose (that is, *divide up*) design models, with supporting techniques for identifying overlaps in design units, and for integrating design models. Recognition that decomposition of object-oriented systems by class is necessary, but not sufficient for good software engineering is not new, and this chapter looks at many interesting approaches to extending the manner in which software artefacts are divided up.

Software design can be seen as a bridge between requirements and code, and therefore, it is interesting to consider related work across the development phases of requirements gathering, analysis/design, and coding. The need to decompose artefacts in each phase, together with the need to recognise and identify overlaps in different artefacts, and the need to integrate artefacts, are common problems across the lifecycle. Therefore, each approach in each phase is examined by considering how these needs are catered for. In addition, since one of the integration strategies described in this thesis caters for reconciliation of conflicts, this category of problem is also examined in this chapter.

Related work in the database field is also included. Decomposition of data for database management systems is primarily either based on relational theory or the object-oriented paradigm, and therefore, from a decomposition perspective, the work is not directly relevant for comparison purposes. However, research into integration of heterogeneous schemas has many similarities in the areas of identifying overlapping elements, reconciling conflicts in elements, and integration of schemas.

The chapter is divided up into the following sections:

- Requirements Engineering Models
- Object-Oriented Analysis and Design Models
- Object-Oriented Programming Models
- Database Models

Within each of these four areas, different approaches are discussed based on their approaches to decomposition, identifying overlaps, integration and, in some cases, reconciliation of conflicts. A discussion section follows which assesses the impact of these approaches on the subject-oriented design model.

## 3.1. Requirements Engineering Models

In the requirements phase, requirements are decomposed based on the units of interest to the requirements gatherer. There will also be the units of interest to the person(s) from whom requirements are elicited. This section discusses viewpoints [Easterbrook 1991] [Nuseibeh 1994], use cases [Jacobson et al. 1999], features [Zave 1999] [Turner 1999], and services/facilities [Mowbray & Zahavi 1995] [Siegel 1996].

**Viewpoints and Perspectives**

Using "perspectives" as a unit for decomposition is the focus of the elicitation of requirements in [Easterbrook 1991], where a supporting framework for multi-perspective integration is described in [Nuseibeh et al. 1994]. The model proposed in [Easterbrook 1991] is that "a separate knowledge base is built for each perspective, to capture the knowledge offered by the person expounding that perspective", thus ensuring that "each perspective is properly represented in the integration process". This approach to decomposition is supported in [Nuseibeh et al. 1994], where a ViewPoints framework supports multi-perspective development, with method integration. This framework structures, organises and manages the different perspectives, and also checks consistency, handling inconsistencies between the different perspectives.

The existence of overlaps in the different *perspectives* of requirements for computer systems is central to this approach to requirements gathering. The approach's process of requirements analysis is based on first identifying and developing the different perspectives, but then comparing them to build an understanding of how the different perspectives relate. Though avoiding the "tough problem" of comparing representation schemes, the approach to comparison of the different perspectives is based on the notion that the originators of the different viewpoints are not wholly unfamiliar with the other viewpoints. Therefore, the originators' suggestions of correspondences between the different viewpoints may be used as a basis for discussion of the overlaps. The supporting framework later described in [Nuseibeh 1994] supports the explicit identification of the general relationships between viewpoints with an *inter-ViewPoint relationship*. Through this relationship, overlaps within viewpoints may be identified, and rules governing the overlap specified. Rules, for example, may specify constraints such as existence rules (a ViewPoint requires the existence of another ViewPoint, or of elements within another ViewPoint), or agreement rules (expressing relationships between the contents of Viewpoints), or exclusion rules (for example, uniqueness of names). These rules are the vehicle for viewpoint integration, as they express the relationships between viewpoints, identifying overlaps and defining rules for those overlaps.

The integration of *perspectives* of requirements in this model begins with comparing the different perspective specifications to assess where the overlaps are. Integration of the perspectives is then about resolving any differences between them. A process of in-depth negotiation between all parties involved in each perspective is described. The negotiation process is intended to resolve the differences in the perspectives. In the supporting framework ([Nuseibeh 1994]), integration involves consistency checking of rules defined between different viewpoints - the *inter-viewpoint relationships*. Viewpoints are “consistent” when all the rules defined between them have been found to hold. The notion of consistency is central to the integration objectives - integration *is* achieving consistency. This is different to the notion of integration in the subject-oriented design model, where integration is either integrating the subject design models into one result model, or providing a specification for the integration of supporting subject programs into one result module.

Resolution of conflicts is through a process of education and negotiation between the parties involved in the different perspectives. The model describes three phases: the exploration of the different perspectives, where the participants learn about each other’s perspectives; the generation of suggestions for resolving conflicts; and the evaluation of these suggestions. The supporting framework described in [Nuseibeh 1994] considers resolution primarily as the *handling of inconsistencies*. The view is that forcing consistency may restrict the creativity and inventiveness of the development process, and therefore, to manage rather than restrict inconsistency supports the reality of inconsistencies in the development process. This management of inconsistency takes the form of identification of where inconsistencies exist based on inter-viewpoint relationships, and acting on them based on the use of actions at the meta-level. These actions specify how to act according to the context of the particular inconsistency identified, and are based on temporal logic with temporal operators. An open issue identified within the framework is the actual resolution of conflicts, with the focus described based on identifying and managing inconsistencies.

**Use Cases**

The approach to decomposing and capturing requirements described in [Jacobson et al. 1999] is based on the notion of use cases. A use case outlines who and what will interact with the system, what functionality is expected from the system, and also captures and defines in a glossary common terms that are essential for creating detail descriptions of the system's functionality.

The policy of working with use cases is based on keeping each use case as separate as possible during the requirements phase. The benefits associated with this approach is that each use case is simpler for the software users to understand during requirements elicitation. Consideration of the inherent overlaps associated with use cases therefore becomes more in focus during the analysis and design phases. Here, there is recognition that analysis and design elements such as classes and their objects may participate in many different use cases. This level of overlap is identified through a series of *use case realisations* that have trace dependency relationships from particular use cases to the analysis and design models realising those use cases. No further reasoning is supported for those overlaps.

The notion of integration in relation to use cases is not considered in [Jacobson et al. 1999], as use cases are explicitly independent from each other for the purposes of maintaining comprehensibility for the end-users. Complications associated with overlap in terms of concurrency, conflict or general interferences between use cases are left for consideration in the analysis and design phases. The structural decomposition visible in use cases is not carried through to the analysis and design models, where the object-oriented paradigm of decomposing based on the notion of class, interface etc. is applied. The link between use cases and analysis and design models is maintained through trace dependency relationships, where elements within the analysis and design models may participate in multiple use cases. Explicit integration is therefore not required.

While there is recognition in the use case modelling approach described in [Jacobson et al. 1999] that there may be conflicts and interferences between different use cases, handling of those conflicts is essentially an intellectual effort during the analysis and design phases. Use cases are explicitly maintained and worked with separately during the requirements phase. Object-oriented analysis and design techniques, which are the responsibility of the analyser and designer, apply to handling the impact of the conflicts in the analysis and design models. Solutions are not fed back to the use cases.

**Features**

In [Zave 1999], decomposition of requirements specifications is by “feature”. Features, described as “an optional unit or increment of functionality” [Jackson & Zave 1998], are at the core of the Distributed Feature Composition architecture. DFC is for a telecommunications domain, where features are treated as independent components through which calls are routed from caller to callee. Features are also the core of feature-oriented domain analysis (FODA) where the purpose is to “capture in a model the end-user’s (and customer’s) understanding of the general capabilities of applications in a domain” [Griss et al. 1998].

The need to reason about “features” from the requirements phase and throughout the software lifecycle is the subject of the work on *Feature Engineering* described in [Turner 1999],[Turner et al. 1999]. The definition of feature used in the work on Feature Engineering [Turner 1999] states that “A feature is a clustering or modularization of individual requirements within that [requirements] specification”. The decomposition described at the requirements phase particularly focuses on identifying the features of a system. The approach maintains the perspective of *identification* of features throughout the lifecycle, with the ultimate contribution at the level of configuration management. Here, configuration management supports the developer “checking-out” all the appropriate software artefacts relevant to particular features. This ensures that the impact of any change made is catered for across all artefacts impacted by a feature.

The notion that features may have overlapping requirements is central to the motivation of feature engineering, which therefore has an important need to identify the overlaps. A prototype configuration management tool supports the explicit specification of feature as a first-class construct. Here, features are identified and their relationships detailed. For example, relationships such as `implementedby` associates features with all the components participating in its implementation. This explicitly identifies components that may implement multiple features. Feature relationships such as `competeswith`, `excludes`, and `requires` may be identified to indicate constraints between features. It is not clear, however, how these relationships between features are used. When there is a need to work with components, a check-in/-out procedure is based purely on the `implementedby` relationships between features and components.

**Services and Facilities**

The notion of services and facilities is the basis for decomposition of requirements for a system in the specification of the OMG work on CORBA [Mowbray & Zahavi 1995], [Siegel 1996]. Examples of services a system supporting distributed objects, and conforming to the CORBA standard, should provide are an object naming service and an object event service. Examples of common facilities provided for by CORBA are user interface facilities, and data interchange facilities.

## 3.2. Object-Oriented Analysis and Design Models

At the analysis and design level, there have been many approaches to enhancing the basic object-oriented model. A significant body of work is centred around decomposition based on *roles*. This section talks about three approaches to roles, OORam [Reenskaug et al. 1995], Catalysis [D'Souza & Wills 1998] and an approach described in [Kristensen & Østerbye 1996]. Other interesting approaches to enhancing the basic object-oriented model discussed here are *contracts* from [Helm et al. 1990], *views* from [Shilling & Sweeney 1989] and design patterns [Gamma et al. 1994]. First though, we look at the standard UML, and discuss its existing composition mechanisms.

**Unified Modeling Language (UML)**

The UML is a "language for specifying, constructing, visualizing, and documenting the artifacts of a software-intensive system" [UML 1999]. Structural and behavioural aspects of systems may be captured by a series of different kinds of models - Class, Object, Use case, Sequence, Collaboration, Statechart, Activity, Component and Deployment diagrams. These diagrams present different "views" of underlying structural and behavioural concepts, and may be "combined" into a single design model. The UML metamodel is structured to support such a separation of different "views" into different models. Where one diagram references a model element that is also referenced in another diagram (for example, operations appear in both class diagrams and interaction diagrams), only one specification of that element is supported, with both diagrams referencing the *same* specification. As such, combining diagrams into the same model does not present any conflict difficulties, or merging of behaviours.

The UML 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 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, as described previously for the use of multiple models of the same kind, 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.

**Role Modeling (OORam)**

Role modelling from the OORam software engineering method [Reenskaug et al. 1995] shows how to apply role modelling by describing large systems through a number of distinct models. The designer constructs a role model for each activity or task carried out in the overall system, or constructs several role models for the same activity at different levels of detail. Using this decomposition approach, separation of the design models may be structured to match requirements specifications, where the different roles objects play to support a particular task are distinct in separate role models.

A central notion of role modelling in OORam is the close relationships between the different role models. This is because the same objects often appear in several of them, playing different roles. Synthesis in OORam is at the level of role models (not on single roles) so an explicit specification of the mapping of all roles in base models to roles in a derived model is required. This is supported by an OORam language which has constructs to identify derived and base models, and the explicit mapping between roles. This serves to support the identification of those roles that overlap in the sense that they should be synthesised in the derived model.

Integration (or synthesis) in OORam is based on synthesising base role models into a derived model. Every base role in a base model is synthesised into a derived role in the derived model. Base model semantics are retained in derived models. While static correctness of the derived model is achievable, approaches to ensuring dynamic correctness of the derived model are less clear. This is recognised within the OORam model, and approaches to ensure "safe" synthesis limit the possibilities available for integration. Two examples of approaches to safe synthesis for dynamic behaviour are defined. The first is called *Activity Superposition* where each base model activity is retained unchanged in the derived model. The second is called *Activity Aggregation* where a base model activity is changed to include the execution of another base model activity in the derived model. As described in [Andersen & Reenskaug 1992], the synthesised role models form the basis for the type requirements of the classes implementing the design.

**Catalysis**

Another approach to role modelling, based on the UML, is Catalysis [D'Souza & Wills 1998]. Catalysis separates design models according to concerns, using horizontal and vertical slices. Vertical slices decompose models according to the point of view of different categories of users. The approach yields different models of the same types and actions. Horizontal slices decompose based on separating technical infrastructures and communications protocols from the business models. This approach to decomposition supports structuring design models to align with both business requirements, and more technical kinds of requirements that may have an impact across all of the business requirements.

In Catalysis, the joining of package specifications is based, by default, on joining those definitions with the same name. Exceptions to this may be explicitly specified using extra invariants which may state that two definitions with different names should be considered to map together, and explicitly stating the name to be used in the result. This approach can also be used to state that two definitions with the same name should *not* map together, by explicitly renaming one of them.

Integration is based on a definition of the UML *import* relationship, called *join*. In general, the resulting definition for each type of element in a package contains the combined set of elements that are defined for that type. For example, a set of all the attributes from the joined packages appears. Constraints are `and`-ed, including preconditions, postconditions, rely conditions and guarantee conditions.

**Role Modelling (Kristensen)**

The approach to role modelling described in [Kristensen & Østerbye 1996] decomposes based on the separation of an object's intrinsic properties from the roles that an object may play. These roles are entities that may contain additional state and behaviour, and are attached to the base object. The analysis of a system may be in terms of the roles of objects, lifting roles to a primary consideration in the design. This supports the structuring of analysis models to match with requirements that specify different tasks to be performed by the same objects playing different roles.

With this approach to role modelling, roles are explicitly related to particular intrinsic objects to which they add role behaviour. The notion of roles working with particular core concepts (and therefore overlapping) is explicitly identified at design time. Though roles have state and behaviour, they may not exist independently (i.e. they do not have identity), and must be attached to intrinsic objects. Multiple role objects may be attached to intrinsic objects, and may be referenced by a single reference to groups of those roles called a *subject reference*. However, a restriction exists that does not allow for overlap between those roles. Though this is recognised as a restriction, it ensures that a remote access through a subject reference, which may reference multiple different roles, is always well defined.

Roles may be aggregated for an intrinsic object. For example, a Professor may be an aggregation of Teacher and Researcher roles. The separation of role specifications from intrinsic object specifications supports the dynamic attachment of roles to different objects at different times. A *subject* is seen as an instantiation of a class with roles, and in this sense, is an integration of a class with particular roles. As described in [Kristensen & Østerbye 1996], restrictions apply on the naming of roles involved in a subject instantiation, for the purposes of avoiding name collisions. An extension to this restriction is described in [Kristensen 1997], where, for the purposes of composition of hierarchies (both role and class), like-named roles and classes are considered to be *the same* and their integration supported only where the resulting hierarchy does not contain cycles.

**Contracts**

A different approach to decomposition of analysis and design models specifies components called contracts, where the focus is on decomposition in an *interaction-oriented* way [Helm et al. 1990], [Holland 1992]. Contracts specify behavioural compositions and obligations on participants. They capture explicitly and abstractly the behavioural dependencies amongst collaborating objects. *Contract specification* identifies the participants in a behavioural composition and their contractual obligations. *Contract conformance* checks classes to ensure that they behave appropriately relative to all the contracts in which they participate. *Contract instantiation* creates objects at run time that interact as described by the contract.

With this approach, contracts are defined independently of classes, and specify the contractual obligations of participants in the contract - therefore, the notion of “overlap” is not an issue. Decomposition is based on separating the specifications for behavioural interactions between collaborating objects, where the identification of objects that conform to the contract specification is done with an explicit “conformance” specification stage. Once an object has been deemed to conform to the contractual obligations of a particular participant in the contract, then it may be instantiated as that participant and behaves as defined by the contract. In the sense where an “overlap” may be seen as a specification of a correspondence, then the specification of class mappings to contracts (with contract conformance declarations) may be seen as the specification of correspondence to a contract participant.

Behavioural compositions specifying the interactions of collaborating objects are specified with contracts. Contracts define the obligations of participants in a contract in terms of the variables, external interfaces and sequences of actions which must be supported in order to participate. Basic contracts can be further composed to specify more complex behavioural specifications with contract *refinement* and *inclusion*. Refinement supports the specialisation of contract specifications, with extensions to its actions or invariants. Contract inclusion supports the union of contract specifications, thereby allowing multiple contract specifications to be composed to more complex specifications. In terms of creating behavioural compositions of objects that participate collaboratively as defined by a contract, this is done through the instantiation of contracts. This requires the identification of objects as participants, and establishing the contract via the methods defined in the contract.

**Views**

Some approaches to extending the decomposition of object-oriented systems are based on the notion of “views” - for example, [Shilling & Sweeney 1989]. Here, large, complex systems may be decomposed based on the “view” of the user. The basis of this architecture relies on extending the object-oriented paradigm in three steps: 1) defining multiple interfaces in object classes; 2) controlling visibility of instance variables; and 3) allowing multiple copies of an instance variable to occur within an object instance. These object extensions are used to create view classes and view instances. A view class is a global abstraction which uses many object classes to provide a unified global behaviour. A view class is defined as a set of ordered pairs of the form *(object class, interface)*. A single object may participate in many view classes. This allows view instances to intersect. The object class specifies how view instances interact by its rules for sharing and accessing instance variables.

Specification of the control of overlaps in this “Views” model is contained in the object classes that participate in the View - that is, the global abstraction of multiple collaborating objects. Each object class has rules for sharing and accessing instance variables by explicitly stating the particular interfaces that may access instance variables and methods. The identification of overlaps (or, corresponding elements) is therefore defined for each class as part of the specification of the different interfaces the class supports. It is the responsibility of the specifier of the view class - that is, the set of ordered pairs (*object class, interface*) that participate in the view - to ensure that the view class is coherent in its inclusion of the appropriate pairs to support the required view.

A View Class specifies the composition of objects important to a particular view with its ordered set of tuples (*object class, interface*). Instantiation of objects is only in the context of an instance of a view class. Composition of the objects is by *joining* each object instance to the view instance. The parts of the objects (interfaces and instance variables) visible to the view are as specified by the view class (interface), and the object class (instance variables). It is the responsibility of the view class designer to ensure that the set of (*object class, interface*) tuples that make up the view is a set that makes sense to support the particular requirement of the view.

**Design Patterns**

The decomposition focus of design patterns [Gamma et al. 1994] is on enabling the design of reusable, extensible software. To this end, decomposition is based on isolating different aspects of a problem into separate design units. Different patterns support this approach from different perspectives; for example, structural decomposition is supported with a Decorator pattern that separates extensions to an object’s functionality in an alternative to subclassing, and behavioural decomposition is supported with the Visitor pattern that supports the definition of new operations without changing the classes of the elements on which it operates. Depending on the kind of separation required in a particular design situation, an appropriate design pattern is chosen and applied.

The notion of overlap is catered for explicitly in design patterns. The level of decomposition for each of the design patterns, where structural, behavioural or creational issues may be decomposed separately from core objects, is designed into the suite of collaborating design elements supporting that pattern. The pattern of collaboration between the appropriate design elements explicitly caters for the overlapping of concepts. Therefore, the identification of corresponding elements which must work together is an essential part of each pattern.

The specification of each design pattern in [Gamma et al. 1994] includes how the appropriate collaborating objects to support a particular design pattern are integrated. Integration is not explicit in the sense of synthesis into a single result, but rather, it is a specification of collaboration of appropriate objects to achieve the goal of the particular design pattern. The level of integration in this sense is explicitly designed into the classes that are identified as participating in the design pattern.

## 3.3. Object-Oriented Programming Models

Approaches to enhancing the object-oriented decomposition paradigm are also prevalent in different programming models. This section discusses *subject-oriented programming* [Harrison et al. 1996], *aspect-oriented programming* [Kiczales et al. 1997], *composition filters* [Aksit et al. 1992], *adaptive software* [Lieberherr 1995] and *metaobject protocols* [Kiczales et al. 1991].

**Subject-Oriented Programming**

Hyper/J™ [Tarr & Ossher 2000] supports what they term “multi-dimensional separation of concerns” [Tarr et al. 1999]. This is an approach to decomposing software into modules, each of which contains the code for (thereby encapsulating) a particular area of interest. These modules are called hyperslices. Examples of the areas of interest that motivate this level of decomposition are functions, data types/classes, features (e.g. “persistence”, “print”, “concurrency control”) and roles. Developers can write separate programs in Java™ to support this decomposition. This work has evolved from the work on subject-oriented programming [Harrison & Ossher 1993], [Ossher et al. 1996].

The modules that implement different units of interest (hyperslices) in Hyper/J are composed by identifying *corresponding* units in different hyperslices, and integrating them. The relationships between corresponding units in different modules are identified in a specification file that has two main parts:

1. it explicitly names the hyperslices involved in the composition (keyword `hyperslices`);
2. it identifies the corresponding units within these hyperslices, and how they are to be integrated (keyword `relationships`).

Some relationships identify the corresponding elements by combining the matching criteria with the integration criteria. For example, `mergeByName` specified in the relationships part of the specification file indicates that units with the same name correspond, and should be merged. Other relationships just identify units that correspond, without an indication of how they should be integrated. For example, the `equate` relationship indicates that a set of units match each other, and the `match` relationship provides a more flexible pattern matching with wild cards. The composition process uses these relationships to identify the units within the different hyperslices that correspond.

This separate specification file is the means for specifying integration of hyperslices. This file identifies the hyperslices to be composed, the units within the hyperslices that correspond, and how they are to be integrated. Integration relationships such as merge and override specify different kinds of integration strategies for corresponding units. `merge` indicates that corresponding units are to be integrated together into a single unit. `override` causes one unit to replace other corresponding units. The actual integration is performed by Hyper/J, the result of which is a composed Java program containing the combination of the input hyperslices as defined by the integration strategy.

**Aspect-Oriented Programming**

Decomposition based on “aspects” is the approach taken in AspectJ™ [Kiczales & Lopes 1999], where an aspect is a unit of interest that “cross-cuts” another unit of interest. Two units of interest cross-cut each other when the available decomposition paradigm supports the encapsulation of one unit of interest, but this presents difficulties in cleanly localising the other. Examples of a cross-cutting unit of interest are “persistence”, “concurrency control” and “distribution”. With AspectJ, such cross-cutting units of interest can be encapsulated, and coded (in Java) separately from the rest of the code. The approach is called “aspect-oriented programming” [Kiczales et al. 1997]. The existence of a “base” program into which aspect code is weaved is the primary difference in the approaches of subject-oriented programming (and therefore, decomposition in subject-oriented design) and aspect-oriented programming. In subject-oriented programming, there is no concept of a base program - each code subject is independent, and completely provides the code for the particular unit of decomposition supported.

AspectJ has extended Java to support constructs that implement the *aspects* that cross-cut programs. The overlaps with standard Java programs are explicitly and clearly defined with new Java language constructs that support:

1. The identification of the points in the base Java program (such as types, messages, instantiations, exceptions or members) where the aspect program defines actions that may be performed on those points. The keyword `crosscut` is used here.
2. The aspect program also specifies the actions to be performed on the identified points, and controls *when* these actions are performed with new keywords - some examples of which are `before`, `after`, `finally` and `catch`.

Integration in aspect-oriented programming using AspectJ is performed at compilation time. The source `.java` aspect and class files are input to an aspect compiler that "weaves" the input source files, and produces Java code containing the integration of the aspect code and the class code. The weaver generates the output Java code based on the specification in the input aspect files. The aspect files indicate the exact points in the class files that have additional actions specified, and where those actions should be integrated (e.g. before, after etc.). The generated Java code may then be compiled with a standard Java compiler.

**Composition Filters**

"Composition filters" are the approach to decomposition described in [Aksit et al. 1992], where decomposition based on "views" integrates database-like features with the object-oriented model. Views are supported with "filters" which are part of the definition of a class. Filters define the guidelines for an object's behaviour and have two components: a filter handler that determines what is to be done with messages, and an accept-set function that defines the conditions under which messages to the object are accepted. Multiple views are defined in terms of filters, where a client object is examined to determine the behaviour to which it has access. Different filters may be defined for each class to support different kinds of views - for example, concurrency or synchronisation. Each filter is responsible for handling all aspects of its associated view. Since both message sends and receives are trapped by filters, filters can perform certain actions relevant for its view, before the actual method is executed. This approach differs from the subject-oriented approach primarily in its handling of separation for a single class, where multiple, collaborating classes are separated into subjects.

In this approach, filters are explicitly attached to class definitions in the language. Each class definition defines the behaviour of any filters on receipt of incoming messages, and the behaviour that may be defined as a result of outgoing messages. Further identification of overlaps is not required.

Integration involves integrating the filters that contain the additional constraints or behaviour to support the separated units of interest. The effect of integrating filters is essentially to `and` them together, with messages being accepted or rejected in a sequential manner.

**Adaptive Software**

The problem with standard object-oriented programming languages addressed by adaptive software [Lieberherr 1995] is the impact of attaching methods to classes. The impact is that the details of the class structure for collaborating objects are encoded into the program. This means that programs are hard to evolve and maintain as changing the class structure requires changes to all code that explicitly refers to that structure. Adaptive software decomposes programs by separating the algorithms on data into code patterns. These patterns, called *propagation patterns*, interact with a class dictionary that defines class structure with minimal dependency on that structure. Minimal dependency is achieved because propagation patterns containing algorithms only refer to class structures implicitly through a level of indirection from the actual class structure, called a *propagation graph*. The propagation graph provides the succinct specification of the group of collaborating classes required for the algorithm in the propagation pattern. This level of decomposition protects the algorithms from changes to the base class structure, minimising the impact of changes.

So, we have *propagation patterns* that implement functionality for groups of collaborating classes, and *propagation graphs* that specify what those classes are. The identification of overlap required to integrate the algorithms with the classes is done with *propagation directives*. How classes should be traversed to suit the algorithms is specified by the propagation directives. The correspondence (or overlap) of the collaborating classes with the appropriate algorithm is specified when a propagation pattern uses the propagation directive specifying the collaborating group and its traversal.

Integration in adaptive software systems with propagation patterns is performed at compile time. The pattern compiler integrates a class hierarchy with algorithms defined in propagation pattern wrappers, as defined by a propagation directive which specifies the traversal through the appropriate collaborating classes. A "wrapper" specification within a propagation pattern may express combinations of methods, where the generated code resulting from the compilation simulates multiple inheritance within the class hierarchy.

**Metaobject Protocols**

The separation of base and meta-levels of programs is the focus for decomposition with metalevel programming. The interface between the base-level and meta-level programs is achieved with *metaobject protocols* [Kiczales et al. 1991]. Metaobject protocols are interfaces to the programming language that allow programmers to customise the behaviour and implementation of programming languages and other system software. Metaobjects trap message sends and receives to objects, and can therefore supplement the behaviour of operations at the base level. With this level of separation, metaobjects may contain support for distribution of objects, concurrency, etc., thereby neatly separating such concerns from the base-level algorithms of the object. However, further decomposition at the meta-level remains an open issue, as it is not possible to separate, for example, distribution support from concurrency support if both are required for the base object. Aspect-oriented programming can be seen as an outgrowth of this work, where decomposition based on any kind of cross-cutting activity is possible.

As described in [Kiczales et al. 1991], metaobjects are defined by metaobject classes, where, for each kind of programming construct (e.g. class, method), a basic metaobject class may be defined. These basic metaobject classes may be further specialised and attached to standard base classes to extend their behaviour. One implementation of this for C++ is defined in [Gowing & Cahill 1996], where categories of possible metaobject classes for C++ have been defined (for example, object creation, method invocation etc.). A programmer may specialise metaobject classes within these categories, defining additional state and behaviour. The notion of identifying overlaps is handled explicitly, where base objects requiring any additional behaviour within the defined categories are explicitly associated with the relevant metaobject(s).

Integration in metaobject protocols amounts to simply attaching the appropriate metaobjects to the base level objects [Kiczales et al. 1991]. Each programming language that handles the specification of metaobject classes has, generally, been extended to support the relationship between the defined metaobjects and base objects, and therefore executes the required meta-behaviour on invocation of the appropriate programming language construct in the base object - for example, object instantiation, method entry, or method exit etc.

## 3.4. Database Models

The means to manage data within an organisation with database management systems first emerged in the late 1960's [Bell & Grimson 1992]. The motivations for decomposing data in different ways were many - for example, to eliminate duplication of data, to avoid problems associated with multiple updates of data, and to minimise inconsistencies across applications [Batini et al. 1986]. Different approaches to decomposition over the decades from the 1960's have been described as first, second and third generation [Stonebraker et al. 1991].

Network and hierarchical database systems were classified as "first-generation" and were prevalent in the 1970's. However, due to the complexity of navigation, these first generation approaches to data management were largely replaced by the "second-generation" of database management systems - relational databases. Decomposition of data in the relational model is in two-dimensional structures known as *tables* or *relations* [Bell & Grimson 1992]. Relational database technology has a strong theoretical basis in mathematical relational theory, and has proven a successful approach to data management. However, because of a perceived limitation in supporting a broader base of applications [Stonebraker et al. 1991], a third generation of database management systems were born, based on the object-oriented paradigm.

Different attempts at defining an object-oriented database management system are described in manifestos from [Stonebraker et al. 1991], and from [Atkinson et al. 1990]. In summary, object-oriented databases manage complex objects, with object identity, and support standard object-oriented principles of encapsulation and inheritance. Other features and characteristics required of object-oriented databases are computational completeness, persistence, concurrency, recovery and an ad-hoc query facility. Object-oriented database management systems follow the structural decomposition paradigms of object-oriented analysis, design and coding paradigms.

From the point of view of decomposition, modern database management systems are primarily either based on relational theory or the object-oriented paradigm. Therefore, from a decomposition perspective, the work is not directly relevant for comparison purposes. The approach proposed in this thesis is motivated by problems with the object-oriented paradigm, and therefore more relevant related work is in areas where the object-oriented paradigm is being extended. However, research into integration of heterogeneous schemas has many similarities in the areas of identifying overlapping elements, reconciling conflicts in elements, and integration of schemas, and therefore this discussion on decomposition in database management systems is useful.

*Reference Architecture for Schema Integration*

An environment with multiple heterogeneous databases, where data is required from each of these different sources, needs an architecture whereby any required data may be integrated, regardless of the source of that data. In [Sheth & Larson 1990], a reference architecture is defined, from which federated database systems (that is, a collection of cooperating database systems that are autonomous and possibly heterogeneous) may be developed. The reference architecture includes descriptions of components that have responsibilities for mapping the schemas from different databases and for checking constraints and integrating data from the different sources. The five-level schema architecture described defines the steps the schemas from different databases go through, from the *local* schema that is private to a component database system of the federation, to the *external* schema that contains data required by a user and/or application. From the perspective of the work that is related to this thesis, the focus is on levels that have integration and reconciliation elements.

*Identifying Overlaps*

In the federated database system architecture described in [Sheth & Larson 1990], schema translation and schema analysis steps provide the means to examine component database systems for overlaps. Where database systems are described using different data models (that is, Common Data Models (CDMs) or different “languages”) schema translation supports the translation of the different models into a uniform CDM, aiding the analysis step since it is easier to compare data described in the same language, than it is to compare data described in different languages. Schema analysis involves comparing the objects in the schema prior to integration, and identifying naming and domain conflicts, structural and constraint differences, and missing data. The identification of the overlaps in the different schemas involves specifying the interrelationships among the schema objects.

Research into integrating database schemas generally conforms to an architecture of identifying overlaps between different schemas and integrating the schemas to provide a single view. From the perspective of identifying overlapping elements within different schemas, approaches vary in the extent to which they automate the process, and the extent to which they support the use of heuristics for identifying overlap. For example, in [Sheth et al. 1993], the relationships between attributes in different schema are identified by a human with *attribute relationships*, but these are considered only a partial identification of overlapping elements. Generation of an attribute hierarchy is supported, further establishing semantics equivalence between attributes.

In general, explicit identification of overlapping elements is prevalent in database schema integration approaches. For example, there are *articulation axioms* from [Collet et al. 1991], *inter-schema correspondence assertions* from [Spaccapietra et al. 1992], *assumption predicates* from [Gotthard et al. 1992], pairing of user-defined *vertices* from *schema graphs* in [Klas et al. 1996] and *object correspondence assertions* from [Navathe & Savasere 1996]. Correspondence types identified in [Navathe & Savasere 1996] are defined as equivalence, contains, contained-in, overlap, disjoint, aggregate and composite. Similarly in [Bertino & Illarramendi 1996], correspondence types are defined as equivalence, inclusion, overlapping and disjoint. In each of these approaches, varying levels of explicit identification and heuristics to support the general identification of possible overlaps are applied, with the integrator confirming or rejecting results from the general heuristics.

*Integrating Schemas*

In the reference federated database system architecture described in [Sheth & Larson 1990], a “federated schema” is the integration of multiple export schemas from component databases. Export schemas are the subset of the component schema (that is, local schema translated to a common data model) that is made available to the federated database system. Implementations of the reference architecture must have a schema integration step that may include automated integration based on the relationships previously defined between the component schema during an analysis for the identification of overlaps, and also, support for a more interactive integration process whereby a user may be guided through a process of defining equivalences for integration. Issues with integrating schemas from the point of view of differences in data representation are identified in [Bright et al. 1992] as: 1) naming differences (synonyms, homonyms); 2) format differences (data types, domain, scale, precision); 3) structural differences (single v. multiple values, differences in types); 4) missing or conflicting data (conflicts in actual data values stored). Approaches to integrating database schema described in this section, in general, contend with these issues.

However, once correspondences in different schema have been established, approaches to integration of schemas are based on *merging* schemas in different ways. [Navathe & Savasere 1996] describe a number of different merging operators that contain strategies to handle the merging of pairs of objects (established as corresponding) as appropriate to their types, and the extent to which their merging requires support to handle conflicts between them. The merging strategies range, for example, from adding a generalisation or specialisation object to capture common attributes and/or their constraints, to the creation of a new entity to contain the union of all attributes. Some restructuring operators are also included where new entity types may be created (or deleted) in the composed schema where necessary. Automated class integration based on formal reasoning is described in [Sheth et al. 1993], where the attribute relationships defined by the user to specify corresponding attributes are used as the basis for formal use of a *classification* algorithm which is based on the semantics of class *subsumption* - that is, whether a class is a superclass of another.

These two approaches are good representatives of the general approaches to integrating schemas - transformation (or some level of structural enhancement) of schemas is a common theme, as also is the use of formal heuristics for some level of automation of the union of schemas.

*Resolving Conflicts*

Conflicts in heterogeneous database models can arise as a result of “systems” reasons (where the hardware, operating system, database management system, transaction management system, or communications protocols are different) or for “semantic” reasons (where there are differences in the way data is modelled, resulting in conflicts in database schemas). Subsuming earlier work on classifying heterogeneities in relational multidatabase systems (for example [Kim & Seo 1991]), [García-Solaco et al. 1996] classifies numerous categories where semantic heterogeneities may arise - namely, differences in extensions (i.e. instances of classes), differences in attributes, methods and names, differences in domains and differences in constraints. This work concludes that detection of semantic heterogeneities is “the most critical task of the reconciliation”, and that it is not possible to fully automate the process due primarily to incompleteness of design methodologies, semantic poorness of local/component schemas, and also because some semantics can only be determined with respect to a particular context that may only be known to the integrator. However, while human intervention is unavoidable, some measure of automation is possible. Therefore this section looks at some representative work in the database field in the area of automating the reconciliation of conflicts in database schemas.

Research into the automation of *mapping* of information from input schemas to integrated schemas is the focus of the approaches in [Härder et al. 1999] and in [Spaccapietra & Parent 1994]. A mapping language, called BRIITY, is described in [Härder et al. 1999], which has been designed to “bridge heterogeneity”. For each classification of conflict, the mapping language has rules to define how each conflict should be resolved. These rules are based on combining the object-oriented paradigm with set theory from relational databases to establish relationships between entities and attributes of the instances of different schemas. The language has explicit constructs to identify the mappings between the types and entities of different schemas. The approach to mapping described in [Spaccapietra & Parent 1994] is based on correspondence assertions defined between related constructs in different schemas. For each assertion, formal rules state how to derive the constructs to be inserted into an integrated schema. Where conflicts exist in corresponding entities, the integration holds the least restrictive representation.

Another interesting and different approach to automated resolution of semantic heterogeneity is based on the use of on-line linguistic tools to interpret a user’s imprecise language in requesting data [Bright et al. 1994]. First, a global data structure is built relating local access terms which are semantically similar. Then, using this global structure and on-line linguistic tools, the user’s imprecise query is interpreted and associated with the precise local system access terms that are semantically closest. This is not the same as resolution in the subject-oriented design sense of resolving to a single output, but is an interesting approach to being as flexible as possible from a user’s perspective.

## 3.5. Discussion

As stated previously, the fundamental goal governing this work is to extend the decomposition capabilities of software artefacts, as applied to software designs. In support of this, the identification of overlaps in different design models, the integration of design models, and the reconciliation of conflicts between design models is required. For this reason, the discussion in the previous sections focused, where appropriate, on how each approach handled these areas. See Table 1 for a summary.

| | Decomposition | Identifying Overlaps | Integration | Reconciling Conflicts |
|---|---|---|---|---|
| *Requirements Engineering Models* | | | | |
| Viewpoints | Capture of perspective of requirements from individuals | Relationships between viewpoints explicitly defined with inter-Viewpoint relationship with rules to govern overlaps | Integration based on negotiation of perspectives, and consistency checking of inter-Viewpoint relationships. Integration is achieving consistency | At gathering phase, based on negotiation and understanding of perspectives. Supporting framework based on managing inconsistencies |
| Use Cases | Based on functionality expected from system | Use cases are kept separate. | | Analysis phase handles inconsistencies |
| Features (Zave) | In telecommunications domain, based on unit of functionality | | | |
| Features (Turner) | Modularisation based on individual requirement | Explicit association of features with system components with an *implementedBy* keyword | | |
| Services/ Facilities | Technical kinds of services - for example, object naming and object events | | | |
| *Object-Oriented Analysis and Design Models* | | | | |
| OORam | Role model for each activity or task | Language defined with explicit constructs to identify mappings between roles in different role models | Base role models are synthesised into a derived model. A notion of "safe" synthesis limits possibilities for integration - two possibilities: activity superposition (each activity retained unchanged) and activity aggregation (activity changed to include execution of another) | |
| Catalysis | Horizontal and vertical slices for different kinds of functionality | Joining generally based on "same name" correspondence, with invariants to define exceptions possible | Based on a definition of UML *import* relationship called *join*. Result contains combined set of elements with constraints *and*-ed | |
| Role Modelling (Kristensen) | Separation of intrinsic object from role object plays | Overlaps defined at design time, with explicit attachment of roles to intrinsic objects. Like-named roles and classes are also considered to be the same. | Roles may be aggregated for a single intrinsic object. Integration of like-named roles and classes only possible when result does not contain cycles. | |

**Table 1: Summary of Related Work**

| | Decomposition | Identifying Overlaps | Integration | Reconciling Conflicts |
|---|---|---|---|---|
| Contracts | A contract separates specification of behavioural compositions and obligations on participants | Classes may be explicitly mapped to contracts, deeming that class as a participant in the contract | By instantiation of contracts, behavioural compositions of collaborating objects are created. Contracts may be composed to define more complex specifications | |
| Views | System decomposed based on "view" of user, with definitions of different interfaces, and variable visibilities and copies for different views within each class | View classes define the set of object classes and interfaces of the required set of collaborating classes. | View Classes specify composition of objects relevant for a particular view. Instantiation of objects is in context of an instance of a view class. | |
| Design Patterns | Isolates different parts of a problem in areas such as structural, behavioural and creational concerns | The interaction of collaborating objects is defined as part of each pattern | Integration not explicit in the sense of synthesis to single result, but as a specification of collaborating classes | |
| *Object-Oriented Programming Models* | | | | |
| Subject-oriented programming | Different modules contain code for different areas of concern along multiple dimensions | Corresponding units in different modules (hyperslices) are defined with relationships | Integration strategy defined with explicit keywords (e.g. *merge, override)* with input modules composed by a compositor that produces an output module. | |
| Aspect-oriented programming | Separates cross-cutting concerns (such as distribution) into separate modules | Aspect language constructs (keyword *crosscut)* specify the parts of the base program affected by an aspect | Integration performed at compile time, with aspect coded weaved in with the based program as specified by the aspect program. | |
| Composition filters | "Filters" support views on classes by defining what is to be done with messages, and the conditions under which messages are accepted | Filters are attached to class definitions (supported by language constructs) | Filters may be integrated (that is *and*-ed) with tests for acceptance of message through the filters in a sequential manner | |
| Adaptive Software | Separates algorithms from the data on which algorithms work, using a level of indirection to work with the class structure required. | *Propagation directives* contain information on the class hierarchy and how it should be traversed by the algorithm (*propagation pattern)* | Performed at compile time, the pattern compiler integrates the algorithms with the class hierarchy as defined by propagation directives | |
| Metaobject Protocols | Base and meta-levels of classes are separated, with metaobject protocols supporting the trapping of messages to an object, for enhancement | Meta-objects and base objects are explicitly associated with supporting language constructs | Integration simple attaches the appropriate metaobjects to the base objects. | |

**Table 1: Summary of Related Work**

| | Decomposition | Identifying Overlaps | Integration | Reconciling Conflicts |
|---|---|---|---|---|
| *Database Models* | | | | |
| | By the third generation of database models, decomposition is based on the standard object-oriented paradigm | Explicit identification of corresponding elements is prevalent with schema integration models - for example with correspondence assertions, assumption predicates, articulation axioms, etc. | Integration is based on schema *union.* Approaches are generally based on *transformation* (some level of structural enhancement) or *formal heuristics* for the automation of the union of schemas. | Research into classifications of different kinds of heterogeneity basis for heuristics of mapping input to output to avoid conflict. This tends to involve transformation. |

**Table 1: Summary of Related Work**

In general, decomposition in requirements engineering models is based on units of relevance to the end user, or on units of relevance for the technical environment. This makes sense, as requirements are generally gathered from end-users, or defined to support a particular environment. It is important, therefore, for validation purposes, that the requirements specification be in a language understood by the end-users, and that their concerns are the primary units of specification. For this reason, it is unlikely that requirements engineering research will radically change how requirements specifications are decomposed in the future.

Compared with the requirements model, there appears to be more flexibility in the approaches to decomposition in analysis and design models. Research in this field is most notable for the interesting ways of attempting to divide up design artefacts. The goals for these attempts are, in general, similar to each other, with approaches trying to make software designs more re-usable, extensible, and comprehensible. Subject-oriented design has these goals in common with many approaches. In general, subject-oriented design distinguishes itself with its support for different kinds of integration of overlapping concepts, thereby enabling more flexible kinds of decomposition.

The approach to decomposition in role modelling in OORam [Reenskaug et al. 1995] is subsumed by the approach taken in the research described in this thesis. Additional decomposition capabilities for technical kinds of concerns are possible with the subject-oriented design model. There are also strong similarities with Catalysis [D’Souza & Wills 1998], with vertical and horizontal slices similar to functional and cross-cutting decompositions. Where the subject-oriented design model distinguishes itself is primarily in its support for different kinds of integration, and its support for specifying patterns of collaborating design elements. The more sophisticated resolution and integration capabilities in subject-oriented design, especially of overlapping elements, support extensions to the decomposition capabilities in Catalysis. The approach to role modelling from [Kristensen & Østerbye 1996] is different from subject-oriented design in the specification of the intrinsic object to which roles are attached. With the subject-oriented design model, there is flexibility for evolving the properties of an object over time, by specifying new or changed properties and composing them with previous versions of an object.

Differences are more significant in the approaches to contracts, views and design patterns. With contracts, decomposition is based on supporting composition of objects as opposed to composition of classes as defined in the subject-oriented design model. This is also true of views [Shilling & Sweeney 1989], while design patterns do not have a notion of overlapping specifications, or integration of designs.

From the cited work within programming models, the subject-oriented design model most emulates the approach supported for code by Hyper/J [Tarr & Ossher 2000]. *Hyperslices* are modules that implement different units of interest, and are directly analogous to design subjects in the subject-oriented design model. The ideas within the subject-oriented design model based on the specification of overlaps (corresponding elements) within different subjects, and the approaches to integrating subjects are based on those within this programming model. At the highest level, where subject-oriented design distinguishes itself (aside from working with designs instead of code) is primarily in the ability to specify patterns of collaborating design elements. At a more detailed level, there are other differences between the rules and capabilities of composition relationships (subject-oriented design) and composition rules (subject-oriented programming). See "Composition of OO Designs: The Model" on page 64 for more details.

Also from the cited work within programming models, the aspect-oriented programming approach [Kiczales et al. 1997] has many similarities with subject-oriented design in terms of the goals that each is trying to achieve. Cross-cutting concerns are separated from "base programs" within the aspect-oriented programming model. Cross-cutting concerns may also be designed as a separate design subject within the subject-oriented design model. However, as in subject-oriented programming, subject-oriented design also does not have the notion of a "base design". Each requirement or area of interest may be designed separately, including functional requirements that are all implemented in the "base program" within aspect-oriented programming. This also has implications for composition specification, as in the subject model (both design and programming), composition specification is separate from the individual subjects, whereas in the aspect model, how aspects are composed with a particular base program is part of the aspect program which also contains the cross-cutting behaviour specification. Nonetheless, the goals of both approaches are sufficiently similar to warrant investigation into the applicability of the aspect-oriented programming implementation as a supporting technology for subject-oriented design (see “Future Work” on page 253 for more details).

Differences with the other programming models are more significant. Composition filters decompose units of interest on a per-class basis [Aksit et al. 1992], whereas the subject approach decomposes based on units of interest of groups of collaborating classes. The more sophisticated reconciliation and integration capabilities of the subject-oriented design model allow more flexibility of decomposition than is available in adaptive software [Lieberherr 1995]. While metaobjects permit the separation of the base and metaobjects, it is not possible to compose metaobjects, and therefore further decomposition of metaobjects to implement different functionality is not possible [Kiczales et al. 1991].

Resolving conflicts in overlapping entities has been the focus of some work in the requirements engineering and the database fields particularly.Work in the analysis/design and programming fields tends to restrict the kinds of overlaps possible to ensure that conflicting elements are not integrated. However, in the requirements engineering field, it is particularly important to attempt to resolve conflicting requirements, as it is not possible to restrict the kinds of requirements that end-users want to include. As a result, it is not possible to avoid the possibility of there being conflicting requirements. These conflicts must be resolved prior to completion of the requirements specifications.

In the database field, the core problem addressed in current research is based on the assumption of heterogeneity in schemas to be integrated. Therefore, algorithms and processes for the resolution of conflicts in heterogeneous schemas are the focus of much research. The subject-oriented design model proposed by this thesis allows differences in specifications of overlapping design models, and therefore reconciliation of potential conflicts is required where corresponding elements are to be integrated into a single element (this occurs in *merge* integration).

## 3.6. Chapter Summary

This chapter examines research work related to the new approach to object-oriented design proposed in this thesis, called subject-oriented design. Since software design can be seen as a bridge between requirements and code, research has been examined within the fields of requirements engineering, object-oriented analysis and design, object-oriented programming and data-base management systems. While the focus of the research described in this thesis is the object-oriented design phase, the very fact that the structures of the artefacts from phases across the lifecycle are fundamentally different is the root cause of many of the problems motivating subject-oriented design. Therefore, because of the “bridge” nature of design, it is particularly interesting to examine the manner in which software artefacts are structured in the different phases.

Within these areas, the themes used to analyse different approaches are based on the primary areas of focus for subject-oriented design - they are: decomposition; identification of overlap; integration; and reconciliation of conflict. In this way, there is an emphasis on the particular parts of related areas of work that are specifically related to the different parts of subject-oriented design. This serves to highlight similarities and differences in a focused way.

From the volume of research that exists for improving and extending the object-oriented paradigm, it may be deduced that there is considerable recognition of the need for improvements across the software development lifecycle. The selection of the work chosen for discussion in this chapter is research that endeavours to provide different ways of dividing up software artefacts. A common theme of all the research discussed here is the need to separate different kinds of units of interest. This need is based on the desire to make software artefacts easier to understand, easier to extend, and easier to re-use.

Now that we have motivated the research described in this thesis, and examined other work in this field, we now take a closer, more detailed look at the subject-oriented design model (see Chapter 4: Composition of OO Designs: The Model” on page 64).

# Chapter 4: Composition of OO Designs: The Model

The root problem addressed in this thesis is the inherent structural mismatch between requirements specifications and object-oriented design specifications. “Chapter 2: Motivation” on page 11 describes and illustrates the negative impact of this structural mismatch - support for individual requirements is scattered across the design and support for multiple requirements is tangled in individual design units. This reduces comprehensibility and traceability, making designs difficult to develop, re-use and extend.

This chapter describes an approach to addressing the structural mismatch problem. The approach is based on providing a means of decomposing artefacts written in one paradigm so that they can structurally match those written in another. In order for there to be such a structural match, it must be possible to decompose object-oriented designs in a manner that aligns with the structure of requirements specifications. Requirements are generally described by feature and capability. So, this means that object-oriented designs must also decompose design models by feature and capability, thereby encapsulating and separating their designs. 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 requires corresponding composition support, as object-oriented designs still must be understood together as a complete design. The core of this thesis is the specification of how design models are composed. Composing design models involves:

1. Identification of Corresponding Elements: As described in “4.1. Decomposing Design Models” on page 65, decomposing design models based on the structure of requirements specifications may result in overlapping parts, where there are different views of those parts in different design models. In

order to successfully compose design models, those overlapping parts (called *corresponding elements*) must be identified.

2. Integration: Integration of design models involves synthesising a single composed design model from a collection of design models. Two kinds of integration are described in this thesis. “Chapter 6: Override Integration” on page 127 gives a detailed description of the semantics of overriding design subjects in the context of UML, and the impact of override on different kinds of design elements. “Chapter 7: Merge Integration” on page 155 provides the same detail for merging design subjects

This chapter describes the composition model with the following sections:

- *Decomposing Design Models:* This section describes the structural matching of design models with requirements specifications.
- *Composing Design Models:* This section gives an overview of the composition model; that is, input design models are integrated to an output design model. It introduces the notion of design models as *design subjects* and describes their structure from the perspective of composition.
- *Specifying Composition:* This section describes the means for specifying how design models should be composed. This is with a new kind of design relationship, called a *composition relationship*.
- *Analysis of the Output of a Composition:* This section analyses the output of a composition, and considers possible difficulties associated with it. Solutions to these difficulties are discussed.
- *Using Subject-Oriented Design:* This section discusses the phases of the development cycle when the approach described in this research is useful, and some implications of its usage.

## 4.1. Decomposing Design Models

For object-oriented design models, matching the structure of requirements means that design models must be decomposed – that is, divided up – into separate models that match that structure. These separate models 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 many and varied. They include units of requirements like features, and so-called *cross-cutting* requirements, (like persistence or distribution) that affect multiple units of functionality. Design subjects can also encapsulate units of change, making evolution of software additive rather than invasive.

Conceptually, a design subject can be written in any design language, but the focus of this thesis is the UML [UML 1999]. A UML design subject can conceptually contain any valid UML diagrams. Scoping for this work, however, involved selecting a subset of the full set of UML diagrams, and is detailed in “Scope of Work” on page 72. Application of this approach to other design languages, and to all UML diagrams remain interesting issues for future research.

Design subjects thus provide a means of decomposing systems that complements those already provided by the other UML diagrams. They permit the encapsulation of all, and only, those design elements that relate to a particular requirement. Whereas the design elements in a conventional UML design model must be defined completely with respect to the entire system, the design elements in a design subject need only contain those details that are relevant to the requirement it encapsulates.

**Structural Matching with Requirements**

The simplest model for structuring design subjects directly with requirements specifications is to have a one-to-one match of requirement with subject. The full requirements specification is the input to the decision-making associated with dividing up the design into design subjects. In “Chapter 2: Motivation” on page 11, a discussion of the requirements specification paradigm notes that there are numerous approaches to requirements gathering and specification based on the notions of features, capabilities, services, etc.

*One-to-One*

In many cases, a division into design subjects based directly on the particular units of division at the requirements specification level will yield a one-to-one match of requirement with subject.

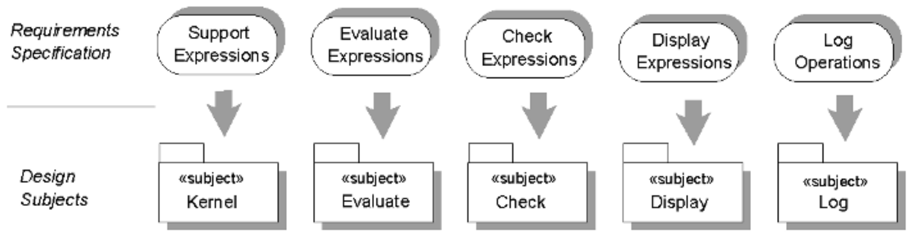


**Figure 14: Requirements and Subjects: One-to-One Structural Match**

For the small example motivating this work described in “Chapter 2: Motivation” on page 11, an analysis of the requirements specification (“Requirements Specification" on page 19) shows that capturing each feature of the SEE in a subject illustrates this simple model (see Figure 14).

Even a requirement which has an impact across all the other requirements, such as the subject "Log" (which logs operation execution across the full SEE), may be separated from those operations, and designed as a separate model. These kinds of requirements are considered to be *cross-cutting* requirements ([Kiczales et al. 1997], [Tarr et al. 1999]), and their separation from the design elements they cut across is a particularly useful capability of this model. This is because cross-cutting requirements are generally tangled up with the design for other requirements, thereby exacerbating difficulties with comprehension, etc. (see "Chapter 2: Motivation" on page 11).

The ability to structure design models in this way alleviates the scattering and tangling problems that motivate this work. Each design subject is easy to understand as it supports only one requirement, with every included design element providing for some need within the requirement, and no redundant design element that is not used for that requirement. Traceability is clear because of this one-to-one match. Any new requirement may also have its own design subject, making changes additive rather than invasive. Reuse of any particular design subject is not complicated by the existence of design elements within the subject that are not relevant.

From a UML perspective, the approach to capturing requirements as Use Cases is likely to yield a one-to-one match with design subjects [Jacobson et al. 1999]. Use cases support the separation of requirements specifications into the different uses of a computer system. This separation is not maintained through the analysis and design with UML, but with an approach such as this composition model, the decomposition of the design models could be based on the individual use cases in a one-to-one match.

*One-to-Many*

There may also be situations where the level of granularity of a particular requirement may yield a complex design subject, which, based on the intuition and experience of the designer, could be further divided up and captured as multiple design subjects. This has the advantage of simplifying the design of the individual design subjects, and also supports their concurrent development by different teams. For example, further analysis of the display requirement of the SEE might highlight the need to display expressions on different kinds of devices, and in different ways; 1) Display an expression as a string on a text window; 2) Display an expression as a tree structure on a graphical window; 3) Display an expression as a string and highlight different constructs in different colours, on a graphical window; etc. The original display requirement might thus be captured as multiple design subjects as illustrated in Figure 15.

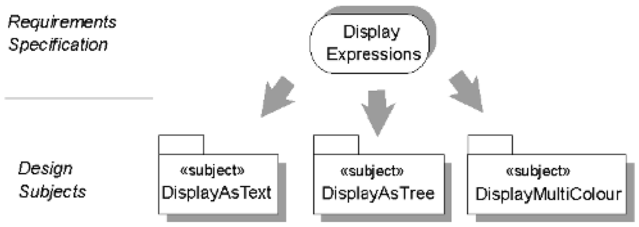


**Figure 15: Requirements and Subjects: One-to-Many Structural Match**

Another possibility of a one-to-many match of requirements with design subjects is where a significant change request may be received from an interested party. One approach to handling such a request, where the impact is significant, is to design the change as a separate design subject, and compose with the design subject to be changed[1]. While the change request may itself be viewed as a new requirement, and therefore the one-to-one structural match applies - on the other hand, from the original requirement's perspective, its correct design is now in multiple subjects.

One-to-many structural matches could be looked upon as having the negative *scattering* properties that result in difficulties associated with comprehensibility, traceability, evolvability and reuse as described in "Chapter 2: Motivation" on page 11. In both cases, a single requirement is scattered across multiple subjects. However, clear traceability to the original requirement still exists in this case. In addition, the rationale for further dividing the subjects is for reasons of decomposing complexity in the first case, and easing change of a subject by designing the change separately in the second case. Finally, since the model supports the composition of subjects, the "many" subjects in the one-to-many structural match may be composed to a single subject, thereby simulating a one-to-one structural match.

*Many-to-One*

It is also possible that multiple requirements may be supported by a single design subject. However, this occurs as a result of a composition, as the result of a composition is itself a design subject. In general, the output of composition of multiple subjects is expected to have scattering and tangling

1. See "Is every Requirement a Subject?" on page 70 for a discussion on whether every change is designed as a separate design subject.

properties as this is the motivation for decomposing design models in the first place. However, the case considered here is where a small, logical grouping of requirements may each have been designed as separate subjects, but, for convenience, composed into a design subject as a single unit. From the SEE example, this might occur where there are different kinds of checking requirements; 1) Check for syntax; 2) Check for conformance to organisation style; 3) Check variables defined are used, and variables used are defined. Each of these three requirements may be designed as separate subjects, which cleanly separates their designs, making them easier to understand. However, from a higher level perspective, they might collectively be considered as a single, check activity (and therefore as one requirement), and composed into a single subject to simplify the inclusion of checking into an expression environment. (See Figure 16).

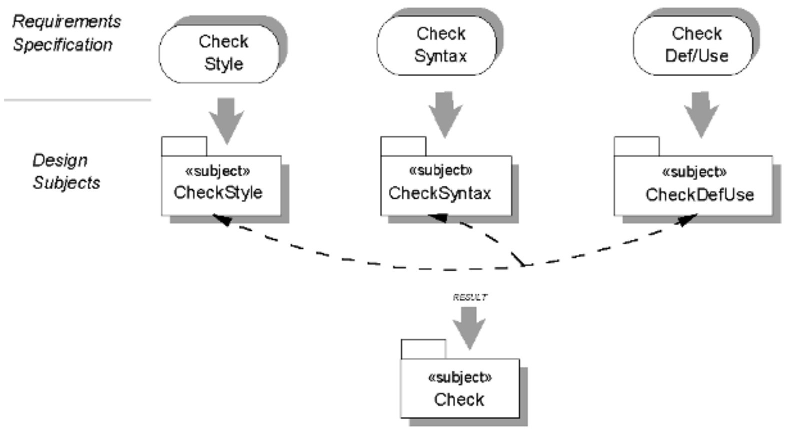


**Figure 16: Requirements and Subjects: Many-to-One Structural Match**

The `Check` subject in Figure 16 is the composition of the `CheckStyle`, `CheckSyntax` and `CheckDefUse` subjects that are a one-to-one match with the requirements for checking. As such, `Check` now contains the design for those three requirements. In one way, `Check` itself could be considered as "tangled" up with a number of different requirements. Tangling is a property previously identified as having a negative impact on comprehensibility, traceability, evolvability and reusability (see "Chapter 2: Motivation" on page 11). However, as regards comprehensibility, the separated checking subjects may still be reasoned about separately. Traceability to the requirements remains clear. Any changes to the existing check requirements, or any new check requirements may still be designed separately and composed where required. Finally, each individual check subject may still be reused separately from the others, and composed separately, where required.

*Many-to-Many*

The final general cardinality possibility considered here is whether many-to-many requirements to design subjects are acceptable. Though the model does not currently explicitly enforce rules to ensure that this situation is avoided, it is *not recommended*. In general, such a case is exactly the kind of situation that the subject-oriented design model is explicitly designed to avoid. The scattering and tangling properties associated with a structural mismatch between requirements and design models feature highly here, and therefore, such a design will exhibit the same difficulties as those described in "Chapter 2: Motivation" on page 11, that are the central motivations for this work.

*Is every Requirement a Subject?*

System change requests received from test teams (or any interested party) may be considered as requirements on the system. Here, where the change request is a significant size, it may often be prudent to design the change as a separate subject, thereby making it more easy to understand and work with, and avoiding the need for invasive change of an existing subject. However, in practical terms, not all change requests might warrant a new design subject. Where the change is small and invasively changing an existing subject is not an issue, it may be more practical to simply change the subject directly.

The trade-off to be made when making such a decision is to balance the perceived need for keeping separate all changes to subjects during the testing phase, against the possible cost of managing all the separate subjects. Keeping all changes separate has the advantage of providing an auditable, historical record of change during testing - quality assurance professionals like this level of auditability for their records and for general accounting purposes for feeding into the next planning phase [IBMa 2000], [IBMb 2000].

Where there is good development environment support, a development team may be able to easily manage multiple separate design subjects. In this case, a need to keep all changes separate may be easily supported. However, where the environment support is insufficient, a balance may need to be considered as to how important it is to keep changes designed separately, versus how difficult it is to manage separate subjects. This will tend to influence the decision of whether to design a particular change request as a new design subject, or just change the subject directly.

**Overlapping Subjects**

It is possible – indeed, expected – that some of the same concepts may be relevant to multiple design subjects. For example, an educational system that contains requirements for teachers and for students both consider the concept of Person (assuming, in this case, that teaching is performed by people).

Thus, if different requirements were each modelled as separate design subjects, they would both include their own views of people. These views may, or may not be identical; for example, one subject might attempt to generalise its perception of basic properties of people, and specialise for its requirement, whereas the other, in a similar attempt at good software engineering, may use delegation for separation of different kinds of properties, and may also have a different view of what the basic properties of people are.

Design subjects may therefore *overlap*, and may include some differences in their views of overlapping parts. This is the strength of design subjects – they permit each of the different parts of a system under design to model the same concepts in whatever way is most appropriate to support that subject's requirement. This ability provides considerable decomposition and encapsulation power. Differences in views can be identified and resolved during composition, as part of the design process. With UML, design elements that support the same concept, but have different views that necessitate different specifications, must be specified separately. And, since there is no means of synthesising a complete design from incomplete pieces in UML, such elements will remain separate throughout the design cycle.

## 4.2. 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 semantics of the design and the impact of composition on the full design. This section discusses the policies employed in this research for composing design subjects and includes:

- *What does a Subject look like?:* Here, the scope of this work is defined, and how the design elements within a subject are viewed by is composition discussed.
- *Composing Design Subjects:* Here, there is a general discussion on what composition is - i.e. the synthesis of input design subjects to an output design subject.
- *Deferring Subject Composition:* Though not the main focus of this research, this section describes how design subjects need not be composed at the design level. With supporting programming models, the decomposi-

tion into design subjects may be maintained in the code, with composition deferred to the code phase.

The description of how to specify composition within the subject-oriented design model then follows in “4.3. Specifying Composition” on page 78.

**What does a Subject look like?**

A design subject is similar to a UML package in that it is a grouping mechanism for model elements. A design subject is represented as a special type of UML package, stereotyped as «`subject`». The difference between a subject and a package is that there is a restriction on the kinds of model elements that a subject may group. This restriction is for the purposes of providing a manageable boundary for this work.

*Scope of Work*

The UML semantics guide states that “A Package may only own or reference Packages, Classifiers, Associations, Generalizations, Dependencies, Constraints, Collaborations, StateMachines, and Stereotypes”. For the purposes of this thesis, we further restrict a subject to a subset of these elements by stating that

*“A Subject may only own or reference Subjects, Classifiers, Associations, Generalizations, Dependencies, Constraints, and Collaborations”*

The restriction does not imply that the composition concept is only appropriate for a subject that owns or references only these model elements. The extent to which “subject” and “package” should be considered synonyms must be investigated, and therefore, the impact of composition on all the model elements that are owned or referenced by packages needs to be considered. This is an important area for future research.

*Tree Structure*

While a subject looks like standard UML design models to the designer, from the perspective of composition, a subject looks like a tree structure. The consideration of a subject as a tree structure for the purposes of composition provides a convenient mechanism for assigning rules to its specification. Composition is specified with composition relationships between design elements. Representing design elements as a tree structure supports the definition of rules relating to scope, precedence and general validity of those composition relationships. See “4.3. Specifying Composition” on page 78 for more details.

This representation of subjects is based on the observation that each UML design element has properties and may (or may not) contain other design elements in a standard tree type of structure. For example, in Figure 17, the tree-like structure of the design elements within the scope of this work is illustrated.

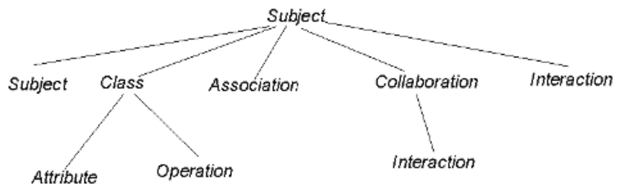


**Figure 17: A Subject as a Tree Structure**

*Composable Elements*

The first observation to be made from Figure 17 is that not all of the UML constructs supported within the scope of this work have been included. The design elements illustrated are those elements which may directly participate in composition relationships, and are therefore considered to be "composable elements". While there are many other design elements within the scope of this work (for example, generalizations, dependencies, parameters, etc.), and which therefore may be impacted by composition, these are the only elements which may be directly related by a composition relationship. The exclusion of other design elements from the set of composable elements is based on two criteria:

- Some elements within the scope of this work logically belong to another element which is itself a composable element. For example, parameters are part of operations. The full signature of operations includes the properties defined by the UML for the Operation metaclass, but also, the set of parameters which are connected to an operation. The semantics of composition in relation to operations is based on this full signature. Therefore, Parameters are excluded as elements which may directly participate in a composition relationship independently of the operation to which they belong. Another example of such an element is AssociationEnd - a UML metaclass which defines the connection of an association to a classifier. These are also considered to be part of the full specification of Associations, and are therefore excluded as elements which may directly participate in composition relationships.
- Other kinds of design elements are broadly considered to be "constraints" on particular composable elements within design subjects, and so, they are also appropriately considered to be part of the full specification of the element(s) to which they are attached - for example, instances of the Con-

straint, Dependency and Generalization metaclasses. These elements are therefore excluded from participating directly in composition relationships.

*Primitive vs. Composite*

A further observation may be made from Figure 17. Some elements are nodes which are composed of other elements further down the tree (e.g. Subject, Class), while other elements are leaves (e.g. Attribute, Operation). The elements which are composed of other elements are called *composites*. The elements which are leaves are called *primitives*. Whether an element is a primitive or a composite has an impact on the semantics of composition, described in this section.

The selection of the design elements that are considered to be composites or primitives is not directly obvious from the UML metamodel. Just considering the UML metamodel directly, for example, we might consider operations to be composites as they contain parameters. However, the distinction between the two is not based on the definitions within the metamodel, but instead based on the semantics of composition.

> Primitive elements are those design elements that are considered in their entirety for the purposes of composition - that is, *all* properties of primitive elements are considered together when establishing correspondences between them, and when integrating them.

Revisiting the example of operations, operations contain parameters, but the full signature of an operation is integrated with the full signature of other operations. 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)
```

With a composition relationship with override integration specified[2], this results in a composition of the two operations. Overriding the first `op1` specification with the second results in an operation specified as public, with three parameters as defined by the second specification. This example illustrates how the full specification of an operation is overridden, and so, in this sense, operations are *primitives.*[3]

2. Where a composition relationship with override integration is specified between two design elements, this means that the specification of one of the design elements is replaced by the specification of the other design element
3. See "Incompatible Elements" on page 100 for a general discussion on composing elements with potentially incompatible properties.

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. Except for attributes, each of these elements, from the perspective of the UML metamodel, appears to be a container for other constructs - operations own parameters, associations own association ends, and interactions own messages. However, from the perspective of composition, they are considered in their entirety as their components are not sensibly designed or reasoned about separately (for example, what's an association end without its association?).

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, as primitives, are examined individually for composition. 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.

For the purposes of composition, three types of elements are recognised as composite - Subject, Classifier and Collaboration. Each of these contain elements that have been identified as primitive composable elements, and therefore, during composition, these elements are considered separately. From the perspective of composition, composites may also contain other composites. An example within the current scope of this work is a subject which may contain other subjects, classifiers or interactions.

## Composing Design Subjects

The model for composing design models is, essentially, the synthesis of multiple (two or more) input design subjects to an output design subject. Each input design subject is an independent tree structure in its own namespace, as defined by the UML. The input subjects are integrated as defined by a (set of) composition relationships[4], and the result is a new, independent tree structure in its own namespace (see Figure 18).

4. In each of the examples in this thesis, a composition relationship is represented as a dotted arc between the elements to be composed. The arrowheads at the ends of the arc have meaning in terms of specifying the integration strategy, and are further explained in "Integration of Inputs" on page 87.

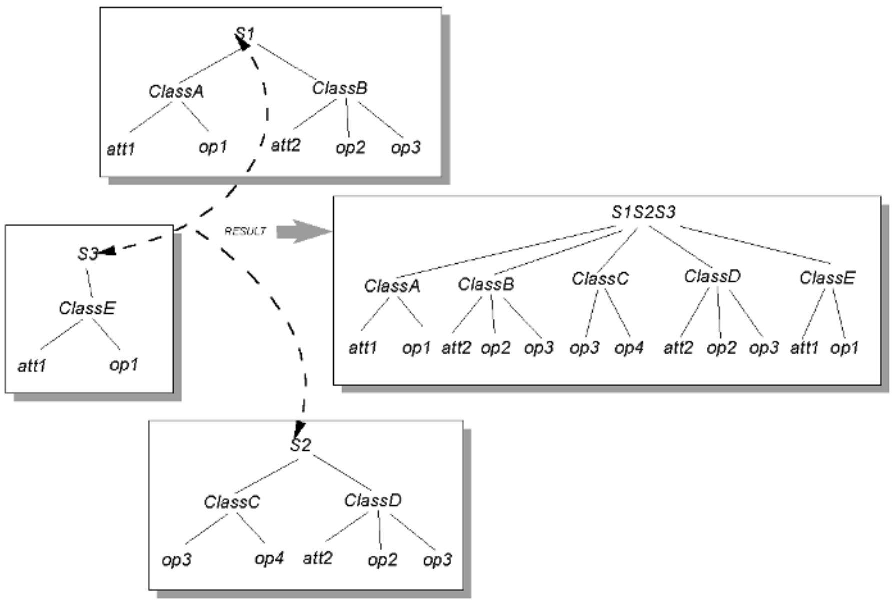


**Figure 18: Composing Design Subjects to New Result**

*Why compose into new model?*

An alternative to composing design subjects into a new "result" design subject might be to make the appropriate changes to an existing subject. This question applies to composition with override integration particularly. Override integration means that design elements in a particular design subject are replaced by design elements in another subject. Here, it is not immediately clear whether it would be better to make the replacements in the existing design subject - that is, change the particular tree structure of the subject being overridden, or copy elements to a new subject as appropriate. For the following reasons, the result of composition of design subjects is a new design subject.

- *Consistency*: While it is not immediately obvious which approach to take for override integration, composing subjects to a new subject is the appropriate course of action for merge integration. Since the semantics of merge is essentially the amalgamation of design subjects, it is appropriate that the result is a new subject. For consistency purposes, a single composition strategy is used. This means that composition with override integration also composes to a new subject.
- *Comprehensibility*: One of the difficulties with conventional object-oriented design is the difficulty in understanding its semantics. This is because of the scattering of the support of a single requirement across the full design, and because of the tangling of the support for multiple

requirements in a single design element. Maintaining the separate design subjects while composing to a new resulting subject supports comprehensibility, as the full design may be understood by understanding the component subjects.

- *Version control*: Maintaining the histories of versions is an important part of software engineering. The histories of decisions, and the clear representations of previous approaches are valuable information for the maintenance and evolution of software. Composing subjects to new subjects, thereby maintaining the separate component subjects supports clean version control. Maintenance of multiple copies is not an issue, as, with this approach, changes to the design are themselves encapsulated in a separate design subject, to be composed where appropriate.

*Composing Overlapping Subjects*

As described in “Overlapping Subjects” on page 70, some of the same concepts may be relevant for multiple subjects, and therefore each subject may contain a specification of that concept from the perspective of the particular subject.

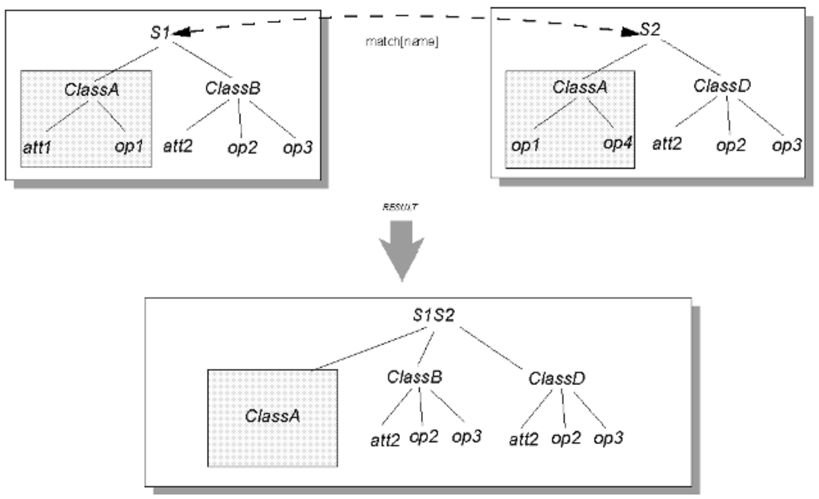


**Figure 19: Composing Design Subjects with Overlap**

The areas of overlap in the input subjects to a composition are identified as *corresponding elements* during composition specification (see “Identifying Corresponding Elements” on page 80). As illustrated in Figure 19, corresponding elements are synthesised in the resulting design subject. The exact nature of this integration depends on the integration strategy defined within the composition specification (see “Integration of Inputs” on page 87).

**Deferring Subject Composition**

Enhancing existing UML decomposition capabilities by adding an ability to decompose based on the structure of requirements specifications provides many benefits relating to comprehensibility and traceability. With supporting capabilities in the programming domain, this separation can be maintained throughout the lifecycle. Such support is available, for example, in Hyper/J™ for Java, from IBM Research [Tarr & Ossher 2000]. With a programming model like that provided in Hyper/J, the decomposition into design subjects described here can be maintained to the code phase. In Hyper/J, composition of the resulting code subjects is specified with *composition rules*, which identify corresponding code elements and specifies how the programs should be integrated. The composition relationship specification for composition of design subjects has been influenced by composition rules from this programming model. Automated generation of the composition rules that are used for composing programs in Hyper/J, from the composition relationships for design subjects described here, remains an important area for future work. Such automated generation is likely to be relatively straightforward, because the concepts are similar. A complete assessment of where the differences lie is required, and is added to future work.

Another programming model that provides similar levels of separation at the code level is the aspect-oriented programming model, as implemented by AspectJ™ [Kiczales & Lopes 1999]. This model has differences with the decomposition/composition approach taken here for design models - most particularly in the notion of a "base" program to which all "aspects" are applied. However, at a conceptual level, the goals of the aspect and subject approaches are similar, in that separation of different kinds of requirements is supported. An interesting area of future research is to assess the applicability of the subject-oriented design model described here as the design approach for aspect-oriented programming.

It is important, however, for the work described in this thesis to define the semantics of composition relationships by describing their impact on the design subjects. Once the semantics of composition relationships at the design level are well-defined, generation of composition rules at the code phase should be straightforward.

## 4.3. Specifying Composition

Composition of design models is specified with a *composition relationship* between the design models to be composed. This compares with the specification of *joining* packages in Catalysis (specified using a stereotyped dependency relationship [D'Souza & Wills 1998]), and also compares with the *synthesis* operation from OORam, which identifies role models to be synthesised [Reenskaug et al. 1995].

A composition relationship identifies corresponding design elements in the related models, and specifies how the models are to be integrated; i.e. the composition's integration strategy. Different kinds of integration strategies may be attached to a composition relationship - for example, the models should be merged, or one model should override another. These two integration possibilities are defined in detail in this thesis.

In this section there is:

- a description of how inputs to a composition are specified,
- a discussion on how corresponding elements are specified,
- a description of rules governing a composition relationship's scope, and
- a description of the kinds of integration currently supported within the model.

## Specifying Inputs

The intellectual exercise of choosing the particular design subjects to be composed depends on the needs defined by the development effort under way. Depending on the original decomposition into design subjects, the selection of the design subjects for a particular manifestation of a combination of requirements may be varied, and is based on the needs of various players within the development process - for example: integration testers may experiment with the composition of multiple different combinations of subjects; system testers may experiment with different combinations again; while acceptance testing by different users may require the composition of a set of subjects to fulfill the business needs of those users. This research does not provide a process for aiding this intellectual exercise, but the possibility of providing rules and guidelines for such selection remains an interesting area for further research. Instead, this section considers the technical aspect of how to specify inputs to a composition with composition relationships.

Composition relationships are defined between composable design elements. The elements that are related by a composition relationship are the inputs to that composition specification. As discussed previously, composition entails synthesising two or more input subjects into an output subject. Therefore, identifying inputs to a composition must first involve identifying the input subjects, and specifying a composition relationship between those subjects. Figure 20 illustrates a composition relationship between subjects - that is, from the perspective of composition, between the roots of the trees to be composed. The composition relationship between the roots of the trees to be composed provides the context for the composition to a single output subject. This relationship is required because it specifies context in the sense that it provides a namespace within which rules associated with naming, element referencing and integration of composed elements in the output subject occurs (see “4.4. Analysis of the Output of a Composition” on page 95 for more details).

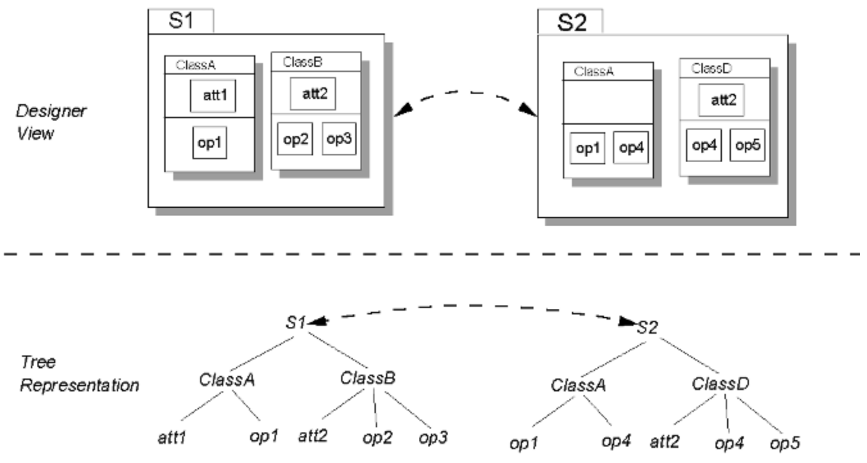


**Figure 20: Subject-Level Inputs to Composition**

Once the context for composition is set with a composition relationship between input subjects, further composition relationships may be defined between composable elements at lower levels of the tree. These define exceptions to the general composition specification defined at subject level. There are many examples of this in forthcoming sections. See “Scope of Composition Relationship” on page 83 for rules associated with specifying composition relationships at levels of the tree lower than the subject level.

### Identifying Corresponding Elements

Elements in different subjects which provide a design for the same concept are said to correspond. Though the elements in the different subjects may provide different views or specifications for a concept, they nonetheless represent the same fundamental concept in the domain. These are the overlaps which were discussed in “Overlapping Subjects” on page 70. Therefore, the semantics of any integration strategy must recognise this correspondence, and act accordingly. For example, an override integration strategy specifies that an element is overridden by its corresponding element in another subject, and elements without corresponding elements are not overridden. Composition relationships specify corresponding elements either explicitly, or implicitly with matching criteria.

*Explicit*

Inputs to a composition relationship explicitly define that those inputs are corresponding. Since composition relationships may be defined between composable elements which are both primitive elements and composite elements, then the correspondence of elements may be explicitly defined between primitive elements and between composite elements.

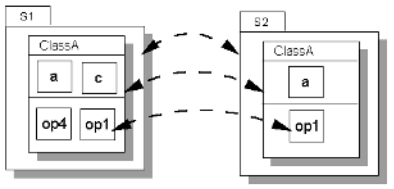


**Figure 21: Explicit Correspondence**

In Figure 21, the following elements are corresponding:

- Subject `S1` corresponds with subject `S2`
- Class `S1.ClassA` corresponds with class `S2.ClassA`
- Operation `S1.ClassA.op1` corresponds with operation `S2.ClassA.op1`.

*Implicit*

*Implicit* specification of correspondence is achieved in a general way that applies to all elements owned by the elements related by the composition relationship - that is, all elements lower in the tree structure than the elements between which the composition relationship has been defined. The general rule is a matching specification attached to the composition relationship and is based on matching the values of properties of design elements of the same type. For example, a match specification may state that a match on the values of the `name` properties of the related elements implies correspondence. In theory, the values of all properties (as described in the UML specification [UML 1999]) associated with the type of the design element may be used for matching criteria. However, within the scope of the research described in this thesis, matching is on name only. Based on the general matching specification, each of the elements within the scope of the composition relationship are compared in order to establish whether they are corresponding (see "Scope of Composition Relationship" on page 83 for more details on the scope of composition relationships). Figure 22 illustrates how multiple composition relationships as illustrated in Figure 21 can be avoided through the use of general matching criteria.

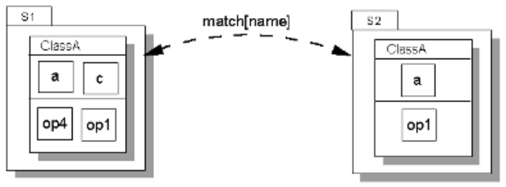


**Figure 22: Implicit Correspondence**

In Figure 22the following elements are corresponding:

- [Eg4.3.1] Subject `S1` corresponds with subject `S2`
- [Eg4.3.2] Class `S1.ClassA` corresponds with class `S2.ClassA` (from matching criterion specified in [Eg4.3.1])
- [Eg4.3.3] Attribute `S1.ClassA.a` corresponds with attribute `S2.ClassA.a` (from matching criterion specified in [Eg4.3.1])
- [Eg4.3.4] Operation `S1.ClassA.op1` corresponds with operation `S2.ClassA.op1` (from matching criterion specified in [Eg4.3.1])

*Exceptions*

A composition relationship with `match[name]` implicit correspondence specifies that identification of corresponding elements is on the values of the `name` property. All components of composites are subject to this check for correspondence. However, in some cases, there may be exceptions, where elements of the same name are not intended to correspond. Composition relationships between the exceptions with a `dontMatch` specification specify that those elements do not correspond. This specification takes precedence over any relationships between their owners (see "Scope of Composition Relationship" on page 83 for more details).

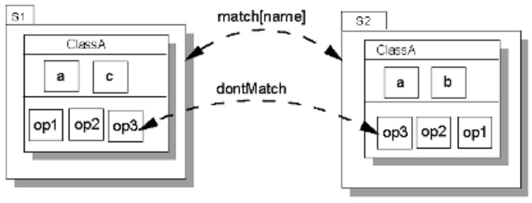


**Figure 23: DontMatch Correspondence**

In Figure 23 the following elements are corresponding:

- [Eg4.3.5] Subject `S1` corresponds with subject `S2`
- [Eg4.3.6] Class `S1.ClassA` corresponds with class `S2.ClassA` (from matching criteria specified in [Eg4.3.6])
- [Eg4.3.7] Attribute `S1.ClassA.a` corresponds with attribute `S2.ClassA.a` (from matching criteria specified in [Eg4.3.6])
- [Eg4.3.8] Operation `S1.ClassA.op1` corresponds with operation `S2.ClassA.op1` (from matching criteria specified in [Eg4.3.6])
- [Eg4.3.9] Operation `S1.ClassA.op2` corresponds with operation `S2.ClassA.op2` (from matching criteria specified in [Eg4.3.6])

Operation `S1.ClassA.op3` does not correspond with `S1.ClassA.op3` because of the relationship with `dontMatch` specified, and therefore they are treated as separate elements in the integration process.

## Scope of Composition Relationship

The specification of composition in a composition relationship between two composable elements applies to all elements at levels of the subject tree lower than the elements related by the relationship, *except* for those elements where additional relationships are defined. The lower levels to which composition relationship specification applies are called the *scope* of the composition relationship. For example, in Figure 20 on page 80, all design elements in the tree are within the scope of the relationship between subject `S1` and subject `S2`.

Within the main context of the composition (that is, the composition relationship between the input subjects - see “Specifying Inputs” on page 79), additional composition relationships may be defined between elements at a lower level of the tree - subject to certain rules[5]. For example, in Figure 24, additional composition relationships are specified at levels of the tree lower than the relationship between the subjects.

5. See “Rules for Specifying a Composition Relationship” on page 84 for a discussion on some rules.

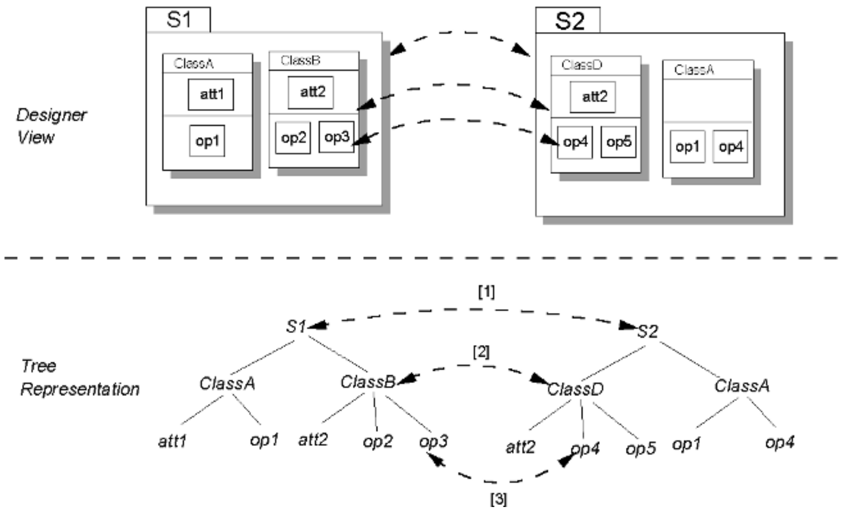


**Figure 24: Multiple Composition Relationships**

*Precedence*

Composition relationships between elements take precedence over relationships at a higher level in the tree. For example, looking at the tree representation in Figure 24, the elements `S1.ClassB.op3` and `S2.ClassD.op4` are composed based on the specification of the composition relationship marked `[3]`. That is, regardless of the integration specification specified in relationships `[1]` or `[2]`, these elements are composed only based on what is specified in `[3]`. Similarly for the composition of `S1.ClassB` and `S2.ClassD`. These are composed based on the specification in the relationship marked `[2]`, though `S1.ClassB.op3` and `S2.ClassD.op4` are excluded because of their participation in another relationship.

## Rules for Specifying a Composition Relationship

As with any design construct, rules are defined to ensure the validity of composition relationships. This section only addresses general rules for composition relationship well-formedness that serve to describe the subject-oriented design model. For the full list of well-formedness rules for composition relationships in the context of the UML see "Chapter 5: Composition Relationship: An extension to the UML Metamodel" on page 109, and for rules directly related to integration strategies, see "Chapter 6: Override Integration" on page 127, and "Chapter 7: Merge Integration" on page 155.

*Inputs are the Same Type*

[Rule 1] Composition relationships may only be specified between elements of the same type.

In the subject-oriented design model, inputs to a particular composition relationship must be the same type. Some work in the database field where integration of schemas is the focus (for example [Batini et al. 1986]), defines equivalences between different constructs in the database model. One example is where a concept might be modelled as an attribute in one schema, and as a separate entity with a relationship in another. The application of this level of flexibility to integration within the object-oriented modelling paradigm remains an interesting area for further research. The restriction defined here is in keeping with similar restrictions defined for integration in the composition of multi-dimensional concerns implemented in Hyper/J, [Tarr & Ossher 2000] and also the aspect-oriented programming model implemented in AspectJ [Kiczales & Lopes 1999]. These are the most likely candidates for direct programming support for the subject-oriented design model (see “Deferring Subject Composition” on page 78), and so this restriction is currently not seen as an issue.

*Context for Composition must be Specified*

> [Rule 2] A composition relationship must be specified between input subjects to define the context for composition of inputs to an output subject, and a context for composition relationships at lower levels of the subject tree.

[Rule 2] has been previously discussed in “Specifying Inputs” on page 79.

*Inputs to a composition relationship at lower level to their corresponding parents*

> [Rule 3] Composition relationships may only be specified between elements whose parents are corresponding, and therefore are integrated in the result.

Composition of elements to a result requires a context and namespace within which to place the composed element. Recall the tree structure of a design subject described in “Tree Structure” on page 72, which illustrated composable elements as either composites or primitives, where composites contain primitives, and some may also contain other composites. In order to maintain this tree structure in the output design subject, each composed element must be placed in an appropriate node of the tree. As illustrated in Figure 25, a composition relationship between elements where the parents are not composed leads to an unscoped namespace within which to place a result of such a composition.

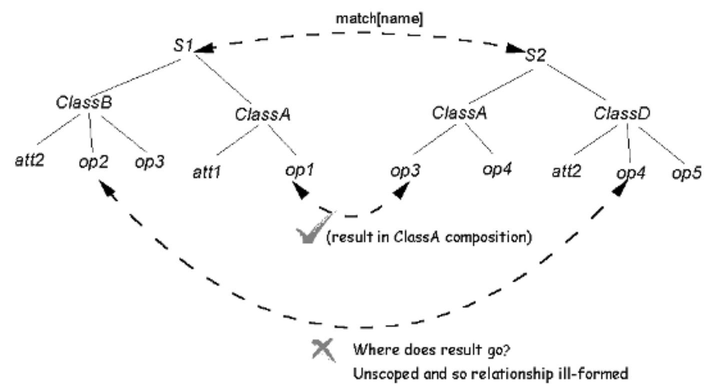


**Figure 25: Composition Relationships and Corresponding Parents**

An implication of this rule is that all inputs to a composition relationship will be at lower levels of the tree to their corresponding parents. For example, in Figure 26 the composition relationship marked `[1]` has been defined as the relationship setting the context for the composition to an output (see [Rule 2]). A further composition relationship between `S1.S3` and `S2` violates [Rule 3], when defined within the context of the relationship marked `[1]`. The rule avoids difficulties which this case may have presented. The namespace within which to place the composed elements of `S1.S3` and `S2`, in the context of a composition between `S1` and `S2`, is ambiguous. Of course, in a different context, as a composition to a different output, a relationship between `S3` and `S2` may be valid. The composition relationship between `S1.S4` and `S2.S5` (marked `[3]`) does not present the same difficulty, as the namespace for the result is in the context of the output of the composition of `S1` and `S2`.

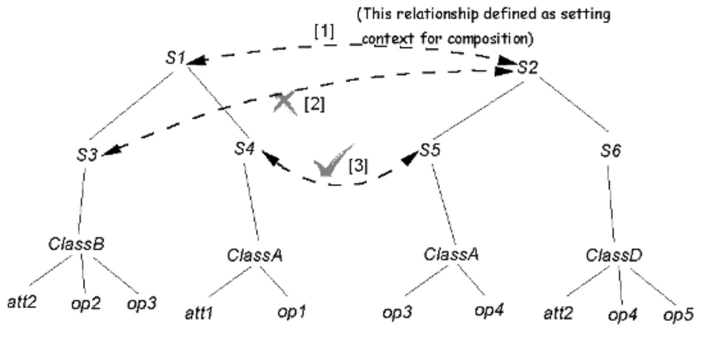


**Figure 26: Composition Relationships at the Same Level in Subject Tree**

*Participation in multiple composition relationships*

There is no restriction to the number of relationships in which a design element may participate. While individual integration strategies may extend restrictions in this area, the general composition model allows an element to participate in multiple composition relationships, subject to [Rule 3]. For example, the set of composition relationships depicted in Figure 27 are possible.

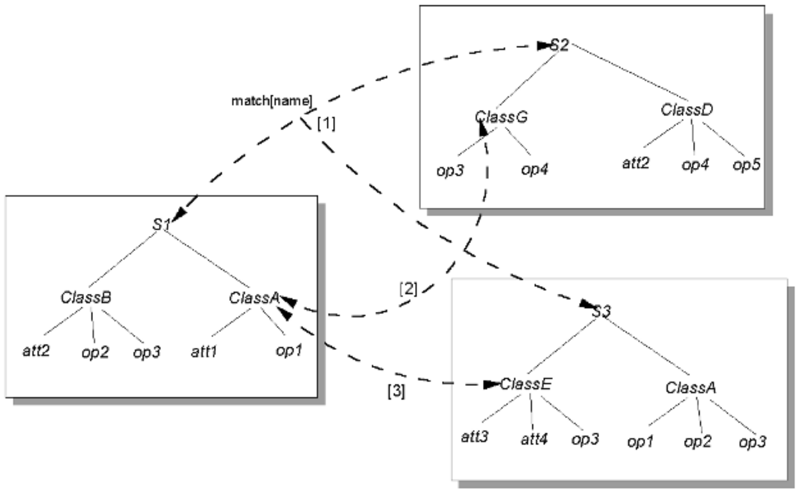


**Figure 27: Participation in Multiple Composition Relationships**

Here, within the context of the composition relationship between subjects `S1`, `S2` and `S3` (marked as `[1]`), `S1.ClassA` participates in two different explicit composition relationships (`[2]` with `S2.ClassG` and `[3]` with `S3.ClassE`). From these two relationships, there will be two result classes within the composed `S1`, `S2` and `S3` which contains an integration involving `S1.ClassA`. `S1.ClassA` is also composed with `S3.ClassA`, as they correspond because of the contextual relationship (marked as `[1]`) which specifies matching by name for identifying corresponding elements.

This example also illustrates that composition relationships at lower levels do not have to have the same number of inputs as composition relationships at higher levels in the tree. This increases the flexibility of the kinds of compositions possible within the context of the composition of one output subject.

## Integration of Inputs

The integration of input subjects to produce an output subject is at the core of composition of design models. The semantics of the integration strategy must detail how corresponding elements are represented in the output subject (that is, the overlapping elements), and how elements with no corresponding elements are catered for in the output subjects. Design elements may be integrated in many different ways, and it is not the intention of the subject-oriented design model to restrict the kinds of integration which can be done. In this thesis, two particular kinds are described - *override* and *merge*. However, since it is not possible to anticipate all the different kinds of integration that might be needed, it is the policy of subject-oriented design to make the integration semantics as extensible as possible. This is done by abstracting the integration specification part of composition relationships at the meta-level, thereby allowing it to be extended as required (see “Chapter 5: Composition Relationship: An extension to the UML Metamodel” on page 109 for more details).

### Override Integration

An existing design subject is changed by creating a new design subject that contains the design of new behaviour, and *overriding* the existing design subject with this new design subject. Overriding an existing design subject is specified with *composition relationships* with *override integration,* specified between the existing design subject and a new design subject.

Overriding design elements is also possible in the overall context of multiple subjects being merged. This is analogous to design elements at lower levels of the subject tree being composed in a certain way, as specified by a composition relationship which takes precedence of a composition relationship at a higher level of the subject tree.

Composition relationships with override integration specify which design elements in the existing design subject are to be overridden by design elements in the new design subject. Any design elements in the existing design subject that are not overridden by design elements in the new design subject are added to the result unchanged. Any design elements in the new design subject that do not override design elements in the existing design subject are added to the result.

As with all kinds of integration, the overridden design subject itself remains unchanged, as the result of integration is to a new output subject (see discussion in “Composing Design Subjects” on page 75). Therefore, integration does not impact any composition specifications in which the overridden subject has previously participated. If it is appropriate for the output of the override integration to participate in any such compositions, then the output subject must be explicitly included in those compositions.

#### *Specifying Override Integration*

Override integration is specified by first selecting the inputs to the composition relationship; the design element to be overridden, and the design element containing the overriding specification. Override integration as part of a composition relationship is represented by a single arrowhead at only one end of the dashed arc, which indicates the element to be overridden. In general, the scoping and rules associated with composition relationships apply when override integration is specified, with two exceptions:

> [Override Rule 1] A composition relationship with override integration may only be specified between *two* composable elements. That is, one composable element is overridden by one other composable element.

Override integration changes the specification of an element to be overridden. This rule is included because without it (that is to allow an element to be overridden multiple times by different elements) there may be unanticipated results. Without explicit ordering of the different integrations, it is not possible to predict the final specification of the overridden element. General ordering of multiple compositions is currently not supported in the subject-oriented design model, but remains an interesting area for future research.

> [Override Rule 2] Within the context of a single composition, a composable element may only participate in one composition relationship as an overridden element.

This rule is an extension to [Override Rule 1], as the same argument applies in the context of a single composition of multiple input subjects to a single output subject.

*General Semantics*

This section illustrates the general semantics of override integration. For a more complete discussion on the impact of override integration on all elements currently supported by the subject-oriented design model, see "Chapter 6: Override Integration" on page 127.

[1] For each component in the overridden composite element, the existence of a corresponding element in the overriding composite element results in the specification of that element to be changed to that of the corresponding element. From Figure 28, the following overrides occur:

- The specification of class `S2.ClassA` is changed to the specification of `S1.ClassA` as a result of override
- The specification of attribute `S2.ClassA.a` is changed to the specification of `S1.ClassA.a` as a result of override
- The specification of operation `S2.ClassA.op1` is changed to the specification of `S1.ClassA.op1`, as a result of override.
- The specification of operation `S2.ClassA.op2` is changed to the specification of `S1.ClassA.op2`, as a result of override (recall that elements may participate in multiple compositions from "Participation in multiple composition relationships" on page 86)

- The specification of operation `S2.ClassA.op3` is changed to the specification of `S1.ClassA.op2`, as a result of override.

Note that there are two specifications of `S1.ClassA.op2` since it was involved in two override compositions. One of these is renamed to avoid a nameclash (arbitrarily chosen). While this conforms to the general composition model, in this case there is some scope for optimisation in future work.

[2] Elements in an overridden composite that are not involved in a correspondence match are unchanged. For example, from Figure 28, the attribute `S2.ClassA.c` has no corresponding elements, and so is added to the result unchanged.

[3] Elements that are components of an overriding composite and are not involved in a correspondence match are added to the result. For example, from Figure 28, the attribute `S1.ClassA.b` has no corresponding elements in `S2`. Since it is a component of an overriding class named `ClassA`, it is added to the specification of `ClassA` as a result of override.

[4] Changes to an overridden subject, either as a result of overriding of corresponding elements, or as a result of adding elements directly to the overridden subject, may not result in name clashes. In the event of name clashes, renaming of clashing elements occurs. For example, from Figure 28, overriding operation `S2.ClassA.op3` with `S1.ClassA.op2` results in a name clash with an already existing operation `S2.ClassA.op2.` To avoid this, the name of the overridden operation is changed.

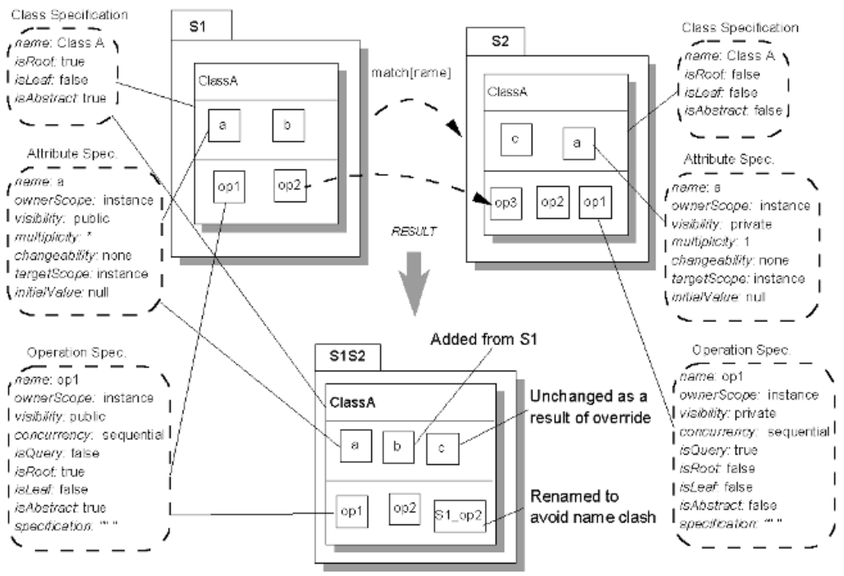


**Figure 28: General Override Semantics**

[5] The composed subject must conform to the well-formedness rules of the UML. Composition of input subjects to an output subject has the potential to result in problems in the output subject. These problems are discussed in “4.4. Analysis of the Output of a Composition” on page 95. Suffice it to say here that where it is appropriate, the subject-oriented design model will assist in avoiding some kinds of problems. In other cases, it is the responsibility of the designer to work to ensure the well-formedness of the output of the composition.

## Merge Integration

Design subjects are merged by specifying composition relationships with merge integration between the subjects to be merged. Composition relationships identify the subjects to be merged, and the design elements within those subjects that specify the same concept (i.e. correspond to each other) and should be considered as one. 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 need to be resolved, composition relationships with merge integration specify guidelines for the reconciliation.

With merged operations, the receipt of a message that may have activated one of the operations in an input subject, now results in the execution of all of the merged operations. Interactions may be attached to a composition relationship with merge integration to determine the order of execution. In general, composition relationships with merge integration conform to the scoping and general rules of composition relationships.

### *Specifying Merge Integration*

Specifying composition relationships with merge integration involves:

- Specifying the input elements to be merged within the context of an overall composition. This context does not have to be a composition relationship with merge integration specified. However, the elements at lower levels of the tree to a composition relationship with merge integration are subject to this integration unless further relationships are defined.
- For elements within the scope of merge integration that are not operations, reconciliation strategies should be attached to the relationship to handle possible conflicts. Reconciliation of conflicting elements is introduced in this section, but for a more detailed discussion, see “Chapter 7: Merge Integration” on page 155.
- For operation elements within the scope of merge integration where the order of their execution is important, an interaction specifying this order

should be attached to the composition relationship. Where the order of execution is not important, and an interaction is not attached to the composition relationship, merge integration generates interactions with the specification that each of the operations is executed. Specifying interactions for ordering of corresponding operations execution is introduced in this section, but for a more detailed discussion, see “Chapter 7: Merge Integration” on page 155.

- Patterns of merge integration may be identified and defined. Some kinds of requirements may have the same impact on multiple classes in a design model. For example, the logging of operations requirement in the SEE example impacts all operations in a model. This pattern of interaction between logging and operations requiring logging may be identified and designed separately for composition where required. See “Chapter 8: Composition Patterns” on page 198 for details on composition patterns.

*Specifying Reconciliation for Conflicts*

When subjects are merged, elements that are specified to support corresponding concepts are identified, and will be merged in the composed subject – that is, for most kinds of elements (except, for example, operations), they will appear once in the merged subject. However, since corresponding elements may have been specified separately, there may be differences in those specifications. There is considerable discussion in [Nuseibeh 1994] as to the nature of conflict between views, with a discussion based around differences in terms of inconsistencies, conflicts, contradictions and mistakes. For the purposes of this work, a conflict is defined as follows:

> If the values of any of the properties of corresponding design elements are different, then these design elements conflict.

Differences between elements must be reconciled for the composed subject. One approach to reconciling conflict is to assign precedence to one of the subjects involved in the merge. When a conflict occurs, the specification of the element in the subject with precedence is deemed to be the specification for the merged element.

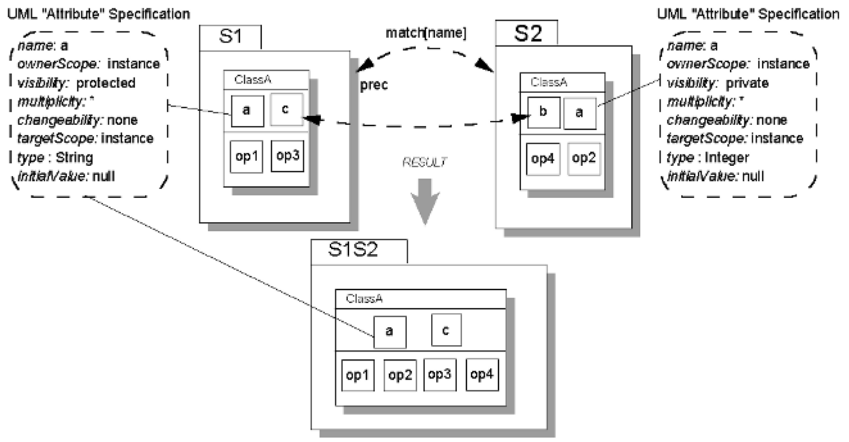


**Figure 29: Merge Integration with Reconciliation Specification**

By adding a precedence indicator to `S1` (see Figure 29), the result of the merge is:

- `S1.ClassA.a` and `S2.ClassA.a` correspond from `match[name]` merge relationship between `S1` and `S2`. Since their specifications are different, and precedence has been specified for `S1` (from composition relationship with merge integration between `S1` and `S2`), `S1.ClassA.a` is added to the result.
- `S1.ClassA.b` and `S2.ClassA.c` correspond from composition relationship between the two. Again, since their specifications are different, and precedence has been specified for `S1`, `S1.ClassA.c` is added to the result.

Other reconciliation strategies are possible and are described in "Chapter 7: Merge Integration" on page 155.

*Specifying Interactions for Ordering Execution of Operations*

When the order of execution of corresponding operations is important, an interaction specifying this order should be attached to the merge relationship.[6] In this case, the attached interaction is added to the merged subject as the specification of the behaviour of corresponding operations (see Figure 30). All operations in the corresponding operation set must be included in any interaction defined.

6. Where the order of execution is not important, no interaction need be attached. In this case, an interaction is generated arbitrarily specifying when each corresponding operation is executed.

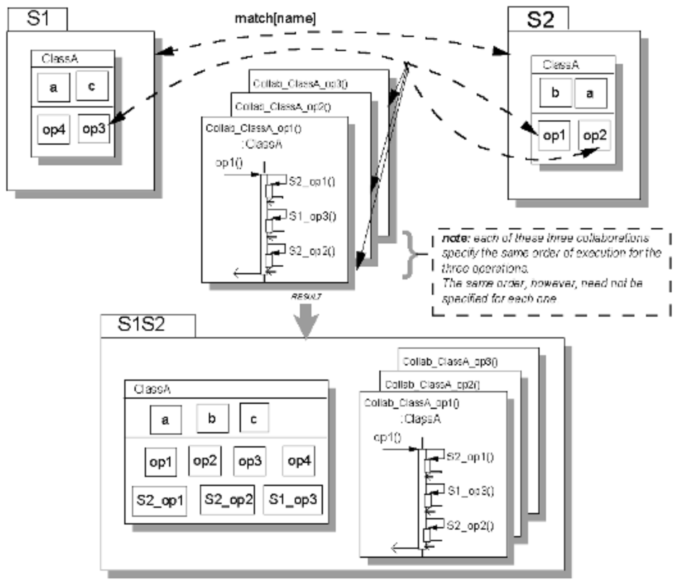


**Figure 30: Merge Integration with Interaction Specification**

In this example, the result of the merge is:

- `S1.ClassA` and `S2.ClassA` correspond from `match[name]` merge relationship between `S1` and `S2`. No conflict exists between the specifications, and so `ClassA` is added to the result.
- `S1.ClassA.a` and `S2.ClassA.a` correspond from `match[name]` merge relationship between `S1` and `S2`. No conflict exists between the specifications, and so `ClassA.a` is added to the result. `S1.ClassA.c` and `S2.ClassA.c` have no corresponding attributes and so are added to the result.
- `S1.ClassA.op3`, `S2.ClassA.op1` and `S2.ClassA.op2` correspond from the merge relationship between them. All the operations are added to the result. The interactions attached to the merge relationship are added to the result indicating that on receipt of an `op1` or an `op2` or an `op3` message, `op1` followed by `op3` followed by `op2` are executed.

Notice in Figure 30 that operations have been added to the result in order to capture the interaction between the corresponding operations. For a full discussion on the options considered for capturing this behaviour, and a description of the approach taken, see “Impact of Merge on Operations” on page 188.

**Notation** Composition relationships are graphically represented with dotted arcs as have been illustrated previously in examples. Composition relationships with merge integration are represented with multi-headed arrows at the inputs to the arcs. Composition relationships with override integration are represented with single-headed arrows, with the arrowhead at the end of the element to be overridden. In many cases, additional relationship specification (for example, implicit matching specification such as `match[name]`) may be attached to the relationship. There are other cases, however, when the extent of the specification associated with a composition relationship makes it unwieldy to represent the full specification graphically in a diagram. It is recommended that a CASE tool support the selection of composition relationships, and the representation of all the appropriate associated specification in supporting dialogs.

In all the examples illustrated in this thesis, each of the composition relationships under discussion are represented in the illustrations. However, the examples are very small, and it is easy to imagine that the number of composition relationships might be large where models are large. For this reason, it is also recommended that a CASE tool supports the representation of just the contextual level composition relationship (i.e., between input subjects), with dialog support illustrating the detail of all composition relationships at lower levels.

The examples in the thesis illustrating tree structures of subjects are purely to support discussion and explanation of the composition model, and are not considered part of the notation.

## 4.4. Analysis of the Output of a Composition

When composing design subjects, there is potential for the resulting subject to be “ill-formed”, from the perspective of the UML well-formedness rules [UML 1999]. One example is that composing design subjects with different generalization graphs may result in cycles, which are not permitted in the UML. There are many cases where composition may result in a breakage of a well-formedness rule of the UML. It is the policy of this composition model to perform the composition as specified by the designer with composition relationships, and highlight breakages to the well-formedness rules on the result. This is for the following reasons:

- *Difficulties with automated semantic reasoning*: A different approach to “compose first – check later” is to attempt to automatically “fix” elements that cause a breakage of the rules. This is possible in many cases and is discussed in “Forwarding of References” on page 96, but in some situations, the solutions to the correct properties to apply to elements in a composed subject are based on the domain of the computer system. “Chapter 6: Override Integration” and “Chapter 7: Merge Integration” illustrate examples in some detail. Though it is not advisable or desirable in many situations to automate “fixes”, it is considered most useful to compose subjects as specified by the composition designer and highlight problems in the result. The designer must solve the highlighted problems for the resulting subject to be well-formed.

- *Unanticipated results*: Where the composition process guarantees to perform the composition precisely as specified by the designer with composition relationships, the designer is protected from unanticipated results that might occur if automation of fixes to potential breakages occurs.
- *Validation of composition relationships*: Another approach to fixing potential problems is to halt the composition process at the first breakage, thereby ensuring that the result always conforms to the well-formed rules of the UML. However, performing the full composition tests the full set of composition relationships defined for the composition process, and provides the opportunity of assessing the impact of composition across the whole design. This may be required to solve some well-formedness problems.

There are, however, some areas in which the composition model may assist in alleviating difficulties in the result. In “Forwarding of References” on page 96 an approach to maintaining outside references to elements changed as a result of composition is discussed. Other difficulties could be avoided by extending the rules associated with specifying composition relationships and are discussed in “Ill-Formedness of Result” on page 99. These rules are not included in this version of the composition model.

## Forwarding of References

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. For example, in Figure 31, operation `S2.ClassA.op3` has a parameter of type `ClassC`, which is valid within the namespace of `S2`. However, when the elements of `S2` are overridden by the elements in `S1`, in particular when `S2.ClassC` is overridden by `S1.ClassB`, the resulting class in the output is called `ClassB`.

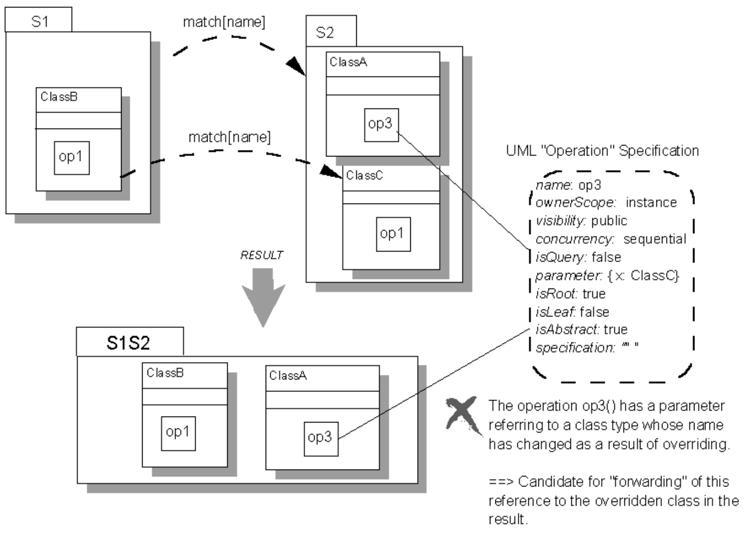


**Figure 31: Forwarding References to Composed Elements**

This example illustrates that in addition to copying the compositions of elements to an output, references to those elements could also be "forwarded" in the same output. This case is not ambiguous as to intent, and therefore, the subject-oriented design model supports the forwarding of references to changed elements within other elements to the output subject.

There are cases, however, where there may be some ambiguity as to which composed elements in the result references should be forwarded to. This occurs because of the possibilities allowing design elements to participate in multiple composition relationships within the same composition context (see "Participation in multiple composition relationships" on page 86). Consider the example in Figure 32. Because of the participation of `S1.ClassA` in two different composition relationships with merge integration, two classes to which `ClassA` has contributed appear in the output subject. This causes ambiguity of forwarding for `ClassX.a` in the result, since it has a type of `ClassA`.

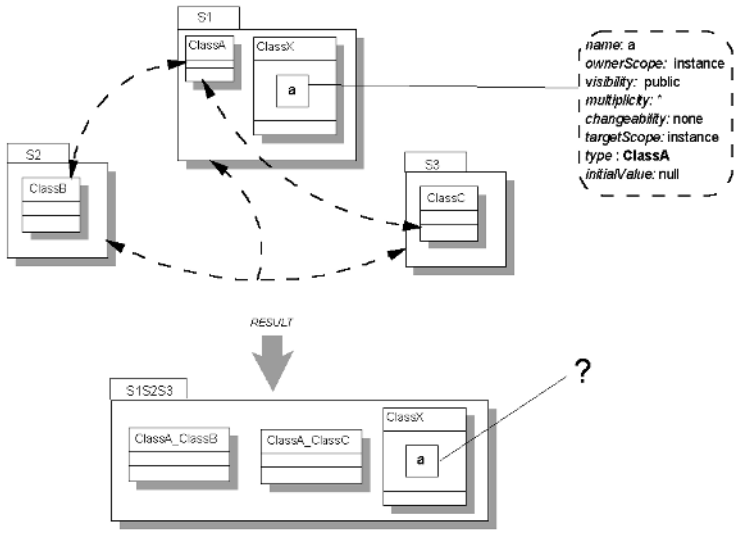


**Figure 32: Ambiguities with Forwarding of References**

To resolve this ambiguity, an additional attachment to composition relationships is supported. This attachment, called `[forwards]`, explicitly states the composition to which *all* elements of a particular input subject forwards. This attachment may be added to any or all ends of a composition relationship, but a restriction has been applied which negates any ambiguity:

> [Forwards Rule 1] Where a design element participates in multiple composition relationships within a single composition context, only one of those composition relationships may be annotated as specifying the result to which all referring elements within the input subject forward.

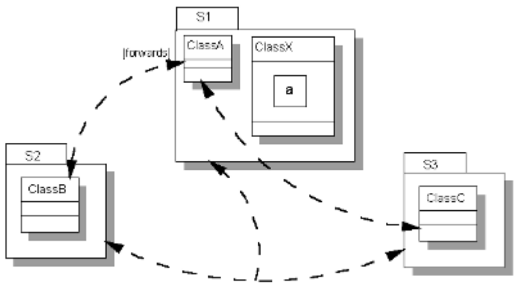


**Figure 33: Resolving Ambiguities with Forwarding of References**

Figure 33 illustrates the `[forwards]` attachment to the previous example. As specified, any elements in subject `S1` will forward to `ClassA_ClassB` in the output. In this example, no annotation is required for elements referring to `ClassB` within `S2`, or to `ClassC` within `S3`, as each of these only participate in one composition relationship.

*Discussion*

While the ambiguity relating to forwarding has been cleared up with this annotation, there is an implied restriction in [Forwards Rule 1]. This rule means that *all* of the elements in a particular input subject which refer to the element participating in multiple compositions will refer to the *same* result in the output. An alternative to this approach is to support the analysis of each individual reference within an input subject and the selection of the particular forwarding result from multiple compositions for each one. This is a more flexible approach, and will be considered for the subject-oriented design model in its future iterations. Detailed research is required to assess the suitability of the approach, as it may cause its own problems - for example, specification of forwarding for individual elements may prove unwieldy and difficult to maintain. However, for the purposes of this thesis, the ambiguity which is the cause of difficulty is closed. The price for the simplicity of the solution is the lack of flexibility.

**Ill-Formedness of Result**

The current flexible approach to allowing composition of any design elements so long as they have the same type (and their parents correspond) has the potential to create other difficulties. This section looks at two areas of concern in particular:

- Constraints on elements specified in input subjects may be lost in the output as a result of overriding or reconciliation of conflicts.
- Elements which may be the same type, but which may be incompatible in other ways, may be composed.

*Loss of Constraints*

Each design construct within the UML has a number of properties that provide information about, or constraints on instances of that construct. For example, attributes and operations have a visibility property which states whether it is public, protected or private. The particular semantics of a design subject may dictate the values of such properties for elements within the subject as a whole. However, as illustrated in Figure 34, such constraints may be easily lost within a composition context as a result of the use of override integration, and also, reconciliation of conflicting elements in merge integration.

In Figure 34, subjects `S1` and `S2` are merged, where elements with the same name are corresponding, and precedence is given to elements within `S1` in the event of a conflict. As a result of this composition specification, `S1.ClassA.a` and `S2.ClassA.a` are corresponding, and therefore merged. However, their specifications are different (particularly, the values of the `visibility` property conflict), but, because of the precedence given to `S1` in the composition relationship, the specification of `ClassA.a` in `S1` is copied to the output. However, this means that any elements in `S2` which worked with `ClassA.a` as a public attribute no longer work. Of course, that may not have been a very object-oriented approach for those elements, but nonetheless, it illustrates the point of where difficulties can arise as a result of composition.

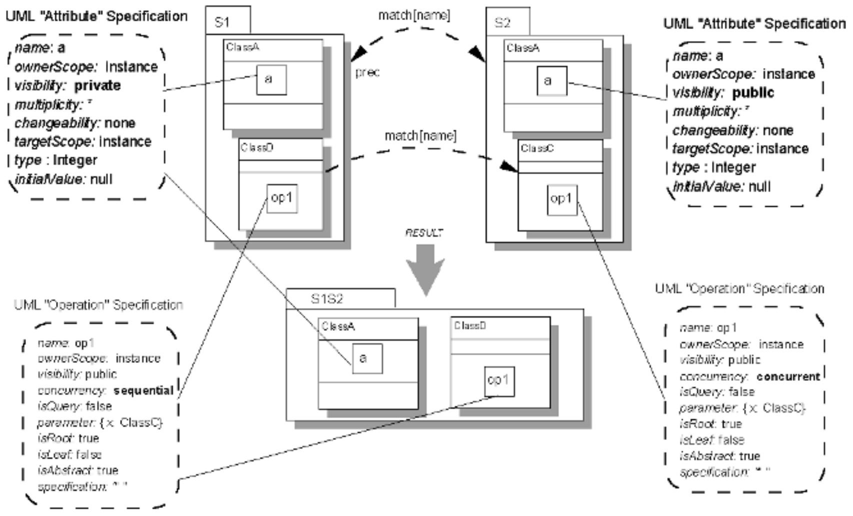


**Figure 34: Loss of (some) Constraints in Input Subjects**

A more sympathetic example is the impact on the operation `S2.ClassC.op1` as a result of being overridden by `S1.ClassD.op1.` The specifications of the two `op1`s are different in the values of the property `concurrency`. The semantics of `S2` may be such that expectations of `ClassC.op1` are that it works concurrently. However, as a result of being overridden by an `op1` that is specified as sequential, expectations of concurrent behaviour specified within the output (copied from `S2`) will cause problems. Another possible problem with operations not illustrated here is changing the visibility of an operation from public to private. There is potential for causing difficulties with changing the specifications of all properties of all constructs in an output.

*Incompatible Elements*

The examples given in the previous section illustrate difficulties with changing the specifications of elements as a result of composition. Another example of where difficulties may arise in the output subject exists where the specifications of composed elements are each added to the result, but merged. The semantics of merging operations adds all corresponding operations to the output, and specifies that on execution of one of those operations, all corresponding operations are executed. This behaviour has potential for composing operations which are semantically incompatible from an execution perspective.

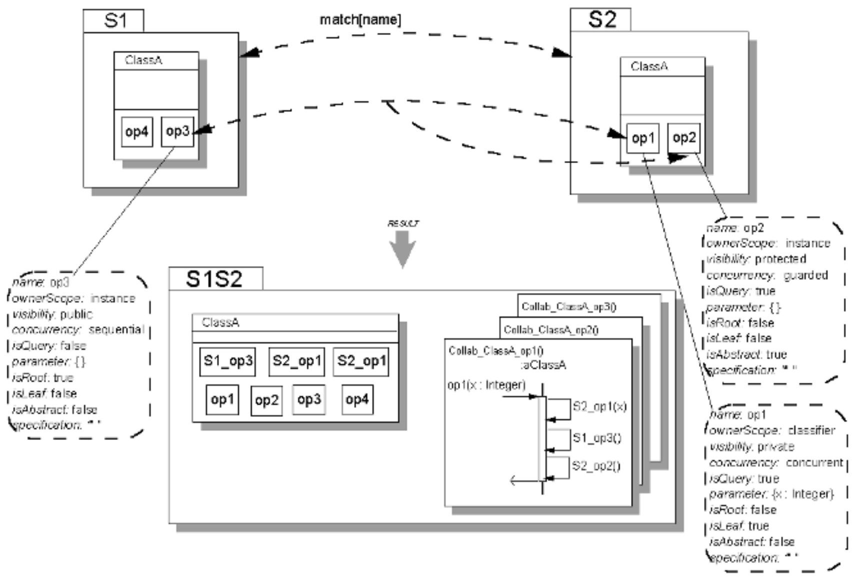


**Figure 35: Composing Incompatible Operations**

Consider the example in Figure 35. Three operations are merged which have differences in their specifications for every property of the operation construct. The semantics that defines that each of the corresponding operations are executed means that the design specification in the output subject specifies (in the automatically generated interaction) that the execution of a public method (op3) will also result in the execution of the protected and private methods. Another difficulty is in relation to the differences in the number of parameters - from an implementation perspective, this is not currently supported in programming models.

This discussion also applies for override integration, as it is currently possible to override one operation with another that is essentially incompatible from an implementation perspective. For example, they may have differing parameter lists (for example, cardinality or types differences), which would have an impact on clients of the overridden operation.

*Discussion* Loss of constraints for input subjects and the possible composition of essentially incompatible operations is an area of concern for the subject-oriented design model. An approach to avoiding such difficulties might be to build a taxonomy of rules associated with the validity of composing elements based on the values of properties (and the combination of values of properties), of elements of all types. For example, one rule might be:

> [Example Rule] Only those elements with the same visibility may be composed.

This approach requires that every possible value of every property of every construct (and every combination thereof), be examined to assess whether a rule guarding against composition is required, where values are different. Catalysis has a small number of rules for joining classes that go some way towards avoiding problems with constraints (for example variable types must be the same [D'Souza & Wills 1998]), but these do not go far enough to avoid all possible difficulties. A full taxonomy of rules based on all possibilities of values is required.

Such a set of rules associated with the specification of composition relationships would guard against the loss of constraints in input subjects, and ensure that incompatible elements are never composed. This piece of work would be a valuable addition to the subject-oriented design model, and is added to the future work.

Without this set of rules (as is the case with the model described in this thesis), it is the responsibility of the designer to use caution when specifying composition relationships. The designer should examine the output to ensure the semantics of the input subjects are preserved. However, allowing the merge integration of incompatible elements results in a model of operation execution that is unsupported both in UML and programming languages. For this reason, and in the absence of an appropriate taxonomy of rules, the subject-oriented design model **deems operations with different specifications to be non-corresponding, and therefore they will not be merged**. A single exception is made to this rule when the conflict in specifications is related to the parameter lists. This case is permissable when the designer specifies an interaction detailing the behaviour when these operations are executed. This exception is described in more detail in "Conflict Rules for Merging Operations" on page 192.

## 4.5. Using Subject-Oriented Design

In this section, the phases of a software development process where the subject-oriented design model may be used are described. Then, some possible issues with, and limitations to, the usage of the model are discussed.

**Usefulness throughout 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 designed elsewhere, which might be integrated with 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.
- *New versions of existing systems are required, based on changes to the supported business process.* The previous design of certain requirements is no longer applicable because of changes to the business process. Requirements are received that describe changes to the behaviour of the system as specified previously. Again, the changed requirements may be designed separately in different design subjects. In this case, the integration of the new design subjects replaces (i.e. overrides) the obsolete requirements in the previous version with the new design subjects.
- *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 requests 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 thesis to impose any particular development process for use with the composition model. This list of possible areas of usefulness throughout a 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 subsequent composition of those models.

## What Size is a Subject?

The subject-oriented design model does not explicitly recommend any particular "size" for a design subject. If a design subject is measured by the number of design elements contained within, then the size will be dictated by what is necessary to support the particular requirement under design by that subject. Other design approaches provide some guidelines as to the size of their different models. For example, the OORam model described in [Reenskaug et al. 1995] provides some loose guidelines for the size of role models, based on the notion that human short term memory can manage seven plus or minus two notions at the same time. The suggested guideline, therefore, is that a role model should consist of between five and nine roles - where fewer than five roles should be synthesised into a larger role model, and where consideration should be given to further breaking up a model with greater than nine roles. While the subject-oriented design model does discuss further decomposing design subjects where an analysis of the requirement it supports lends itself to such division (see Figure 15 on page 68), such decomposition is recommended based on possible logical divisions within the requirement, and not the "size" of the design subject.

## Duplication of Effort

As described in "Overlapping Subjects" on page 70, it is expected that some of the same basic domain concepts may be used in multiple design subjects. These domain concepts may require different specifications in different subjects to support different requirements. For example, requirements to check and evaluate expressions both work with a basic expression, but have different behaviour to handle the different requirement. Therefore, there are benefits in the ability to design these perspectives separately. The benefits include increased comprehensibility, traceability, evolvability and reuse capabilities. However, there is also some potential for overlap where the same concepts do not require different specifications for the different requirements they support. In this case, there is a danger of duplication of designer effort in the design of those concepts. This danger is inherent in this approach, but can be alleviated with careful decomposition of the design models. In addition to decomposing the design models based on structuring with the requirements, consideration could also be given to areas of the domain which may be re-used unchanged in many parts of the design. Such areas of the domain might also be separated into a design subject. In terms of matching with requirements, this is viewed as a case similar to that illustrated in Figure 15, with one difference - one of the subjects may be re-used for multiple requirements.

Where the area of overlap is very small, or not obvious to the designers immediately, it may be more difficult to initially assess that it should be designed as a separate subject, and duplication of effort may occur. This is an area of concern which requires further research to assess its impact. Part of this assessment might be to calculate the benefits of decomposition in this area against the cost of some duplication of effort because of overlap where the specifications for different requirements are the same.

### How Complex is Composition Specification?

Composition specification with composition relationships is flexible in the kinds of compositions allowed. Within the context of a composition relationship between elements at the roots of the subject trees to be composed, multiple other composition relationships may be specified between elements at levels lower in the tree, 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 levels of communication between different design teams and thereby decreasing the complexity of composition specification versus totally isolating the design teams, thereby increasing the likelihood of more complex composition specification. Depending on the personnel make-up of the overall team in terms of levels of experience and knowledge, and 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, and also with experience with the specification of composition relationships, thereby supporting more isolation.

### Feature Interaction Problem

The so-called *feature-interaction problem* is well documented for the telecommunications domain ([Jackson & Zave 1998], [Zave 1999], [Turner 1999]), and seems like an ideal problem for which subject-oriented design would find a solution - the problem should, in some cases, influence the choice of input subjects to a composition. The feature interaction problem is defined in [Zave 1999] as:

> A bad feature interaction is one that causes the specification to be incomplete, inconsistent, or unimplementable, or that causes the overall system behavior to be undesirable.

[Turner 1999] describes defining “conflicts” or “competes” or “constraints” relationships between features in order to capture problems between their potential interactions. The subject-oriented design model currently does not support such relationships between subjects, but support is possible with some extensions to the model. For example:

- an extension of the dependency relationship in UML to include stereotypes to support similar kinds of dependencies to those described in [Turner 1999].
- an extension to the rules associated with the specification of composition relationships to cater for such dependencies when defining inputs to a composition. For example, a rule might be included that states that subjects that conflict may not participate in the same composition context.

An interesting part of this future work is a study of how such extensions to the subject-oriented design model will support the specification of how features interact and how they may conflict when or if they are composed.

## 4.6. Chapter Summary

This chapter provides a description of the subject-oriented design approach to designing software. Motivated by the need to remove scattering and tangling properties in standard object-oriented designs, the model is based on adding decomposition capabilities to structure designs more directly with the structure of requirements specifications. Corresponding composition capabilities support considerable flexibility in the decomposition of design models.

First, this chapter describes how design models may be decomposed into *design subjects*. A design subject encapsulates a requirement, providing a complete design for that requirement, without redundant design elements. Changes to the design as a result of new requirements may themselves be encapsulated into design subjects, thus making changes to the design additive rather than invasive. Impediments to the reuse of designs were described in “Chapter 2: Motivation” on page 11 as rooted in the tangling of multiple requirements in design models. With the approach described in this chapter, each design subject supports a single requirement, where every element within the design subject is needed to support that requirement, and no redundant design elements are included. Even requirements that cross-cut other designs may be designed separately and without explicit reference to other design models.

The model is then further developed with a description of the means of composing design subjects - *composition relationships*. Composition relationships identify the subjects to be composed, overlaps within those subjects to be integrated as overlapping concepts, and how the elements should be integrated. Considerable flexibility is illustrated with different combinations of composition relationships supported. Patterns of composition may be identified and specified separately, providing support for the encapsulation of cross-cutting requirements, and their re-use.

The framework for composition involves the composition of input subjects to an output subject. This chapter analyses this output and illustrates how difficulties may occur as a result of composition. One category of difficulties relating to references to integrated elements is handled by the composition relationship. Solutions to other categories of difficulties - the loss of input subject constraints and the possible incompatibility of integrated operations - are proposed but not included in this version of the model.

The composition relationship is a new design construct which needs to be added to the UML metamodel. This is described in “Chapter 5: Composition

Relationship: An extension to the UML Metamodel" on page 109. A more detailed description of the semantics of the two integration strategies described in this research, override and merge, are in "Chapter 6: Override Integration" on page 127, and "Chapter 7: Merge Integration" on page 155. How to specify patterns of collaborative behaviour is described "Chapter 8: Composition Patterns" on page 198.

# Chapter 5: Composition Relationship: An extension to the UML Metamodel

The model for composing object-oriented designs described in this thesis is based on composing a number of input subjects into an integrated output subject ("Chapter 4: Composition of OO Designs: The Model" on page 64). Overlapping elements in input subjects are integrated as corresponding concepts. Different kinds of integration strategies are possible. The means for specifying composition proposed and developed in this thesis is a new kind of design relationship called a *composition relationship.*

This new design construct for specifying how to compose design models (the composition relationship) needs to be defined in the context of the design language used. This chapter describes how the UML may be extended to include the notion of a composition relationship. A composition relationship is an extension to the language, and as such, is defined within the context of the UML. This is achieved by extending the UML Metamodel as currently described in [UML 1999].

## 5.1. The UML Metamodel

As stated previously in "4.1. Decomposing Design Models" on page 65, a design subject may, conceptually, be written in any design language, but the focus of this thesis is the UML [UML 1999]. The UML is the OMG's standard language for object-oriented analysis and design specifications. The OMG currently defines the language using a *metamodel.* The metamodel defines the syntax and semantics of the UML, and is itself partially described using the UML. The metamodel is described 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, 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.

Because of its usage of natural language, this description of the UML is not a completely formal specification, and therefore, it is assumed in this research that ambiguities exist within its specification. The difficulties associated with extending the UML (in this case, by adding composition capabilities) are further compounded by the fact that the UML is in the early stages of its life, and is continuously undergoing changes - for example, work on Version 2 starts this year. The changes are being made for a number of reasons, including corrections, and filling gaps in the existing specification.

The ideal situation in which to extend the UML would be if the standard language, upon which this work is based, were completely and formally defined, and not undergoing change. Since this is not the case, the problem must be worked around. Providing a complete and formal specification of the standard UML is beyond the scope of this work, and ensuring that it does not undergo further change is beyond our control, not to mention inappropriate at this time. What is within our control, and within the scope of this work, is providing a semi-formal description of the syntax and semantics of composition relationships, in a style compatible with the current UML specification. Since the UML could be considered a moving target, this work anchors itself on the version 1.3 beta R7 - the version most current when the bulk of this research was performed. Changes to the UML subsequent to this version will not be catered for in this thesis, but must be incorporated into future work in this area.

The composition capabilities proposed and described in this thesis are important additions to the UML. For this reason, their incorporation into the standard UML is considered a high priority. Therefore, it is appropriate that the description of this work is in a similar style to that of the description of the UML, and that references to constructs of the UML are as they are described by the OMG.

This thesis, therefore, describes the extensions required to the UML with the following subsections containing the relevant views of the extensions:

- a subsection with UML class diagrams describing the constructs of the composition, and their relationships. This includes definitions of the kinds of constructs that may participate in composition relationships (called composable elements), followed by the composition relationship itself.
- a subsection containing the well-formedness rules describing the constraints on instances of composition relationships.
- a subsection containing descriptions of the semantics of composition relationships. This includes a description of how corresponding elements are identified, and the semantics of forwarding references to elements in output subjects.

Details of the semantics of the supported integration strategies, and their impact on the language metamodel, are in subsequent chapters.

## 5.2. Composable Elements

As discussed in “Composable Elements” on page 73, not all of the constructs supported within the scope of this work are composable elements - that is, elements which may directly participate in composition relationships. The exclusion of some design elements is based on two criteria; first, whether the element logically belongs to another element and the semantics of that element mean that it does not make sense for the element to be composed by itself, and secondly, whether the element is considered to be a constraint on another element. One example of the first case is Parameter. Parameters are a logical part of the complete signature of an operation or method, and therefore it does not make sense for them to participate in separate compositions. Another example is AssociationEnds. These are logically part of the full definition of associations, and therefore it does not make sense for them to be considered separately for compositions. An example of the second case is instances of Constraints, which are appropriately considered as part of the model element to which they are attached. Other model elements that are not included are deemed part of the full specification of one of the model elements that may participate.

Figure 36 describes which constructs may be related by a composition relationship. The style for restricting the kinds of model elements that may participate in composition relationships 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 element 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.

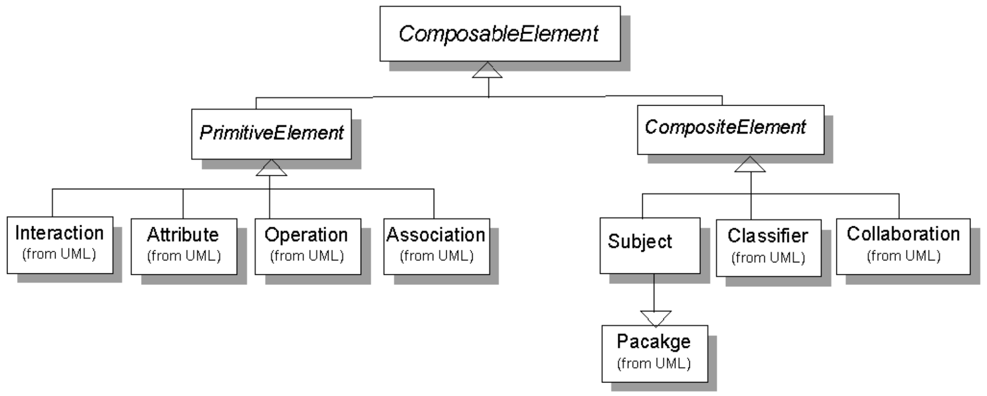


**Figure 36: Elements that may participate in Composition Relationships**

*ComposableElement Metaclass*

A composable element is a model element that may participate in a composition relationship. ComposableElement is an abstract metaclass.

*CompositeElement Metaclass*

A composite is a composable element that may contain other composable elements. Components of a composite are not considered part of the full specification of the composite for the purposes of composition, and are therefore considered separately for composition. The relationships between the composites and their components are unchanged from the specifications in the UML semantics, and are therefore not included here.

CompositeElement is an abstract metaclass.

*PrimitiveElement Metaclass*

A primitive is a composable element whose full specification may be composed with other primitives.

PrimitiveElement is an abstract metaclass.

*Subject Metaclass*

A subject is a subclass of Package, and has a more restrictive set of elements that may be owned or referenced than Package. A subject may only own or reference subjects, classifiers, associations, dependencies, generalizations, constraints and collaborations.

## 5.3. Composition Relationship

Composition relationships are the means for specifying how design elements should be composed. Composition relationships indicate elements that correspond, and how they should be integrated. This section describes the syntax of a composition relationship in the context of the UML metamodel. The meta-class diagram in Figure 37 illustrates:

- that composition relationships are specified between composable elements
- that a contextual composition relationship between subjects defines the context for a composition of subjects
- that composition relationships between design elements must be in the context of a contextual composition relationship (except when the composition relationship is itself the contextual one)
- that the specification of integration as an abstract metaclass attached to a composition relationship supports its specialisation for different integration strategies
- that the integration of design elements results in output design elements that are the result of the composition
- that a contextual composition relationship defines a namespace for outputs of the integration of design subjects and their components

The model supporting composition of design models also describes the need for forwarding of references to elements from within an input subject to references to appropriate elements in an output subject.

The meta-class diagrams illustrating the meta-class structure of a composition relationship are not sufficient to define the rules associated with a well-formed composition relationship. Similarly to the UML metamodel specification, well-formedness rules for composition relationships are also described in this section.

**Description of Constructs**

Each of the metaclasses in the class diagrams defining the syntax of a composition relationship are listed in this section with a description of their purpose. For each metaclass, a table describing any attributes and/or associations is included.

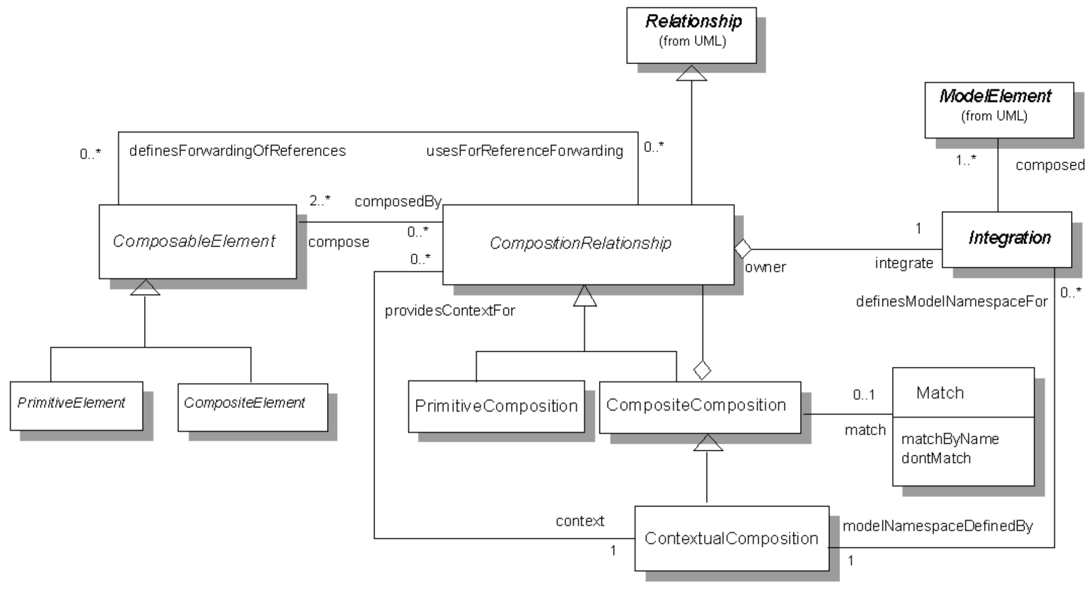


**Figure 37: Composition Relationship**

*ComposableElement Metaclass*

A composable element may participate in a composition relationship.

***Associations***

| | |
|---|---|
| *composedBy* | The associated composition relationships specify how this composable element will be composed with the other related composable elements. |
| *usesForReferenceForwarding* | The associated composition relationship defines the composed element to which references to its input element should forward. The cardinality for this relationship is 0..* because composable elements may participate in multiple composition contexts, or none at all. A well-formedness rule is included to ensure that there is only one forwarding composition specified within a single composition context. |

*CompositionRelationship Metaclass*

A composition relationship is a relationship between composable elements, recognising overlaps in concept specifications by identifying corresponding elements, and specifying how elements are to be integrated.

CompositionRelationship is an abstract metaclass.

***Associations***

| | |
|---|---|
| *compose* | The composable elements related by this composition relationship. |

| | |
|---|---|
| *integrate* | The integration strategy for this composition |
| *context* | The contextual composition relationship that provides the composition context for this composition |
| *definesForwardingOfReferences* | All references to this input element (from *compose* relationship), in its container subject, forward to the composed result specified by this composition relationship. |

*CompositeComposition Metaclass*

A composite composition relationship is a composition relationship between two composites. Composites have properties other than their components [UML 1999], and these property specifications from corresponding composites are integrated, as defined by the integration semantics. A composition relationship between composites specifies how correspondences between the composites' components are identified, and also specifies their integration semantics. Where a composite is itself a component of another composite, its composition relationship takes precedence over any composition relationship its owner may participate in.

***Associations***

| | |
|---|---|
| *match* | The general matching criteria to be used to establish correspondence between the components of the composite. |

*ContextualComposition Metaclass*

A contextual composition relationship defines the context within which a composition of input subjects occurs. All further composition relationships between design elements that are components of the input subjects (that is, at levels further down the subject tree - see "Tree Structure" on page 72) are defined within the context of a contextual composition relationship - that is, they must specify a `context` relationship to a contextual composition relationship. The contextual composition relationship also defines a namespace within which it, and each of the composition relationships for which it provides a context for, is contained.

***Associations***

| | |
|---|---|
| *providesContextFor* | Any relationships between components of the input subjects related by a contextual composition relationship are defined within the context of this relationship. |

| | |
|---|---|
| *definesModel-NamespaceFor* | The namespace of the output subject resulting from the integration of the input subjects and their components is defined by the input subjects to the contextual composition relationship. The name of the output subject is the concatenation of the names of the input subjects. |

*PrimitiveComposition Metaclass*

A primitive composition relationship is a composition relationship between two primitives. The full specifications of elements are composed with the full specification of the corresponding elements. A primitive composition relationship takes precedence over any composition relationship between composites that own the primitives.

*Match Metaclass*

With matching specified as part of the relationship, correspondence is established based on a match in the value of the `name` property of the elements.

***Attributes***

| | |
|---|---|
| *matchByName* | Indication that matching for correspondence identification is based on the value of the `name` property of elements. |
| *dontMatch* | A composition relationship between elements that specifies `dontMatch` indicates that those elements do not correspond. |

*Integration Metaclass*

Integration is an abstract metaclass that defines how corresponding elements are to be integrated. The result of the integration of corresponding elements is copied to one or more new design elements.

As an abstract metaclass, it is the intent that Integration be specialised to define the semantics of any integration strategy required. How this is achieved for override integration is described in “Chapter 6: Override Integration” on page 127, and for merge integration in “Chapter 7: Merge Integration” on page 155.

***Associations***

| | |
|---|---|
| *composed* | The result of integration (as defined by the semantics of subtypes of this metaclass) is copied to one or more new model elements. |
| *owner* | The composition relationship to which the integration specification is attached. |

| | |
|---|---|
| *modelNamespace-DefinedBy* | The contextual composition relationship that defines the namespace of the output subject that contains the result of an integration between composable elements. |

## Well-Formedness Rules

The well-formedness rules described in this section are included to ensure that composition of UML design models conforms to the general composition model as described in "Chapter 4: Composition of OO Designs: The Model" on page 64. The style used to define the rules is similar to that used to define the well-formedness rules of the UML. A textual description of the rule is followed, where appropriate, by an OCL (Object Constraint Language [Warmer & Kleppe 1999]) specification. The reasons why these rules are required are also included with each rule, which may, in some cases, be simply a reference to the appropriate part of the description of the model in "Chapter 4: Composition of OO Designs: The Model" on page 64.

### *Structural Rules*

**[1]** Composition relationships may only be specified between design elements of the same type[1].

```
self.compose->forAll (c1, c2 |
     c1.oclType = c2.oclType )
```

where `self` is an instance of `CompositionRelationship`

This rule is included because it is required by the composition model as described in "Inputs are the Same Type" on page 84.

**[2]** PrimitiveComposition relationships may only be specified between primitive elements.

```
self.compose->forAll( c |
     c.oclIsKindOf(PrimitiveElement))
```

where `self` is an instance of `PrimitiveComposition`

In this metamodel, a distinction is made between composition relationships that are between primitive elements and between composite elements since the specification for composite elements includes the possibility of attaching match criteria for components of the composite (see "Primitive vs. Composite" on page 74 for a description of the distinction between the two). Since the distinction is made at the meta-levels, this rule is included to ensure that primitive composition relationships are between primitive elements.

1. Operations used in well-formedness rules (e.g., `compose`) are defined in "Additional Operations" on page 120.

**[3]** CompositeComposition relationships may only be specified between composite elements.

```
self.compose->forAll( c |
     c.oclIsKindOf(CompositeElement))
```

where `self` is an instance of `CompositeComposition`

See previous rule, as discussion also applies to composite composition relationships.

**[4]** A contextual relationship is not defined within the context of another contextual relationship

```
self.oclIsKindOf(ContextualComposition) implies
     self.context.isEmpty
```

where `self` is an instance of `CompositionRelationship`

A contextual relationship is a relationship between the roots of a subject tree that defines the composition context for composition of the elements at lower levels of the tree (see “Specifying Inputs” on page 79). Since this relationship is between the roots of the subject tree, it is meaningless for the relationship itself to have a context, as there are no higher levels of the tree.

**[5]** A contextual relationship is only defined between subjects.

```
self.oclIsKindOf(ContextualComposition) implies
     self.compose.forAll( c |
          c.oclIsKindOf(Subject))
```

where `self` is an instance of `CompositionRelationship`

This rule reinforces that contextual composition relationships must be between subjects.

**[6]** All kinds of composition relationships other than the contextual composition relationship are defined with a `context` relationship to contextual composition relationship.

```
self.oclIsTypeOf(PrimitiveComposition) or
self.oclIsTypeOf(CompositeComposition) implies
     not self.context.isEmpty
```

where `self` is an instance of `CompositionRelationship`

The specification of composition of input subjects first involves the specification of a composite composition relationship between the roots of subject trees - that is, between the input subjects. This relationship defines a name-space within which composition of elements at levels of a subject tree lower than the root occurs. Therefore, every composition relationship between levels of a subject tree lower than the roots must be defined relative to the composition relationship between the roots of the tree (see “Rules for Specifying a Composition Relationship” on page 84).

**[7]** 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

```
self.compose->forAll( c |
     self.context.compose->exists( s |
          c.owningSubject = s))
```

where `self` is an instance of `CompositionRelationship`

This rule reiterates that composition relationships between design elements at levels of a subject tree lower than the root must be in the context of a contextual relationship involving the root of each tree containing those elements.

**[8]** Composition relationships may only be specified between elements whose parents are corresponding, and therefore will be composed.

The specification of the semantics for identifying corresponding elements is described in “Semantics for Identifying Corresponding Elements” on page 122. These semantics should be considered for testing the well-formedness of composition relationships against this rule.

*Common Integration Rules*

**[9]** A composition relationship specified between input subjects defines the namespace for composed elements in an output subject

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

where `self` is an instance of `CompositionRelationship`

As described in “Specifying Inputs” on page 79, a contextual composition relationship defines the context for composition of all design elements within the input subjects, providing a namespace for their integration.

*Forwarding*

**[10]** Where a design element participates in multiple composition relationships in multiple composition contexts, within a single composition context only one of those composition relationships may specify the result to which all referring elements within the input subject forward.

```
self.usesForReferenceForwarding->forAll( c1, c2 |
     c1 <> c2 implies c1.context <> c2.context )
```

where `self` is an instance of `ComposableElement`

This rule is included to ensure that ambiguity for forwarding of references is removed by having only one possibility defined (see “Forwarding of References” on page 96).

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

```
self.inputComposableElements->forAll( cEl |
   exists(cr : CompositionRelationship |
     cr.definesForwardingOfReferences.includes(cEl)
      and cr.context = self ) )
```

where `self` is an instance of `ContextualComposition`

***Additional Operations***

**[1]** The operation `compose` returns a Set containing all related elements

```
compose : Set(ComposableElement);
compose = self.compose
```

where `self` is an instance of `CompositionRelationship`

**[2]** The operation `composedBy` returns a Set containing the composition relationships in which a composable element participates

```
composedBy : Set(CompositionRelationship);
composedBy = self.composed
```

where `self` is an instance of `ComposableElement`

**[3]** The operation `composed` returns a Set containing the composed elements

```
composed : Set(ModelElement);
composed = self.composed
```

where `self` is an instance of `CompositionRelationship`

**[4]** The operation `owningSubject` returns the subject that owns the composable element

```
owningSubject : Subject;

owningSubject = self.namespace
```
[2]

where `self` is an instance of `ComposableElement`

**[5]** The operation `usesForReferenceForwarding` returns the set of composition relationships defined as the result for forwarding of references

```
usesForReferenceForwarding :

          Set(CompositionRelationship);

usesForReferenceForwarding =

          self.usesForReferenceForwarding
```

where `self` is an instance of `ComposableElement`

**[6]** The operation `providesContextFor` returns the set of composition relationships for which the contextual composition relationship provides a context.

```
providesContextFor : Set(CompositionRelationship);

providesContextFor = self.providesContextFor
```

where `self` is an instance of `ContextualComposition`

**[7]** The operation `inputComposableElements` returns the set of composable elements that directly participate in composition relationships within the context of a single composition

```
inputComposableElements : Set(ComposableElement);

 self.providesContextFor->forAll(c |

 inputComposableElements->union(c.compose))
```

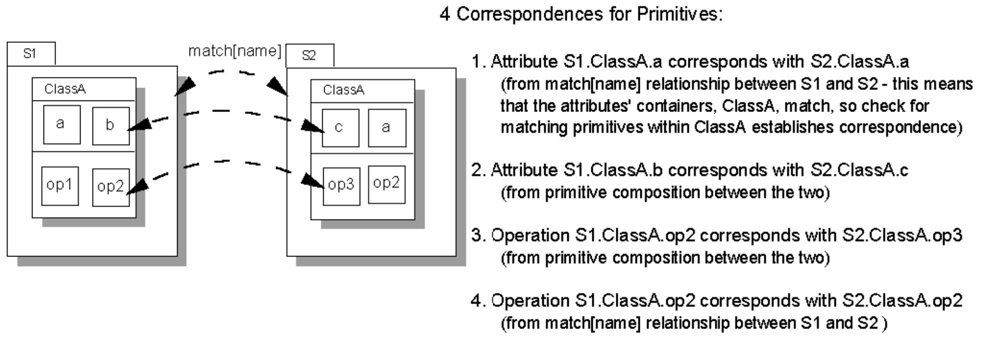
where `self` is an instance of `ContextualComposition`

2. The UML Metamodel states that a “namespace is used for unstructured contents such as the contents of a package..”. Since Subject is a stereotyped Package, then Namespace is considered in this thesis to be the designated name of the subject container of model elements.

## Semantics for Identifying Corresponding Elements

Where corresponding elements exist in input subjects, those elements must be identified prior to integration. This is because the semantics of integration must take the potential for overlapping of subjects into account. This section describes the semantics of how corresponding elements are identified based on a composition relationship. The semantics of the other primary responsibility of a composition relationship - integration - are described in subsequent chapters.

[1] Correspondence between primitives is established either directly with a primitive composition relationship, or indirectly based on matching from the specification of its bounding composition. Correspondence between primitives is not possible where the elements are components of non-corresponding composites. See Figure 38.



4 Correspondences for Primitives:

1. Attribute S1.ClassA.a corresponds with S2.ClassA.a
   (from match[name] relationship between S1 and S2 - this means that the attributes' containers, ClassA, match, so check for matching primitives within ClassA establishes correspondence)
2. Attribute S1.ClassA.b corresponds with S2.ClassA.c
   (from primitive composition between the two)
3. Operation S1.ClassA.op2 corresponds with S2.ClassA.op3
   (from primitive composition between the two)
4. Operation S1.ClassA.op2 corresponds with S2.ClassA.op2
   (from match[name] relationship between S1 and S2 )

**Figure 38: Correspondences between Primitives**

[2] Correspondence between composites is established in two ways:

- either directly with a general matching rule from a composite composition relationship,
- or indirectly with a general matching from a composite composition relationship between any owning composites at higher levels of the tree.

Correspondence matching between a composite’s components is established

- either by matching as specified in the composite composition relationship between their owners,
- or by additional relationships which take precedence over the composite composition relationship between their owners.

Any elements that participate in composition relationship with a “`dontMatch`” specification, do not correspond.

See Figure 39 for an illustration.

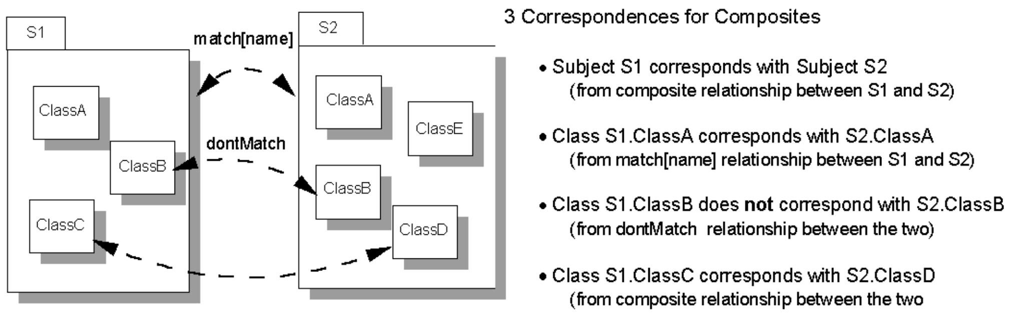

S1
ClassA
ClassB
ClassC
match[name]
dontMatch
S2
ClassA
ClassE
ClassB
ClassD
3 Correspondences for Composites
• Subject S1 corresponds with Subject S2
(from composite relationship between S1 and S2)
• Class S1.ClassA corresponds with S2.ClassA
(from match[name] relationship between S1 and S2)
• Class S1.ClassB does **not** correspond with S2.ClassB
(from dontMatch relationship between the two)
• Class S1.ClassC corresponds with S2.ClassD
(from composite relationship between the two

**Figure 39: Correspondences between Composites**

**Semantics for Forwarding References to Composed Elements**

The integration of corresponding input design elements results in an output design element which may be different from the input design elements, depending on the integration semantics. As defined by this composition model (see “Forwarding of References” on page 96), design elements that reference any of the design elements that are input to a composition will reference the resulting output element in the output subject.

[1] Every integration strategy composes design elements to one or more output design elements that are added to the composed contextual namespace.

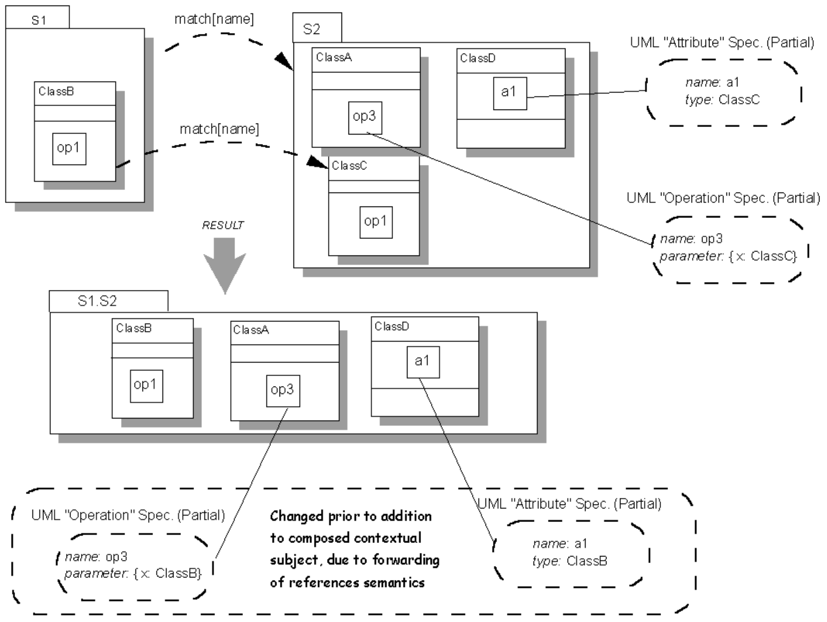

S1
ClassB
op1
match[name]
match[name]
S2
ClassA
op3
ClassD
a1
ClassC
op1
UML "Attribute" Spec. (Partial)
name: a1
type: ClassC
UML "Operation" Spec. (Partial)
name: op3
parameter: {x: ClassC}
RESULT
S1.S2
ClassB
op1
ClassA
op3
ClassD
a1
UML "Operation" Spec. (Partial)
name: op3
parameter: {x: ClassB}
Changed prior to addition to composed contextual subject, due to forwarding of references semantics
UML "Attribute" Spec. (Partial)
name: a1
type: ClassB

**Figure 40: Forwarding of References Semantics**

Prior to the addition of each output design element, any references to other design elements are examined. These referenced design elements are themselves added to the same composition contextual namespace, either unchanged or changed in some way as a result of the composition. Where changes to referenced elements has occurred, the semantics of forwarding references to them requires that the change is reflected in the referring element. See Figure 40 for an illustration.

[2] Where a design element referenced in an input subject participates in multiple compositions, the change to the reference is based on the composition relationship specified as its forwarding composition. See Figure 41 for an illustration.

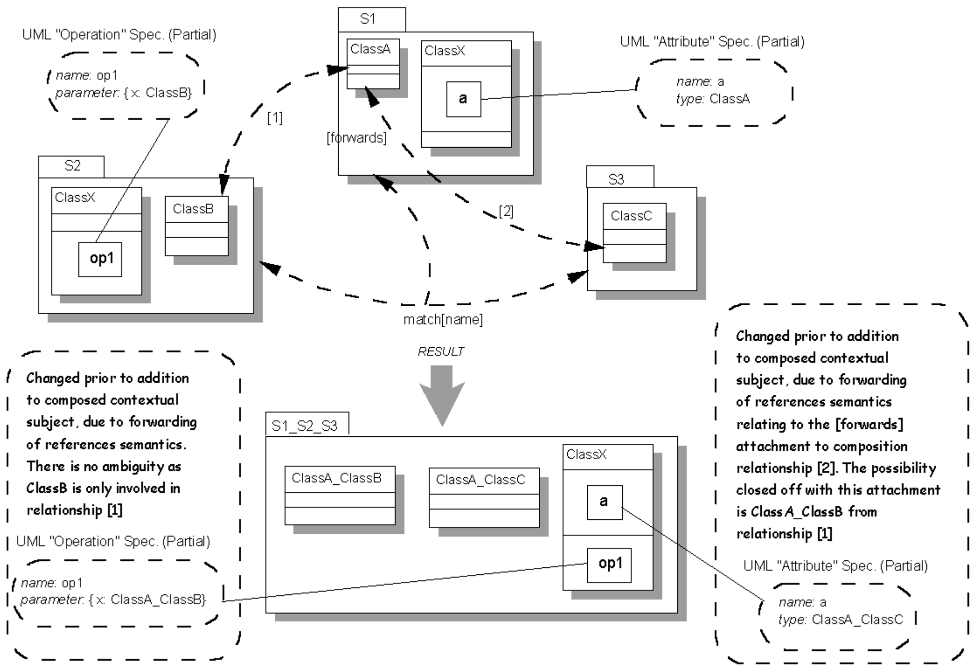

UML "Operation" Spec. (Partial)
name: op1
parameter: {x: ClassB}
S1
ClassA
ClassX
a
UML "Attribute" Spec. (Partial)
name: a
type: ClassA
[1]
[forwards]
S2
ClassX
ClassB
op1
S3
ClassC
[2]
match[name]
RESULT
Changed prior to addition to composed contextual subject, due to forwarding of references semantics. There is no ambiguity as ClassB is only involved in relationship [1]
UML "Operation" Spec. (Partial)
name: op1
parameter: {x: ClassA_ClassB}
S1_S2_S3
ClassA_ClassB
ClassA_ClassC
ClassX
a
op1
Changed prior to addition to composed contextual subject, due to forwarding of references semantics relating to the [forwards] attachment to composition relationship [2]. The possibility closed off with this attachment is ClassA_ClassB from relationship [1]
UML "Attribute" Spec. (Partial)
name: a
type: ClassA_ClassC

**Figure 41: Forwarding Ambiguous References with Attachment to Relationship**

[3] In addition to the situation where composable elements directly participate in multiple composition relationships, ambiguity may also occur as a result of implicit correspondence matching of elements from a composition relationship at a higher level of the subject tree. If there is no `[forwards]` attachment to a relationship in which the element causing the ambiguity directly participates, forwarding occurs to the result of the implicit matching. This is because it is not appropriate to allow a direct `[forwards]` attachment to a relationship between composite elements, as this relationship affects *all* elements at levels lower in the subject tree, not just the element causing the ambiguity. See Figure 42 for an illustration.

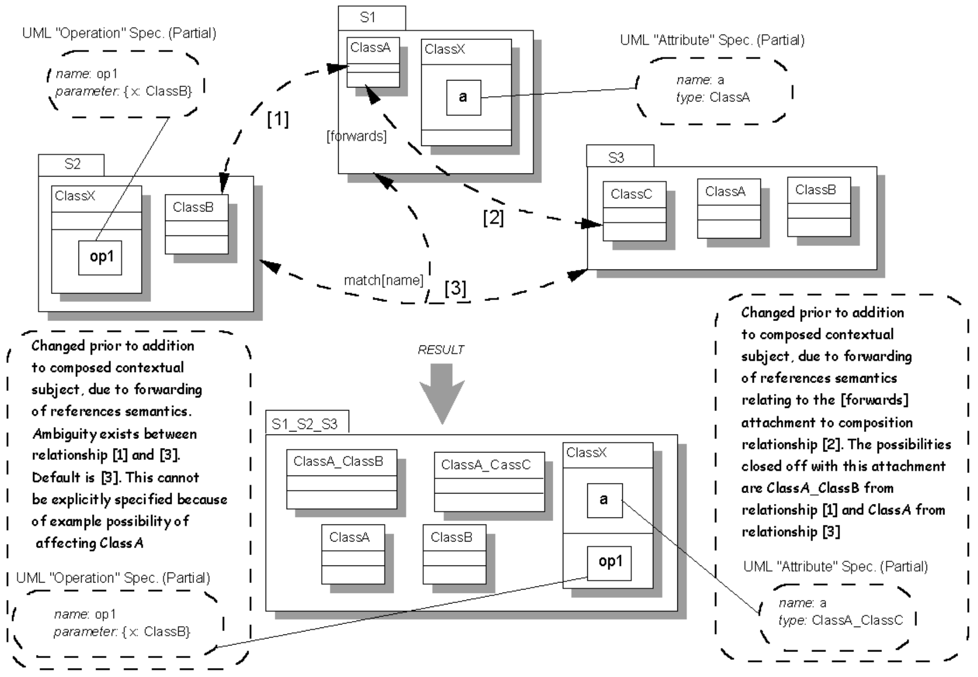

UML "Operation" Spec. (Partial)
name: op1
parameter: {x: ClassB}
S1
ClassA
ClassX
a
UML "Attribute" Spec. (Partial)
name: a
type: ClassA
[1]
[forwards]
S2
ClassX
ClassB
op1
S3
ClassC
ClassA
ClassB
[2]
match[name]
[3]
Changed prior to addition to composed contextual subject, due to forwarding of references semantics. Ambiguity exists between relationship [1] and [3]. Default is [3]. This cannot be explicitly specified because of example possibility of affecting ClassA
RESULT
S1_S2_S3
ClassA_ClassB
ClassA_CassC
ClassX
a
ClassA
ClassB
op1
UML "Operation" Spec. (Partial)
name: op1
parameter: {x: ClassB}
Changed prior to addition to composed contextual subject, due to forwarding of references semantics relating to the [forwards] attachment to composition relationship [2]. The possibilities closed off with this attachment are ClassA_ClassB from relationship [1] and ClassA from relationship [3]
UML "Attribute" Spec. (Partial)
name: a
type: ClassA_ClassC

**Figure 42: Forwarding to Implicit Composition Output as Default**

## 5.4. Chapter Summary

This chapter defines a composition relationship as an extension to the UML metamodel, using the same language and style as the specification of the semantics of the UML itself. Meta-class models describe the constructs associated with a composition relationship, and together with defined well-formedness rules, constraints on the syntax of composition relationships are specified.

Composing design subjects entails identifying corresponding elements within the design subject, and integrating the elements within the input subjects to a composed result in an output subject. This chapter also defines the semantics of identifying corresponding elements. Integration of elements is defined in the abstract so that concrete integration strategies may be seamlessly added to the metamodel. Common semantics for all kinds of integration are defined - that is, the integration of elements to an output subject, and the forwarding of references to elements in input subjects to appropriate references in the output subject.

Further extensions to the metamodel are required for each individual integration strategy that may be required. This thesis describes two kinds of integration, override and merge, and the extensions to the metamodel to support these integration strategies are included in “Chapter 6: Override Integration” on page 127, and “Chapter 7: Merge Integration” on page 155.

# Chapter 6: Override Integration

Override integration is used when elements in an existing design subject need to be changed. For example, new requirements may 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. Another possible scenario requiring override integration is when separate groups are working on individual subjects, where one group's element(s) specification(s) may change another group's specifications. Overriding an existing design subject, or elements within a design subject, is specified with *composition relationships* with *override integration.* These composition relationships are specified between the design subject requiring change, and a different design subject containing the new elements.

Composition relationships with override integration specify which design elements in the existing design subject are to be overridden by design elements in the new design subject. Any design elements in the existing design subject that are not overridden by design elements in the new design subject are added to the result unchanged. Any design elements in the new design subject that do not override design elements in the existing design subject are added to the result. This section details the semantics of composition relationships with override integration, and has the following subsections[1]:

- a subsection with UML class diagrams describing the constructs of the override, and their relationships
- a subsection containing the well-formedness rules describing the constraints on instances of overrides
- a subsection containing descriptions of the semantics of override

1. Only changes to the syntax and semantics of composition relationships (as specified in "Chapter 5: Composition Relationship: An extension to the UML Metamodel" on page 109) that are appropriate for override integration are described in this chapter.

## 6.1. Syntax

Override integration specifies that the specification of one design element is overridden by the specification of its corresponding element. Override integration is defined as a subclass of the Integration metaclass from the composition relationship (see Figure 43).

The semantics of override integration require that the cardinalities of the composable elements that may participate in a composition relationship are changed. As specified in “5.3. Composition Relationship” on page 113, a composition relationship may be specified between two or more composable elements. However, this is not appropriate when the integration strategy is override, as the semantics of override dictate that one composable element is overridden by one other composable element.

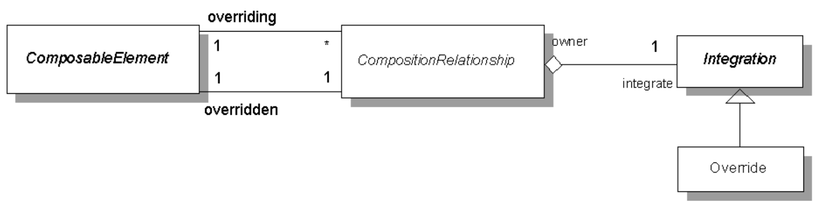


**Figure 43: Override Integration**

*CompositionRelationship Metaclass*

Override integration overrides one element with the specification of its corresponding element. This restricts the cardinalities of the composable elements related by the composition relationship to which the override integration specification is attached.

***Associations***

| | |
|---|---|
| *overridden* | The composable element whose specification is overridden |
| *overriding* | The composable element whose specification overrides the overridden element |

*Override Metaclass*

Override integration specifies that the specification of the overridden element is replaced by the specification of the overriding element.

## 6.2. Well-Formedness Rules

Override integration imposes more stringent restrictions on the number of composable elements that may participate in a composition relationship than are defined for the general case (see “Composition Relationship” on page 113). In the general case, two or more composable elements may be related by a single composition relationship. However, for a composition relationship with override integration, this is restricted to one composable element overriding one other (see Figure 43). This restriction means that the well-formedness rules must specify the replacements for a number of the rules defined for the general case. For each of the rules defined in this section, there is an indication, where appropriate, of which general rule is rewritten to suit a composition relationship with override integration. Every general rule defined in “Composition Relationship” on page 113 *not* explicitly replaced here applies to ensure the well-formedness of composition relationships with override integration.

**[1]** The composition relationship to which override is attached relates composable elements based on its `overridden` and `overriding` associations only.

```
self.owner.compose =
 self.owner.overriding->union(self.owner.overridden :
     ComposableElement) : Set(ComposableElement)
```

where

- `self` is an instance of `Override`
- and `compose` is an operation defined in the well-formedness rules for general composition relationships in “Additional Operations” on page 120

**[2]** The overriding and overridden elements are different.

```
self.owner.overridden <> self.owner.overridding
```

where `self` is an instance of `Override`

This rule is included as it does not make sense to override an element with itself. From the perspective of override semantics, this results in a design element that is unchanged in any way.

**[3]** Within the context of a single composition, a composable element may only participate in one composition relationship as the overridden element.

```
self.owner.context.providesContextFor->
 forAll(cr1, cr2 |
     cr1 <> cr2 implies cr1.overridden <>
                                cr2.overridden)
```

where

- `self` is an instance of `Override`

- and `providesContextFor` is an operation defined in the well-formedness rules for general composition relationships in "Additional Operations" on page 120

Override integration changes the specification of an element to be overridden. This rule is included because without it (that is to allow an element to be overridden multiple times by different elements) there may be unanticipated results. Without explicit ordering of the different integrations, it is not possible to predict the final specification of the overridden element. General ordering of multiple compositions is currently not supported in the subject-oriented design model.

## 6.3. Semantics

As stated previously, override integration is used to override design specifications in an existing design subject with design specifications in a design subject that reflect a change to the requirements since the existing design subject was created. Overrides indicate which elements in the existing design subject are to be overridden by which elements in the overriding design subject.

This section first discusses, in "General Semantics" on page 130, the general semantics of override that apply to all types of elements. Sections "Impact of Override on Subjects" on page 132 to "Impact of Override on Collaborations" on page 148 then consider the impact of override on each of the different types of elements, highlighting any differences with the general semantics.

### General Semantics

The identification of correspondences is the same as for all composition relationships and is described in "Semantics for Identifying Corresponding Elements" on page 122.

[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. From Figure 44, the following overrides occur:

- The specification of class `S2.ClassA` is changed to the specification of `S1.ClassA` as a result of override
- The specification of attribute `S2.ClassA.a` is changed to the specification of `S1.ClassA.a` as a result of override

- The specification of operation `S2.ClassA.op1` is changed to the specification of `S1.ClassA.op1`, as a result of override.
- The specification of operation `S2.ClassA.op2` is changed to the specification of `S1.ClassA.op2`, as a result of override.
- The specification of operation `S2.ClassA.op3` is changed to the specification of `S1.ClassA.op2`, as a result of override.

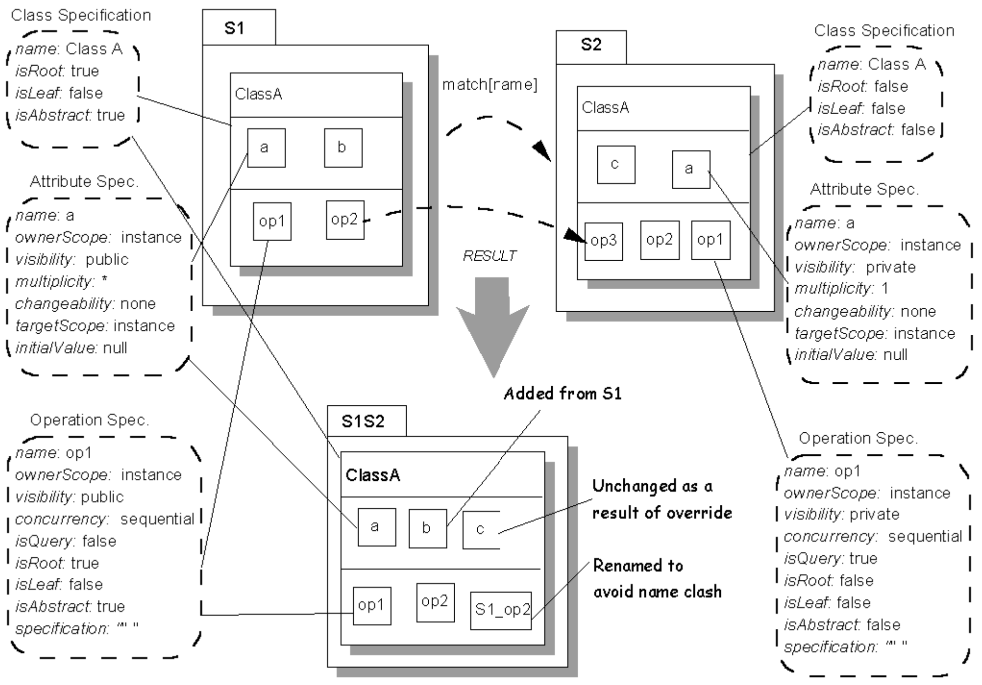


**Figure 44: General Semantics for Override Integration**

[2] Elements in an overridden composite that are not involved in a correspondence match remain unchanged. For example, from Figure 44, the attribute `S2.ClassA.c` has no corresponding elements, and so is added to the result unchanged.

[3] Elements that are components of an overriding composite and are not involved in a correspondence match are added to the overridden composite. For example, from Figure 44, the attribute `S1.ClassA.b` has no corresponding elements in `S2`. Since it is a component of an overriding class named `ClassA`, it is added to the specification of `ClassA` as a result of override.

[4] Changes to an overridden subject, either as a result of overriding of corresponding elements, or as a result of adding elements directly to the overridden subject, may not result in name clashes. In the event of name clashes, renaming of clashing elements occurs. For example, from Figure 44, overriding both `S2.ClassA.op3` and `S2.ClassA.op2` with `S1.ClassA.op2`

results in a name clash. To avoid this, the name of one of the overridden operations is changed.

[5] All references to elements in the result that may have changed from the specification in the input subject are changed as described in “Semantics for Forwarding References to Composed Elements” on page 123.

[6] The composed subject must conform to the well-formedness rules of the UML.

## Impact of Override on Subjects

This section discusses what happens to subject specifications as a result of override. (See “Appendix A: Partial Illustrations of UML Metamodel” on page 269 for an illustration of the UML specification of Package, from which Subject is stereotyped). The following are illustrated with an example:

- How correspondences are established
- The results of override on elements both corresponding and non-corresponding
- Checking the UML Well-Formedness Rules on the results of override
- Consideration of deviations from (or additions to) the general semantics defined in the previous section.

When the composition relationship between subjects does not have general correspondence matching criteria associated with it, there is not considered to be any corresponding elements in the subject’s contents, unless specified with additional relationships between its contents. The following subsections describe the impact of override on the example illustrated in Figure 45:

### *Correspondences*

- [Eg6.1] `S1` corresponds with `S2` because of the composition relationship between the two. This relationship specifies matching on name for identification of correspondence between the components, and is the contextual relationship for this composition example.
- [Eg6.2] `S1.S3` corresponds with `S2.S3` (Eg6.1)
- [Eg6.3] `S1.S4` corresponds with `S2.S5` (because of the relationship between the two.)

- [Eg6.4] `S1.ClassA` corresponds with `S2.ClassA` (Eg6.1)

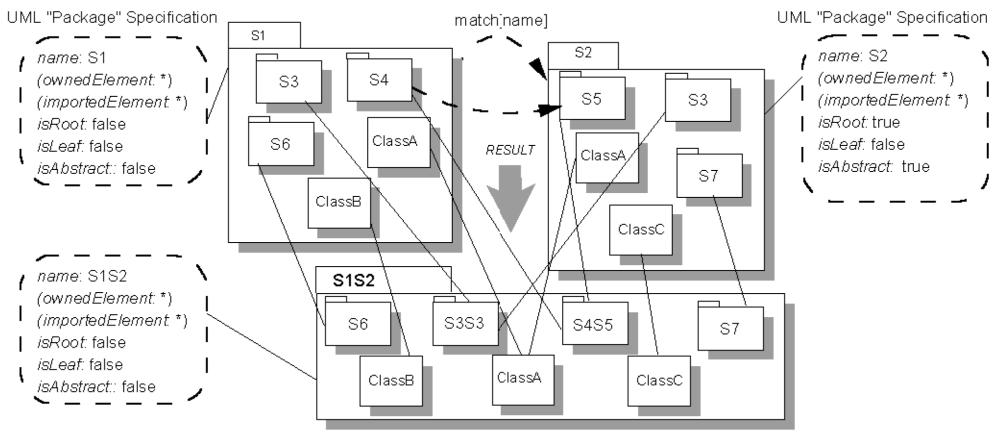


**Figure 45: Impact of Override on Subject Specifications**

*Result of Override*

Elements with correspondences:

- The specification of `S2` is changed to the specification of `S1`. This excludes the `ownedElements` and the `importedElements` as these are components of subjects. In addition, naming for subjects in the result is by appending the names of the overriding and overridden subjects. This conforms to the specification of the namespace of the output of the composition as defined by the contextual composition relationship and described in “Well-Formedness Rules” on page 117.
- The specification of `S3` in the resulting subject is that of the specification of `S1.S3`. The components of `S3` (in `ownedElements` and `importedElements`) are considered separately.
- The specification of `S5` in the resulting subject is that of the specification of `S1.S4`, with the names of the two concatenated. The components of both (in `ownedElements` and `importedElements`) are considered separately, with the resulting components contained in the `S4` in the result.
- The specification of `ClassA` in the resulting subject is that of the specification of `S1.ClassA` (see “Impact of Override on Classifiers” on page 134 for more details on classifiers). The components of `ClassA` are considered separately.

Elements with no correspondences:

- `S1.S6`, and `S1.ClassB` have no corresponding elements in `S2`. They are therefore added to the resulting subject, unchanged in any way, and without further consideration of their components.
- `S2.S7`, and `S2.ClassC` have no corresponding elements in `S1`. They are therefore added to the resulting subject, unchanged in any way, and without further consideration of their components.

*Check on UML Well-Formedness Rules*

The well-formedness rules for packages are not broken in this example.

*Differences with General Semantics for Override*

The semantics for overriding Subjects conforms to the general semantics for override, except for the naming of the result of composing subjects even when the composition relationship between those subjects is *not* the contextual composition relationship. Instead of overriding the name as per the general semantics for all composable elements, the names of subjects are always concatenated. The reason for this is to distinguish between the result and the overridden subject, and to make clear which subjects are composed.

## Impact of Override on Classifiers

This section discusses what happens to classifier specifications as a result of override. (See "Appendix A: Partial Illustrations of UML Metamodel" on page 269 for an illustration of the UML specification of Classifier). With an example, the impact of override on Classifiers is illustrated.

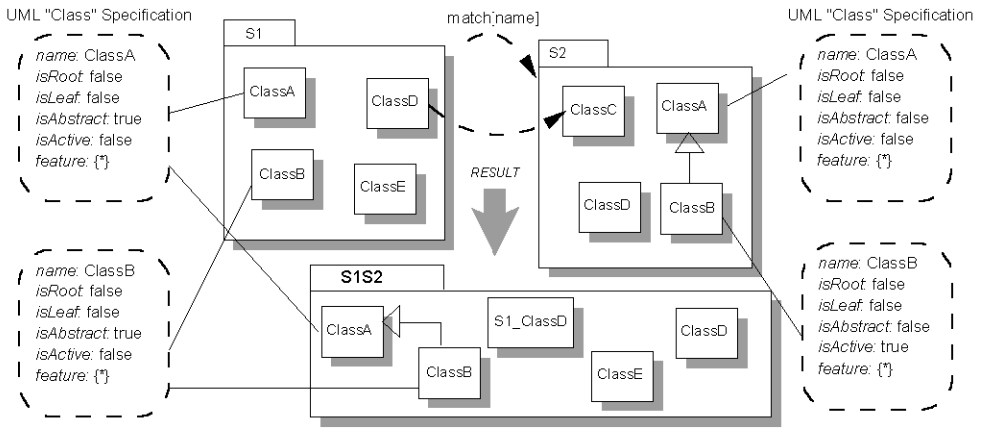


**Figure 46: Impact of Override on Classifier Specifications**

When the override relationship between classifiers does not have general correspondence matching criteria associated with it, there are not considered to be any corresponding elements in the classifier's contents, unless specified

]

with additional overrides between its contents. The subsections that follow describe the impact of override on the example illustrated in Figure 46.

*Correspondences*

- [Eg6.5] `S1` corresponds with `S2` because of the composition relationship between the two. This relationship specifies matching on name for identification of correspondence between the components
- [Eg6.6] `S1.ClassA` corresponds with `S2.ClassA` (Eg6.5)
- [Eg6.7] `S1.ClassB` corresponds with `S2.ClassB` (Eg6.5)
- [Eg6.8] `S1.ClassD` corresponds with `S2.ClassC` (from the composition relationship between the two)
- [Eg6.9] `S1.ClassD` also corresponds with `S2.ClassD` (from Eg6.5). Recall that composable elements may participate in multiple composition relationships (see "Participation in multiple composition relationships" on page 86) and override integration only restricts the overridden element, not the overriding element (see "Well-Formedness Rules" on page 128). Any correspondence *not* required which occurs implicitly as a result of a matching specification attached to a relationship at a higher level in the subject tree must be explicitly excluded with a composition relationship with a `dontMatch` attachment.

*Result of Override*

Elements with correspondences:

- In the result, `ClassC` has the specification of `S1.ClassD`, with one change. Since there is already a `ClassD` in `S2`, `S1.ClassD` is renamed to avoid a name clash. `S1.ClassD` is renamed to "`S1_ClassD`".
- The specification of `ClassD` in the resulting subject is that of the specification of `S1.ClassD`. The components of `ClassD` are considered separately.
- The specification of `ClassA` in the resulting subject is that of the specification of `S1.ClassA`. The components of `ClassA` are considered separately.
- The specification of `ClassB` in the resulting subject is that of the specification of `S1.ClassB`. The components of `ClassB` are considered separately.

Elements with no correspondences:

- `S1.ClassE` has no corresponding elements in `S2`. It is therefore added to the resulting subject, unchanged in any way, and without further consideration of its components.

*Check on UML Well-Formedness Rules*

The example illustrated in Figure 46 does not result in a breakage of the well-formedness rules of the UML.

However, with a small change as illustrated in Figure 47, it is easy to see where a breakage might occur. The illustration highlights (with a big X) where a breakage of the well-formedness rules of the UML may occur.

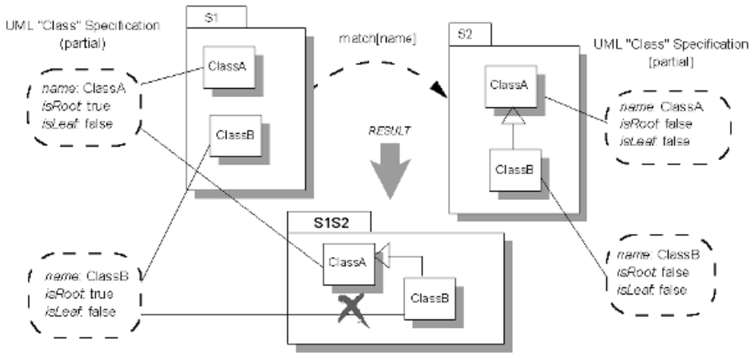


**Figure 47: Breaking Well-Formedness Rules for Classifiers**

This example results in one breakage of the UML well-formedness rules. Classifier is a subtype of GeneralizableElement (see "Appendix A: Partial Illustrations of UML Metamodel" on page 269), and must conform to the well-formedness rules of all generalizable elements. One rule for generalizable elements states that "A root cannot have any Generalizations" [UML Semantics Guide page 2-53, GeneralizableElement, Rule [1]]. The overriding `ClassB` specifies `ClassB` as being a root class, but `ClassB` in `S2` is specialised from `ClassA`. It is the responsibility of the designer to decide what action is appropriate. In this case, the designer could either remove the generalization, or change the value of `isRoot` in `ClassB`.

*Differences with General Semantics for Override*

- The semantics for overriding Classifiers must also take into consideration the impact of override on association ends. See " Impact of Override on Associations and Generalizations" on page 140 for more details.
- The semantics for overriding Classifiers must also take into consideration the impact on role specifications for collaborations. See " Impact of Override on Collaborations" on page 148 for more details.

## Impact of Override on Attributes

This section discusses what happens to attribute specifications as a result of override. (See “Appendix A: Partial Illustrations of UML Metamodel” on page 269 for an illustration of the UML specification of Attribute). The impact of override on Attributes is illustrated with an example.

The following subsections describe the impact of override on the example illustrated in Figure 48.

*Correspondences*

- [Eg6.10] `S1` corresponds with `S2` because of the composition relationship between the two. This relationship specifies matching on name for identification of correspondence between the components
- [Eg6.11] `S1.ClassA` corresponds with `S2.ClassA` (Eg6.10)
- [Eg6.12] `S1.ClassB` corresponds with `S2.ClassC` (from the relationship between the two. This relationship specifies matching on name for identification of correspondence between the components)
- [Eg6.13] `S1.ClassA` .a corresponds with `S2.ClassA.a` (Eg6.10)
- [Eg6.14] `S1.ClassB` .a corresponds with `S2.ClassC.a` (Eg6.12)
- [Eg6.15] `S1.ClassB.f` corresponds with `S2.ClassC.f` (Eg6.12)
- [Eg6.16] `S1.ClassB.f` corresponds with `S2.ClassC.e` (from the composition relationship between the two)

*Result of Override*

Elements with correspondences:

- The specification of `ClassA` in the resulting subject is that of the specification of `S1.ClassA`. The components of `ClassA` are considered separately.
- The specification of the attribute `a` in the resulting `ClassA` is that of `S1.ClassA.a.`
- In the result, `S2.ClassB` has the specification of `S1.ClassC.` The components of `S1.ClassC` and `S2.ClassB` are considered separately.
- The specification of the attribute `a` in the resulting `ClassB` is that of `S1.ClassB.a.`
- The specification of the attribute `f` in the resulting `ClassB` is that of `S1.ClassB.f.`
- In the result, `S2.ClassC.e` has the specification of `S1.ClassB.f` with one change. Since there is already an attribute `f` in `ClassC` (which

is overridden by `ClassB.f`), renaming of attribute `f` occurs to avoid a name clash. Attribute `f` is renamed to “`S1_ClassB_f`”.

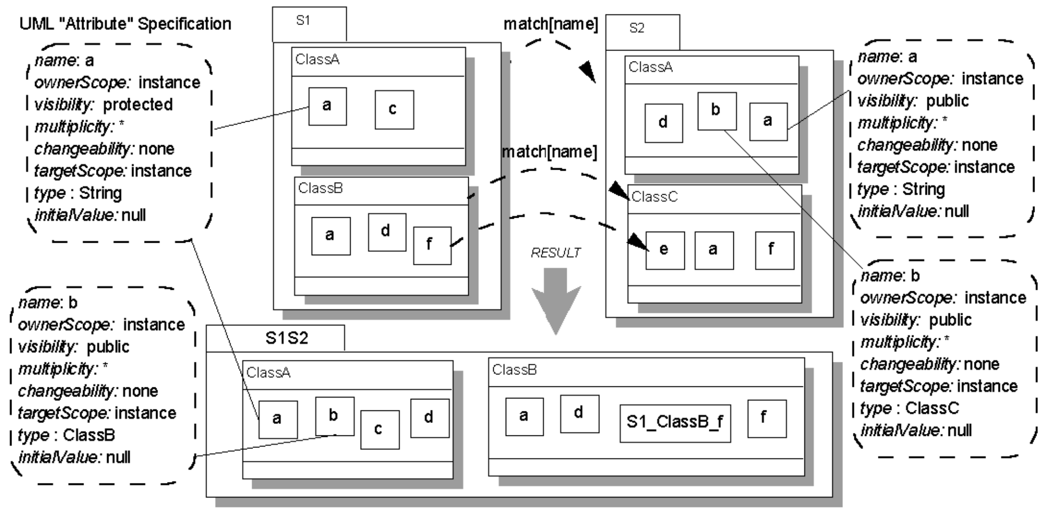


**Figure 48: Impact of Override on Attribute Specifications**

Elements with no correspondences:

- Attributes `S1.ClassA.c` and `S1.ClassB.d` have no corresponding attributes and so are added unchanged to the resulting `ClassA` and `ClassB`.
- Attribute `S2.ClassA.d` has no corresponding attribute and so is added unchanged to the resulting `ClassA`.

Elements requiring change as a result of “forwarding” semantics

- Attribute `S2.ClassA.b` has a type of `ClassC` in `S2`. However, `S2.ClassC` is overridden by `S1.ClassB` and, therefore, all references to `ClassC` in `S2` must be changed to its new specification, which is `ClassB`.

*Check on UML Well-Formed-ness Rules*

The well-formedness rules for attributes are not broken with this example.

*Differences with General Semantics for Override*

The semantics for overriding Attributes conforms to the general semantics for override.

## Impact of Override on Operations

This section discusses what happens to operation specifications as a result of override. (See “Appendix A: Partial Illustrations of UML Metamodel” on page 269 for an illustration of the UML specification of Operation) The impact of override on Operations is illustrated with an example in Figure 49.

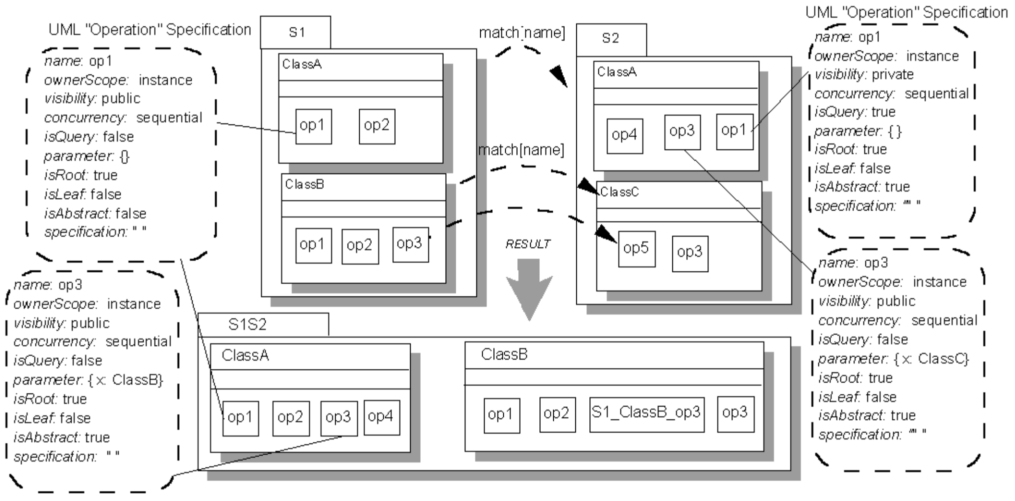


**Figure 49: Impact of Override on Operation Specifications**

*Correspondences*

- [Eg6.17] `S1` corresponds with `S2` because of the composition relationship between the two. This relationship specifies matching on name for identification of correspondence between the components
- [Eg6.18] `S1.ClassA` corresponds with `S2.ClassA` (Eg6.17)
- [Eg6.19] `S1.ClassB` corresponds with `S2.ClassC` (from the composition relationship between the two. This relationship specifies matching on name for identification of correspondence between the components)
- [Eg6.20] `S1.ClassA.op1` corresponds with `S2.ClassA.op1` (from Eg6.17)
- [Eg6.21] `S1.ClassB.op3` corresponds with `S2.ClassC.op3` (from Eg6.19)
- [Eg6.22] `S1.ClassB.op3` corresponds with `S2.ClassC.op5` (from the composition relationship between the two)

*Result of Override*

Elements with correspondences:

- The specification of `ClassA` in the resulting subject is that of the specification of `S1.ClassA`. The components of `ClassA` are considered separately.

- The specification the operation `op1` in the resulting `ClassA` is that of `S1.ClassA.op1.`
- In the result, `ClassB` has the specification of `S1.ClassC.` The components of `S1.ClassC` and `S2.ClassB` are considered separately.
- In the result, `ClassC.op3` has the specification of `S1.ClassB.op3.`
- In the result, `ClassC.op5` has the specification of `S1.ClassB.op3` with one change. Since there is already an operation `op3` in `ClassC` (which is overridden by `ClassB.op3`), renaming of operation `op3` occurs to avoid a name clash. Operation `op3` is renamed to "`S1_ClassB_op3`".

Elements with no correspondences:

- Operations `S1.ClassA.op2`, `S1.ClassB.op1` and `S1.ClassB.op2` have no corresponding operations and so are added unchanged to the resulting `ClassA` and `ClassB`.
- Operation `S2.ClassA.op4` has no corresponding operations and so are added unchanged to the resulting `ClassA`.

Elements requiring change as a result of "forwarding" semantics

- Operation `S2.ClassA.op3` has a parameter type of `ClassC` in `S2`. However, `S2.ClassC` is overridden by `S1.ClassB` and, therefore, all references to `ClassC` in `S2` must be changed to its new specification, which is `ClassB`.

*Check on UML Well-Formedness Rules*

The well-formedness rules for operations are not broken with this example.

*Differences with General Semantics for Override*

The semantics for overriding Operations must also take into consideration the impact on collaborations. See " Impact of Override on Collaborations" on page 148 for more details.

## Impact of Override on Associations and Generalizations

This section discusses what happens to the association and generalization specifications as a result of override. (See "Appendix A: Partial Illustrations of UML Metamodel" on page 269 for an illustration of the UML specification of Relationship). The impact of override on Associations and Generalizations is illustrated with a series of examples. In this section, for brevity, only the correspondences particular to associations are considered in detail.

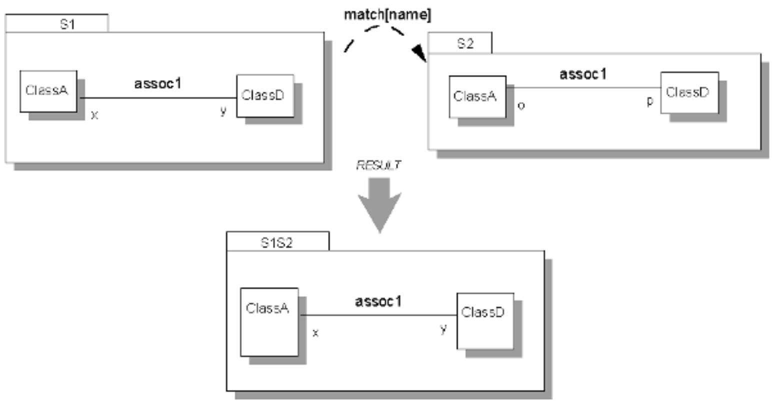


**Figure 50: Example 1: Impact of Override on Associations**

*Result of Override for Example 1 Figure 50*

Associations are manifested in code as attributes of a class, so the first example, in Figure 50 illustrates how the semantics for overriding are similar to attributes.

- `S1.assoc1` and `S2.assoc1` correspond because of the match-by-name composition relationship between `S1` and `S2`. The specification of `S2.assoc1` is changed to that of `S1.assoc1` in the result.

*Result of Override for Example 2 Figure 51*

As with all elements, associations with no corresponding associations are added unchanged to the result (see Figure 51).

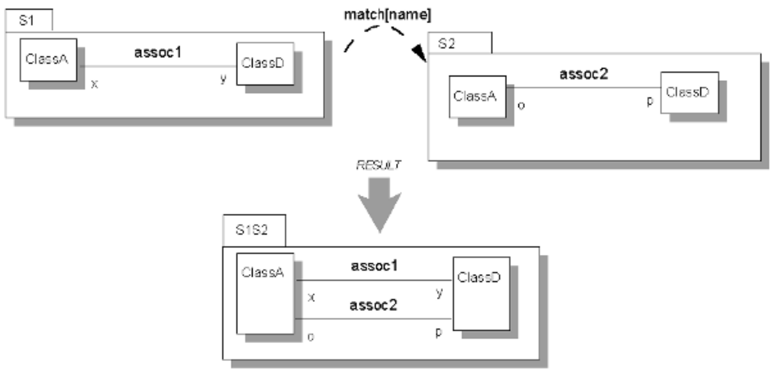


**Figure 51: Example 2: Impact of Override on Associations**

*Result of Override for Example 3 Figure 52*

One exception to the general semantics for associations is that associations may override other associations even if the classifiers that are the types of the association ends are not corresponding, without changing the association end type classifiers of the overridden association (see Figure 52).

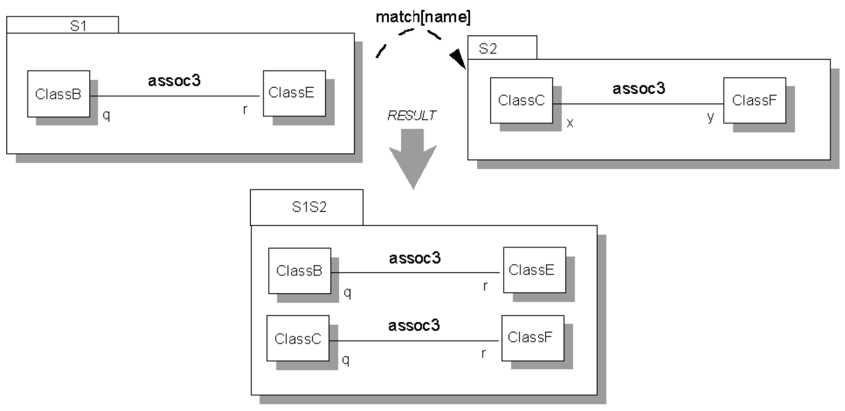


**Figure 52: Example 3: Impact of Override on Associations**

- `S1.assoc3` and `S2.assoc3` correspond because of the match-by-name composition relationship between `S1` and `S2`. The specification of `S2.assoc3` is changed to that of `S1.assoc3` in the result. The types of the classifiers of the association ends are excluded from the full specification for override, and remains the same as `S2.assoc3`.
- `S1.assoc3`, the association between `S1.ClassB` and `S1.ClassE` is added unchanged to the result.

*Result of Override for Example 4 Figure 53*

Associations may also be overridden using an explicit override (see Figure 53).

- `S1.assoc3` and `S2.assoc4` correspond because of the override between the two. The specification of `S2.assoc4` is changed to that of `S1.assoc3` in the result. The types of the classifiers of the association ends are excluded from the full specification for override, and remains the same as `S2.assoc4`.
- `S1.assoc3`, the association between `S1.ClassB` and `S1.ClassE` is also added unchanged to the result.

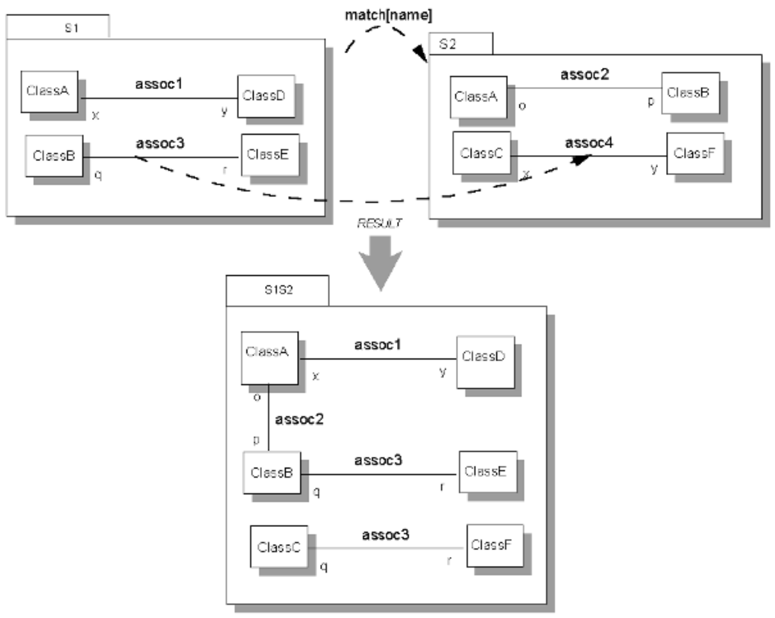


**Figure 53: Example 4: Result of Override on Associations**

*Matching Un-named Associations*

Associations without names are commonly used within UML design models. The UML semantics ([UML 1999] page 2-21) has the following description of an association's name:

> "The name of an association which, in combination with its Classifiers, must be unique within the enclosing namespace (usually a Package)."

This implies that there may be only one association without a name between the same set of classifiers, but that there may be many associations without a name between different sets of classifiers. Associations with no name present a dilemma for the subject-oriented design model. Conceptually, it is unlikely that un-named associations between different classifiers are corresponding, even if they "match" based on a match by name attachment. Therefore, it is tempting to make an exception for associations without a name, and exclude them from name-match checking for correspondence. On the other hand, more than one association without a name between the same set of classifiers appears to contradict the uniqueness description of association names in the UML.

To cope with both, the subject-oriented design model makes the correspondence general matching by name exception for associations with no name, *except* for (some) associations between the same classifier sets. In other words, associations with no name between different classifier sets *do not* correspond. As for associations with no name between the same classifier sets, consideration is taken in conjunction with the specification of its AssociationEnds. As defined by the UML, the "bulk of the structure of an Association is defined by its AssociationEnds" ([UML 1999] page 2-21, `connection` association). Association ends also have names, which are described in [UML 1999], page 2-23 as:

> "The rolename of the end. When placed on a target end, provides a name for traversing from a source instance across the association to the target instance or set of target instances. It represents a pseudo-attribute of the source classifer (i.e., it may be used in the same way as an Attribute) and must be unique with respect to Attributes and other pseudo-attributes of the source classifier."

This definition suggests that consideration of the correspondence of associations without names should be in conjunction with the names of the association ends. Therefore, associations between the same set of classifiers are considered to be corresponding if all of their association end names are the same. Otherwise, the associations are deemed to be non-corresponding.

*Generalizations* A generalization is a relationship between a more general element and a more specific element. A generalization is not a composable element, but this section considers the impact of override on generalizations. All generalizations in the scope of an override are added to the result. As illustrated in Figure 54, this may result in a multiple inheritance graph, where single inheritance was specified in the overriding and overridden subjects.

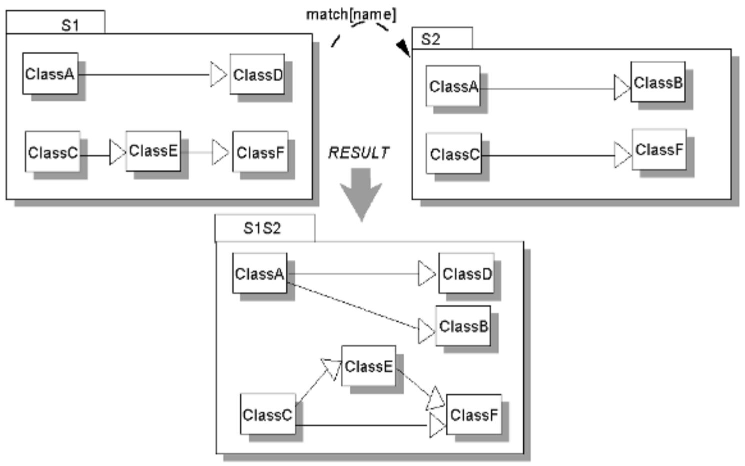


**Figure 54: Example 1: Impact of Override on Generalizations**

In Figure 54, the resulting `ClassC` is generalised from `ClassF` through two routes – directly, and from `ClassE`. This does not break the well-formedness rules as defined by the UML, but may not be the desired semantics. As with all design effort using generalizations, care should be taken with override to ensure that the result is as desired.

*UML Well-Formedness Rules*

Override integration may result in breakages to the well-formedness rules for generalizations. In "Impact of Override on Classifiers" on page 134, one example was illustrated relating to the specification of root classes. Another example is illustrated in Figure 55 and relates to the well-formedness rule "Circular inheritance is not allowed" (See UML Semantics Guide in [UML 1999] page 2-53, GeneralizableElement, Rule [3]).

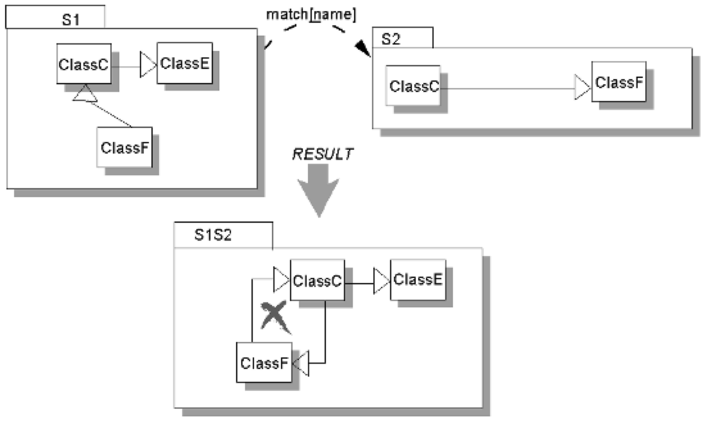


**Figure 55: Example 2: Impact of Override on Generalizations**

There has been some work in the area of eliminating cycles in composed hierarchies which could be incorporated here. In [Walker 2000], there is a proposal to eliminate cycles based on separating the type hierarchy from the implementation hierarchy in the input subjects. Generalizations are maintained in the type hierarchy, but only the implementation classes are deemed to correspond for the purposes of integration. In this way, cycles are not created in the composed implementation classes. Further investigation into the inclusion of such an approach is added to future work.

*Differences with General Semantics for Override*

- The type classifiers of association ends are not included in the full specification for override. This means that the result of overriding classifiers is that for every AssociationEnd `ae` where `ae.type` = overridden classifier, this is changed to be the overriding classifier.
- The semantics for overriding Associations must also take into consideration the impact on role specifications for collaborations. See " Impact of Override on Collaborations" on page 148 for more details.

## Impact of Override on Dependencies

This section discusses what happens to dependency specifications as a result of override. (See “Appendix A: Partial Illustrations of UML Metamodel” on page 269 for an illustration of the UML specification of Dependency). The impact of override on Dependencies is illustrated with an example. In this section, for brevity, only the correspondences particular to dependencies are considered in detail.

A dependency is a “using” relationship, which states that the implementation or functioning of one or more elements requires the presence of one or more elements. Dependency is not a composable element, but this section considers the impact of override on dependencies.

As illustrated in Figure 56, all dependencies in the scope of an override are added to the result.

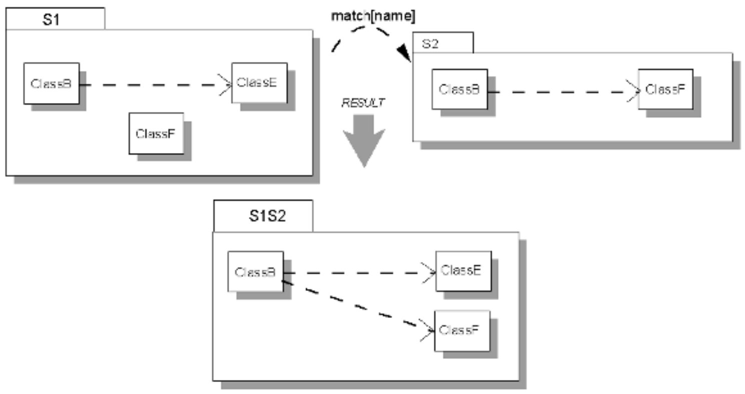


**Figure 56: Impact of Override on Dependencies**

*Result of Override:*

- Dependency between `S1.ClassB` and `S1.ClassE` added to result
- Dependency between `S2.ClassB` and `S2.ClassF` added to result – dependency will be from overridden `ClassB` to overridden `ClassF` (from match-by-name override between `S1` and `S2`).

*Check on UML Well-Formedness Rules*

The UML defines no well-formedness rules for Dependency.

## Impact of Override on Constraints

This section discusses what happens to constraint specifications as a result of override. (See “Appendix A: Partial Illustrations of UML Metamodel” on page 269 for an illustration of the UML specification of Constraint). The impact of override on Constraints is illustrated with a series of examples. In this section, for brevity, only the correspondences particular to constraints are considered in detail.

A constraint is a boolean expression on an associated element, which must be true for the model to be well formed. Some constraints are predefined in the UML, others may be user defined. All constraints are included in the rule for override, which states that the resulting model must be well-formed. Constraint is not a composable element, but this section considers the impact of override on constraints.

*Result of Override in Figure 57*

As illustrated in Figure 57, all constraints in the scope of an override are added to the result.

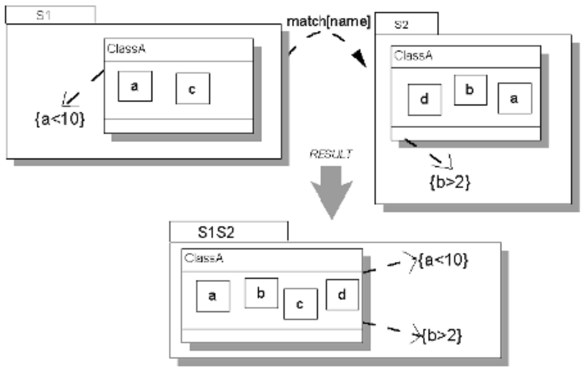


**Figure 57: Example 1: Impact of Override on Constraints**

- Constraints on attributes `S1.ClassA.a` and `S2.ClassA.b` added to result

*Result of Override in Figure 58*

As with the direct writing of constraints on a model, care should be taken to ensure the constraints in the result of an override remain as intended. Adding constraints in this manner may result in unanticipated or conflicting implications.

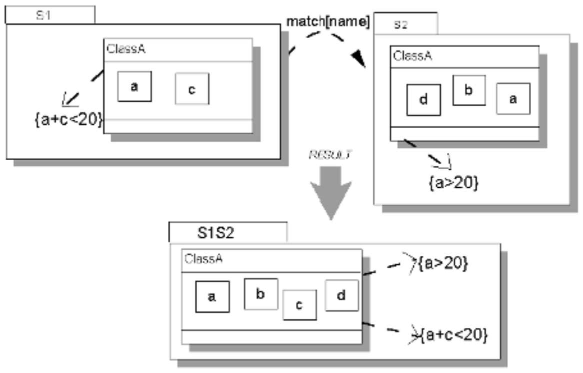


**Figure 58: Example 2: Impact of Override on Constraints**

For example, in Figure 58, constraints on `ClassA.a` imply that `ClassA.c` must always be negative.

- Constraints on attributes `S1.ClassA.a+S1.ClassA.c` and `S2.ClassA.a` are added to result.

*Result of Override in Figure 59*

Constraints on relationships behave in the standard way during overriding. Relationships that are overridden also have their constraints overridden (see Figure 59).

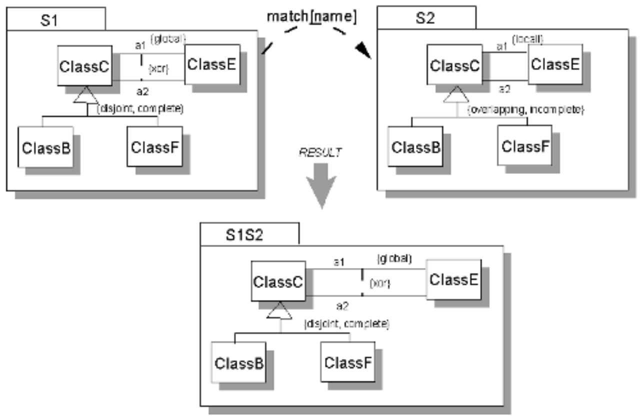


**Figure 59: Example 3: Impact of Override on Constraints**

*Check on UML Well-Formedness Rules*

Constraints are included in the well-formedness specification of a model.

## Impact of Override on Collaborations

This section discusses what happens to collaboration specifications as a result of override. (See “Appendix A: Partial Illustrations of UML Metamodel” on page 269 for a partial illustration of the UML specification of Collaboration). The impact of override on Collaborations is illustrated with a series of examples. In this section, for brevity, only the correspondences related to collaborations are considered in detail.

A collaboration specifies how objects interact with each other to complete a particular task. Through a series of messages specifying the communication between the objects, actions are activated (which result in the activation of operations) to complete the collaboration. According to the UML semantics, collaborations may be presented at two different levels – the specification level or the instance level. This thesis considers collaborations at only the specification level.

Collaborations are named model elements within the model, and represent either a single operation or a single classifier. Operations and classifiers may have several collaborations defined. A single collaboration may have multiple interactions defined, which are themselves named model elements. Collaborations are therefore composites (as defined for override), and interactions are primitives. As with all composable elements, collaborations and interactions are overridden with corresponding collaborations and interactions.

*Result of Override in Figure 60*

Figure 60 illustrates an example of the impact of override on collaborations where corresponding operations do not have corresponding collaborations.

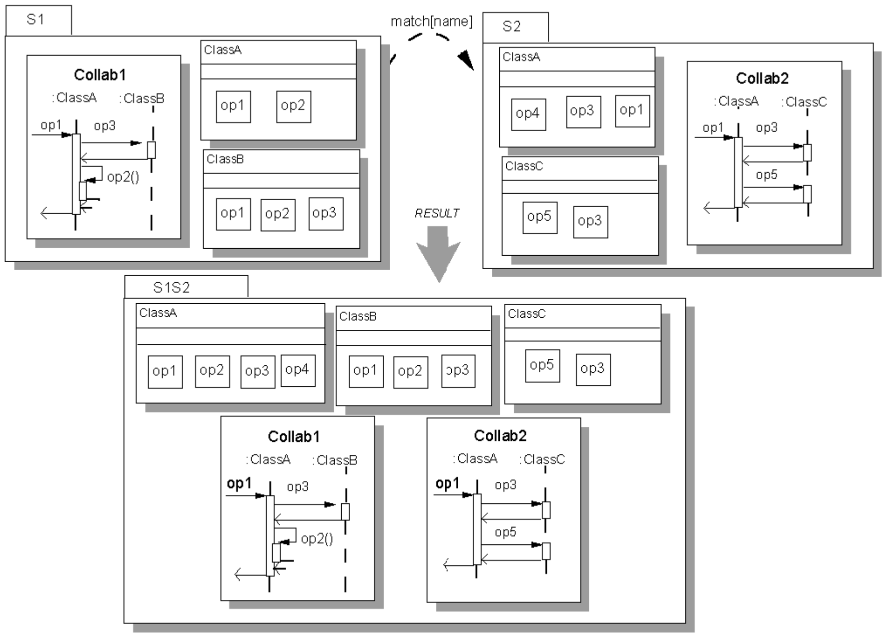


**Figure 60: Example 1: Impact of Override on Collaborations**

- Operation `S1.ClassA.op1` overrides `S2.ClassA.op1`. The specification of `S1.ClassA.op1` is added to the result.
- Collaboration `S1.Collab1` (giving a definition of a collaboration for `S1.ClassA.op1`) has no corresponding collaboration in `S2`. `S1.Collab1` is added to the result.
- Collaboration `S2.Collab2` (giving a definition of a collaboration for `S1.ClassA.op1`) has no corresponding collaboration in `S1`. `S2.Collab2` is added to the result.

Given that the collaborations are named differently, they are not deemed to correspond. However, the result is ambiguous as to the correct collaboration for `op1`, and so the designer needs to assess what to do. One approach based on using an additional composition relationship is described in Figure 62.

*Result of Override in Figure 61*

Figure 61 illustrates an example of the impact of override on collaborations where corresponding collaborations exist.

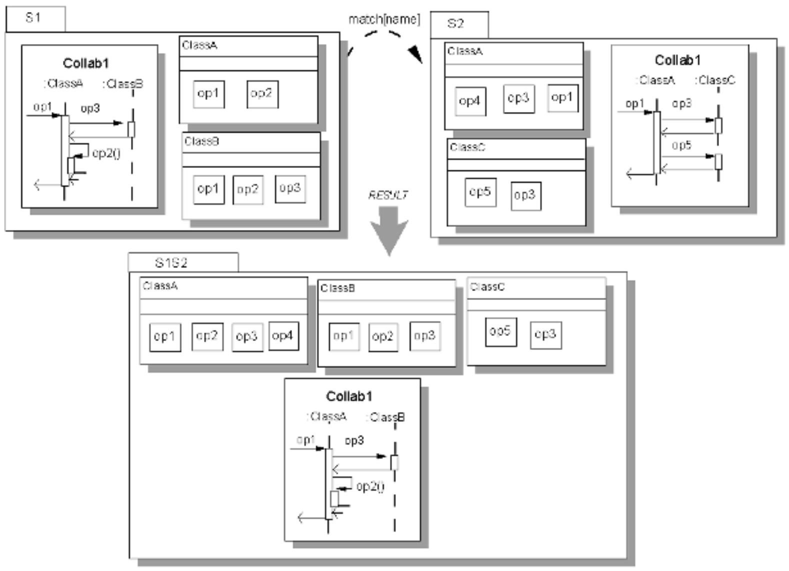


**Figure 61: Example 2: Impact of Override on Collaborations**

- Operation `S1.ClassA.op1` overrides `S2.ClassA.op1`. The specification of `S1.ClassA.op1` is added to the result.
- Collaboration `S1.Collab1` overrides `S2.Collab1`. `S1.Collab1` is added to the result.

*Result of Override Figure 62*

Figure 62 illustrates an example of the impact of override on collaborations with overrides specified between them. This approach solves the ambiguity difficulty in Figure 60.

- Operation `S1.ClassA.op1` overrides `S2.ClassA.op1`. The specification of `S1.ClassA.op1` is added to the result.
- Collaboration `S1.Collab1` overrides `S2.Collab2` because of the override between the two. `S1.Collab1` is added to the result.

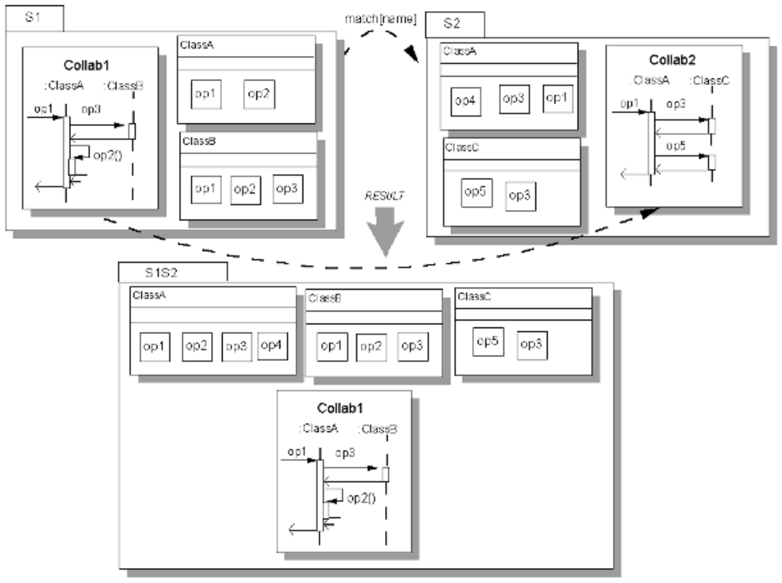


Figure 62: Example 3: Impact of Override on Collaborations

*Result of Override in Figure 63*

Operations are invoked as a result of the messages that are defined in collaborations. If an operation invoked on receipt of a particular message is overridden, and its signature is changed in any way, the operation invoked on receipt of the same message is also changed.

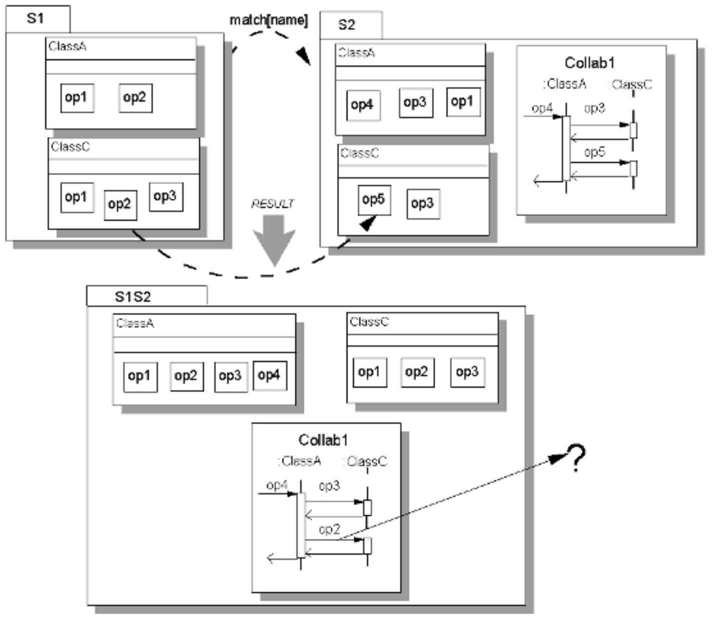


Figure 63: Example 4: Impact of Override on Collaborations

In Figure 63, operation `S2.ClassC.op5` is overridden by `S1.ClassC.op2` as illustrated. There is the possibility that overriding operations will have an impact on collaborations. Figure 63 illustrates this possibility and highlights the potential problem with a “?”. The following supporting text answers the implied question by describing the result of override.

- Operation `S1.ClassC.op2` overrides `S2.ClassC.op5`. The specification of `S1.ClassC.op2` is added to the result. `S2.ClassC.op5` has been overridden and does not appear in the result.
- Collaboration `S2.Collab1` has no corresponding collaboration and so is added to the result.
- Each collaboration in `S1` is examined so that every interaction `i` in every collaboration `c`, where `c.i.message.action.operation = S2.ClassC.op5`, is changed so that `c.i.message.action.operation = S1.ClassC.op2`.

The approach to changing references to `S2.ClassC.op5` to `S1.ClassC.op2` is in keeping with standard forwarding semantics. However, the question remains: what is to be done with the message? There are two options as to the approach to take for `c.i.message`. First, the message could remain unchanged, and this approach would be in keeping with the clear separation of message and operation in the metamodel. The operation has been overridden, which need not have any impact on the message. However, while this approach is true to the UML metamodel (and indeed, the object-oriented paradigm), it is not in keeping with standard usage of the language. “Standard usage” may be safely assumed here as even the UML notation does not define a mechanism to distinguish between message and operation in interaction diagrams. Therefore, in order to take this approach, a new notation would need to be invented to support the separation. While this would not, in itself, be a problem, there is the disadvantage of going against standard usage of the UML as defined by the UML notation. This has associated difficulties in comprehension for designers used to using interactions in the UML in the way they are currently defined. Furthermore, the distinction is not carried through to object-oriented programming models such as C++ or Java. Therefore, override semantics takes a second approach. In addition to forwarding the appropriate operation name change, the corresponding message is also updated to reflect the change to the operation. This result, therefore, answers the question in the illustration - the operation related to the call action of the message is overridden, and the message changed correspondingly.

*Collaboration Roles*

Collaborations also provide a context for participants playing different *roles* within the collaborations. See "Appendix A: Partial Illustrations of UML Metamodel" on page 269 for a partial illustration of the UML specification of Collaboration that shows the metaclasses that represent roles for associations and classifiers. These roles are in the context of sending and receiving messages

When classifiers are overridden, related collaborations for that classifier will now define their roles for the overriding classifier. Each collaboration is examined so that:

every interaction `i` in every collaboration `c`

- where `c.i.message.sender.base` = overridden classifier, this is changed so that it now refers to the overriding classifier
- where `c.i.message.receiver.base` = overridden classifier, this is changed so that it now refers to the overriding classifier
- where `c.ownedElement.base` = overridden classifier, this is changed so that it now refers to the overriding classifier

When associations (with association ends) are overridden, related collaborations for that association will now define their roles for the overriding association. Each collaboration is examined so that:

every interaction `i` in every collaboration `c`

- where `c.i.message.communicationConnection.base` = overridden association, this is changed so that it now refers to the overriding association
- where `c.i.message.communicationConnection.base` = overridden association end, this is changed so that it now refers to the overriding association end
- where `c.ownedElement.base` = overridden association, this is changed so that it now refers to the overriding association

## 6.4. Chapter Summary

This chapter defines the syntax and semantics of composition relationships with override integration. Changes to the UML metamodel to support the syntax are illustrated as an extension to the composition relationship metamodel as described in "Composition Relationship" on page 113. Well-formedness rules for composition relationships with override integration are given. These rules primarily restrict the cardinalities of composition relationships between composable elements, imposing a rule which ensures that override integration is the overriding of one composable element with one other. Other than the rules explicitly replaced in this chapter for composition relationships with override integration, all rules for general composition relationships, defined in “Well-Formedness Rules” on page 117, apply for the relationships with override integration.

The semantics for override integration is defined by illustrating the impact of overriding on each of the design elements currently supported in the thesis. First, general semantics for overriding are defined, which are, in summary, that the specifications of elements are replaced by corresponding, overriding elements, and any elements without corresponding elements are added unchanged to the result. However, some of the different kinds of design elements are treated slightly differently in some cases. In order to fully define the semantics, the impact of override on each construct is examined, with any change from the general semantics highlighted as appropriate.

The next chapter details the semantics of the second integration strategy described in this thesis -- merge integration.

# Chapter 7: Merge Integration

Merge integration is used when separate design models (subjects) contain specifications for different requirements of a computer system. This may have occurred for several reasons. For example, within a system development effort, separate design teams may have worked on different requirements concurrently. In this case, merge is especially useful where a requirement has an impact across the whole design – for example a requirement stipulating that objects reside in a distributed environment is likely to affect all objects. Distribution behaviour may be designed separately and merged with the rest as required. Another use of merge integration is the case where designs may exist for requirements from a previous version of the system. These requirements are still appropriate for the system, and therefore need to be merged with new requirements. Also, designs may be reused from sources outside the current development effort. The full system design is obtained by merging the designs of the separate design subjects.

Composition relationships, with merge integration, are the means to specify how subjects should be merged. Composition relationships identify the subjects to be merged, and the design elements within those subjects that specify the same concept (i.e. correspond to each other) and should be considered as one. 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 need to be resolved, composition relationships with merge integration specify guidelines for the reconciliation.

Merging operations essentially means joining behaviours, and so, with merged operations, the receipt of a message that may have activated one of the operations in an input subject now results in the execution of all of the merged operations. Collaborations may be attached to a composition relationship with merge integration to determine the order of execution.

This chapter is divided up into three sections:

- *Description:* This section gives a general overview of merge integration, introducing each of the different concerns.
- *Metamodel Extensions:* This section defines the extensions required to the composition metamodel to support merge integration.
- *Semantics:* This section gives details of the semantics of merge integration in terms of its impact on the supported UML constructs.

## 7.1. Description

Composition relationships with merge integration may be specified between subjects, between model elements that are owned or referenced by a subject, and, in general, between model elements that are owned or referenced by those elements – for example, classifiers owns operations between which composition relationships may be specified. The kinds of elements between which it makes sense to specify composition relationships are listed in the rules. The relationship may only be specified between elements of the same type – for example, a classifier with a classifier, a subject with a subject, etc. For brevity, merge integration will hereafter be referred to as "merge".

**Merge as a Simple Union**

At the simplest level, where there are no corresponding elements in the subjects, merge results in the merged subject containing all the design elements of both subjects. For example, in Figure 64, `S1` has two classes, `S1.ClassA` and `S1.ClassB`. `S2` has two classes, `S2.ClassC` and `S2.ClassD`. Merging `S1` and `S2` results in a subject with four classes.

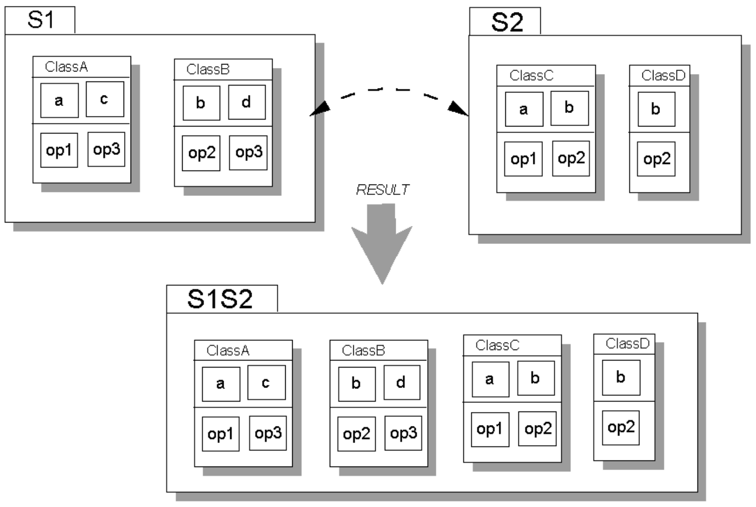


**Figure 64: Simple Merging of Subjects**

**Merge with Corresponding Classes, Attributes**

When subjects have corresponding classes and attributes, those elements appear once in the merged subject. See Figure 65 for an example, which yields the following result:

- `S1.ClassA` and `S2.ClassA` correspond from the `match[name]` composition relationship between `S1` and `S2`. Since they are corresponding, `ClassA` only appears once in the result.
- `S1.ClassA.a` and `S2.ClassA.a` correspond from the `match[name]` composition relationship between `S1` and `S2`. Since they are corresponding, `ClassA.a` only appears once in the resulting `ClassA`.
- `S1.ClassB` and `S2.ClassD` have no corresponding elements and are added unchanged to the result.

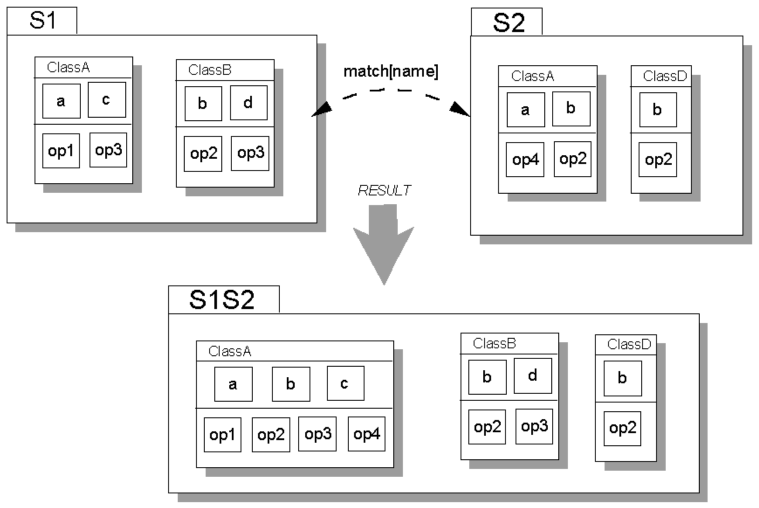


**Figure 65: Merge with Corresponding Classes and Attributes**

**Merge with Conflicts in Corresponding Elements**

Of course, merging corresponding elements like classifiers and attributes where one element appears in the result[1] is only simple when the specifications of the corresponding elements are exactly the same. Since the subjects are designed separately, there is potential for differences in the specifications of corresponding elements. Figure 66 illustrates some examples of where conflicts may exist. In the example, the elements where conflicts occur are highlighted with a "?". "Reconciling Conflicts in Corresponding Elements" on page 158 gives answers to these questions.

In this example, we have two cases where the specifications of corresponding attributes conflict.

1. This applies to all elements except operations, constraints and collaborations

- `S1.ClassA.a` and `S2.ClassA.a` correspond from the `match[name]` composition relationship between `S1` and `S2`. However, their specifications are different, and so which specification appears in the merged subject?
- `S1.ClassA.b` and `S2.ClassA.c` correspond from the merge relationship between the two. Again, their specifications are different (they have different names), and so which specification appears in the merged subject?

To resolve these questions, the different specifications must be reconciled before being added to the result of the merge.

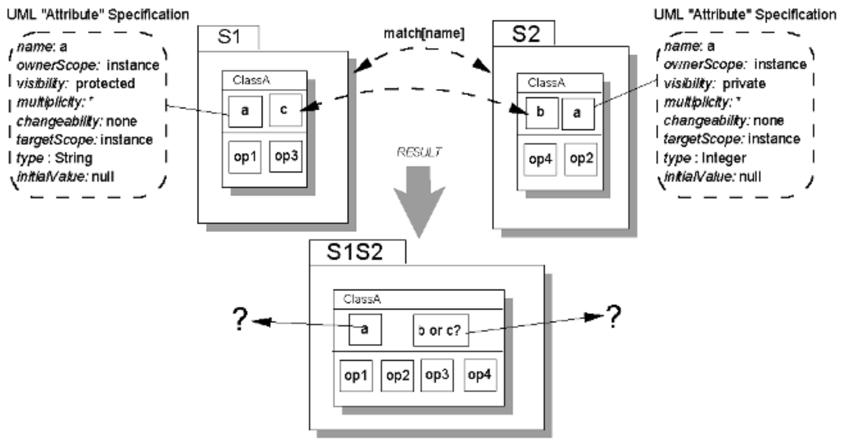


**Figure 66: Conflicts in Corresponding Elements**

## Reconciling Conflicts in Corresponding Elements

When subjects are merged, elements that are specified to support corresponding concepts are identified, and will be merged in the composed subject – that is, for most kinds of elements (except, for example, operations), they will appear once in the merged subject. However, since corresponding elements may have been specified separately, there may be differences in those specifications. These differences must be reconciled for the composed subject.

### *Assigning Precedence to a Subject in the event of a Conflict:*

One approach to reconciling conflict is to assign precedence to one of the subjects involved in the merge. When a conflict occurs, the specification of the element in the subject with precedence is deemed to be the specification for the merged element.

By adding a precedence indicator to `S1` (see Figure 67), the result of the merge is now:

- `S1.ClassA.a` and `S2.ClassA.a` correspond from the `match[name]` composition relationship between `S1` and `S2`. Since their specifications are different, and precedence has been specified for `S1` (from composition relationship between `S1` and `S2`), `S1.ClassA.a` is added to the result.
- `S1.ClassA.b` and `S2.ClassA.c` correspond from the merge relationship between the two. Again, since their specifications are different, and precedence has been specified for `S1`, `S1.ClassA.c` is added to the result.

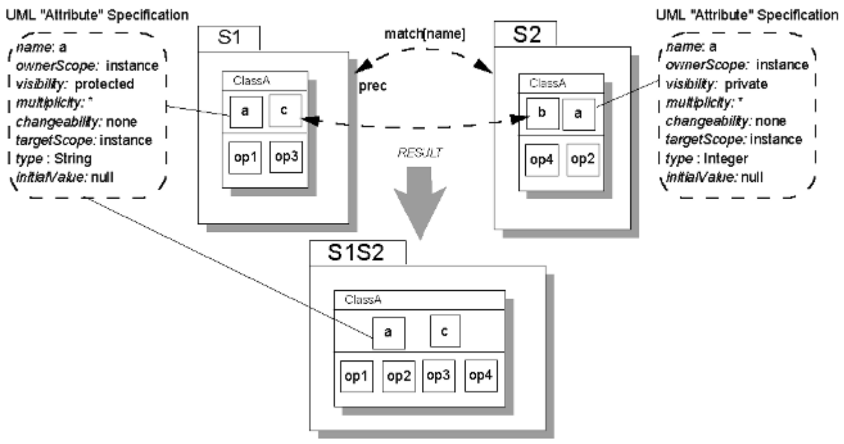


**Figure 67: Reconciliation with Subject Precedence**

*Other Reconciliation Possibilities*

It is possible to attach other kinds of reconciliation strategies to a composition relationship with merge integration. These strategies work similarly to the precedence strategy in that once a conflict is detected, the appropriate strategy determines the specification of the element that is added to the result. Other examples of reconciliation strategies are:

- Attach an explicit specification for the merged element to be used in the event of a conflict. For example, in anticipation of the conflict in attributes, a specific attribute specification may be attached to the composition relationship. An element specification attached to a composite merge is applied to a specific conflict between particular named component elements. An element specification attached to a primitive merge is applied directly to the elements related. The named component elements are assumed to correspond, either explicitly or implicitly, as defined by the composition relationship. Explicitly named components that do *not* correspond as defined by the composition relationship are ignored - that is, they *do not specify* additional corresponding elements. The notation for this attachment is:

`reconcil[ explicit [` {*list_of_input_elements*}, {*values*} ]]

- Attach default values for different types of constructs that should be used in the event of a conflict between corresponding elements of that type. For example, one of the properties of an attribute is “owner scope”. If one attribute specifies its owner scope as `instance` and its corresponding attribute specifies its owner scope as `classifier`, then a default specification for conflicts for attributes may reconcile this conflict as defaulting to `instance`. The notation for this attachment is:

`reconcil[ default [` *construct_name,* {*values*} ]]

- Attach a transformation function to be applied to conflicting corresponding elements to determine the specification for the merged element. This specification of such a transformation function is the responsibility of the designer specifying merge, and should result in a valid element specification. The notation for this attachment is:

`reconcil[ transform [` {*list_of_input_elements*}, *program_name* ]]

*Reconciliation Semantics - General*

A designer attaches reconciliation strategies to a composition relationship, and indicates the order in which each of the attached strategies should be examined. When the integration process encounters a conflict between corresponding elements that requires a reconciliation, each of the reconciliation strategies attached to the composition relationship that specifies those corresponding elements is examined, in order, to find the appropriate reconciliation. However, if the attached reconciliation strategies (or indeed, if there has been none attached) do not result in a reconciled element, then **each of the corresponding elements is added to the output separately**. Elements are renamed to avoid a name clash.

### Merge with Corresponding Operations

Merging operations means joining behaviours and so, or operation elements, merge means that on receipt of any message that resulted in the execution of an operation in an input subject, all corresponding operations are now executed. This means that all corresponding operations are added to the result. This section introduces:

- How a collaboration is generated as a result of a merge, to specify that all corresponding operations are executed on receipt of an appropriate message. In this case, the order of execution is not important, and so the designer need not specify the order by attaching a collaboration.

- How a collaboration may be attached to a composition relationship with merge integration to specify an order of execution for corresponding operations.

*Composition relationship with No Attached Collaboration*

Where no collaboration is attached to a composition relationship with merge integration, the behaviour of the output subject in relation to the merged operations is automatically specified with a new collaboration specification (see Figure 68). This collaboration specifies that an invocation of one of the corresponding operations results in the invocation of all corresponding operations. In this case, it is assumed that the order of execution is not important. In addition, where new collaborations are automatically specified as described here, each of the corresponding operations must have the same argument list. For options relaxing this restriction, see “Merging Operations with Attached Collaborations” on page 191.

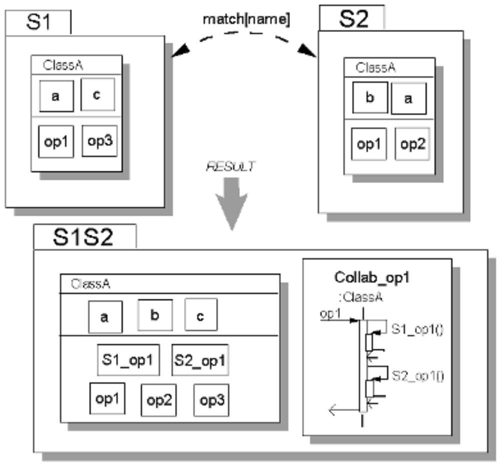


**Figure 68: Merging Corresponding Operations**

In this example, the result of the merge is:

- `S1.ClassA` and `S2.ClassA` correspond from the `match[name]` composition relationship between `S1` and `S2`. No conflict exists between the specifications, and so `ClassA` is added to the result.

- `S1.ClassA.a` and `S2.ClassA.a` correspond from the `match[name]` composition relationship between `S1` and `S2`. No conflict exists between the specifications, and so `ClassA.a` is added to the result.

- `S1.ClassA.op1` and `S2.ClassA.op1` correspond from the `match[name]` composition relationship between `S1` and `S2`. After renaming to avoid a name clash, both operations are added to the result. A new collaboration is created and added to the result indicating that on receipt of an `op1` message, both `S1.op1` and `S2.op1` are executed.

*New Operations created to capture merged collaborative behaviour*

The approach to capturing the behaviour of merged operations is based on renaming corresponding operations from the input subjects, and creating new operations with the same name as those in the input subjects. These new operations may be used to create collaborations that define the execution of all the corresponding (now renamed) operations, without any ambiguity. The ambiguity avoided with this approach is one which would cause an infinite loop. For example, the specification of a collaboration for `op1` that specifies that `op1` is one of a number of operations executed is the specification of an infinite loop.

A different approach is possible based on the clear separation of message and operation in the UML metamodel. Using this separation, collaborations could be defined specifying that on receipt of a particular *message*, all the corresponding *operations* would execute. However, while this separation is explicitly defined in the UML metamodel, the UML notation does not support the specification of messages on collaborations. This problem could be solved by inventing a notation to support messages, which would mean that additional operations would not have to be added to the composed class (as in Figure 68), and a solution could be defined that is "pure" in relation to the object-oriented paradigm. However, it goes against standard usage of the UML, and therefore has corresponding difficulties relating to how designers expect to use, and their general understanding of, interaction diagrams. It is therefore decided to use the approach illustrated in Figure 68 (and subsequent examples of merging operations) as it uses the standard UML language.

The approach taken based on creating new operations to define the delegation behaviour is open to some refinement using forwarding semantics. This is described in "Merged Operations and Forwarding of References" on page 195.

*Attaching a Collaboration to a composition relationship*

When the order of execution of corresponding operations is important, a collaboration specifying this order should be attached to the composition relationship. In this case, the attached collaboration is added to the merged subject as the specification of the behaviour of corresponding operations (see

Figure 69). All operations in the corresponding operation set must be included in the collaboration. For options relating to operations with different argument lists, see “Merging Operations with Attached Collaborations” on page 191.

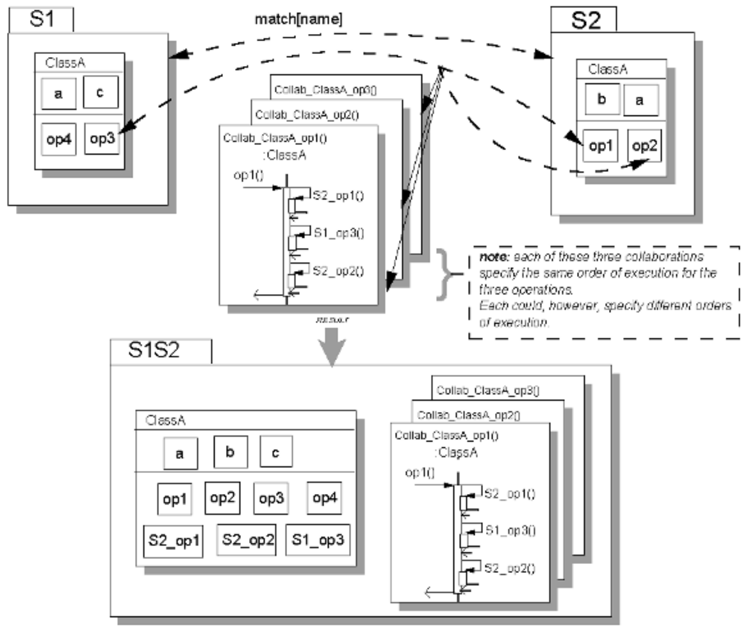

S1
match[name]
S2
ClassA
a
c
op4
op3
b
op1
op2
Collab_ClassA_op3()
Collab_ClassA_op2()
Collab_ClassA_op1()
:ClassA
op1()
S2_op1()
S1_op3()
S2_op2()
note: each of these three collaborations specify the same order of execution for the three operations. Each could, however, specify different orders of execution.
S1S2
S2_op1
S2_op2
S1_op3

Figure 69: Attaching Collaborations to Composition Relationship

In this example, the result of the merge is:

- `S1.ClassA` and `S2.ClassA` correspond from the `match[name]` composition relationship between `S1` and `S2`. No conflict exists between the specifications, and so `ClassA` is added to the result.
- `S1.ClassA.a` and `S2.ClassA.a` correspond from the `match[name]` composition relationship between `S1` and `S2`. No conflict exists between the specifications, and so `ClassA.a` is added to the result. `S1.ClassA.c` and `S2.ClassA.b` have no corresponding attributes and so are added to the result.
- `S1.ClassA.op3`, `S2.ClassA.op1` and `S2.ClassA.op2` correspond from the composition relationship between them. All the operations are added to the result, and renamed to avoid ambiguity with operations added (`op1`, `op2` and `op3`) to support the specification of the merged behaviour. The collaborations attached to the composition relationship are added to the result indicating that on execution of `op1`, `op2` or `op3`, `S2_op1` followed by `S1_op3` followed by `S2_op2` are executed.

- `S1.ClassA.op4` has no corresponding operations and is therefore simply added to the result.

The remainder of this chapter discusses the semantics of merge for design models. Using the UML metamodeling style, the section has the following subsections:

- A subsection with UML class diagrams describing the constructs of merge, and their relationships.
- A subsection containing the well-formedness rules describing the constraints on instances of merge.
- A subsection containing descriptions of the semantics of merge.

## 7.2. Merge Integration Syntax

This section describes merge integration using UML class diagrams to represent the metaclasses relevant for its description, and their relationships. The class diagram includes metaclasses from the UML metamodel with which composition relationships interact, and new metaclasses representing merge integration itself. The description of the constructs in the metamodel does not include descriptions of those constructs that are already described in the UML semantics.

A composition relationship with merge integration specifies design elements that are to be merged. For some design elements (e.g. classifiers, attributes), merging corresponding elements means one of the elements is copied to the result. A composition relationship may attach reconciliation specifications for possible conflicts between such corresponding elements. For operations, constraints and collaborations, all corresponding elements are added to the result. A composition relationship may attach a collaboration to specify the order of execution of corresponding operations. To handle each of these situations, the syntax of a composition relationship has the following parts:

- Identification of corresponding elements for composition relationships. This is described in “5.3. Composition Relationship” on page 113 and applies to composition relationships with merge integration.
- The basic composition relationship with merge integration, as described in “Merge Integration” on page 165
- The syntax associated with attaching reconciliation specifications to a composition relationship with merge integration, as described in “Reconciliation of Conflicts” on page 165.

- The syntax associated with attaching collaborations to specify the order of operation execution, as described in " Collaborations for Merged Operations" on page 167

**Merge Integration**

Figure 70 describes merge integration as a subclass of the Integration metaclass described in "5.3. Composition Relationship" on page 113.

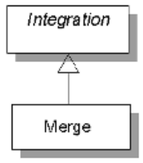


**Figure 70: Merge Integration**

*Merge Metaclass*

Merge integration specifies that corresponding elements are merged. The semantics of merge integration depends on the kind of elements being merged.

**Reconciliation of Conflicts**

For some design elements (e.g. classifiers, attributes), merging corresponding elements means one of the elements is copied to the result. Merge integration specifications may attach reconciliation specifications for possible conflicts between such corresponding elements (Figure 71).

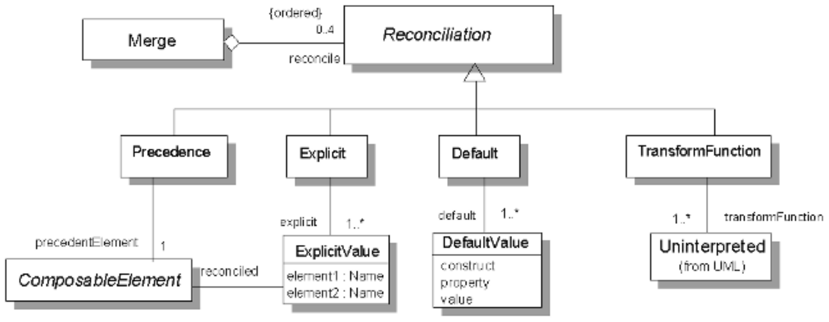


**Figure 71: Reconciliation Specification**

*Merge Metaclass*

An additional property to support reconciliation is its association with Reconciliation.

***Associations***

| | |
|---|---|
| *reconcile* | The reconcile association is an ordered association with reconciliation strategies. The ordering defines the order in which reconciliation strategies are used to reconcile conflicts between elements. The order defined as a default is: 1) Explicit 2) TransformFunction 3) Precedence 4) Default. This order is customisable. |

*Reconciliation Metaclass*

Reconciliation specifies the manner in which conflicts between the specifications of corresponding elements should be reconciled. There are four kinds of reconciliation supported: Precedence, Explicit, Default and TransformFunction.

Reconciliation is an abstract metaclass.

*Precedence Metaclass*

Precedence reconciliation specifies a composable element whose values take precedence in the event of a conflict between specifications of corresponding elements.

***Associations***

| | |
|---|---|
| *precedentElement* | The element that should take precedence in the event of a conflict. This is generally specified as a subject, but may be any element participating in the relationship. |

*Explicit Metaclass*

An explicit reconciliation provides the specification that is to be used in the composed subject instead of the specifications of particular corresponding elements that are participating in the merge composition.

***Associations***

| | |
|---|---|
| *explicit* | The element contains the references to the named elements for which an explicit specification is required, and an associated specification of the explicit values. |

*ExplicitValue Metaclass*

An explicit value contains the names of the corresponding elements for which an explicit specification is specified, and defines the explicit values using a reference to the element to be used in the composed result. The named component elements are assumed to correspond, either explicitly or implicitly, as defined by the composition relationship. Explicitly named components that do *not* correspond as defined by the composition relationship are ignored - that is, they *do not specify* additional corresponding elements.

***Attributes***

| | |
|---|---|
| *element1* | The name of one of the corresponding elements |
| *element2* | The name of another of the corresponding elements |

***Associations***

| | |
|---|---|
| *reconciled* | The specification that is to be used in the composed subject instead of the corresponding elements' specifications. |

*Default Metaclass*

Default reconciliation specifies the default values for elements of a particular type, and so, in the event of a conflict between elements of that type, the default values are used

***Associations***

| | |
|---|---|
| *default* | The default values for properties of composable elements. |

*DefaultValue Metaclass*

A default value contains the default value of a particular property belonging to a particular construct.

***Attributes***

| | |
|---|---|
| *construct* | The default is specified for this construct |
| *property* | The default is specified for this property of the construct |
| *value* | The default value for the property |

*TransformFunction Metaclass*

Transform function reconciliation specifies a function to be executed against conflicting corresponding elements to determine the reconciled specification.

***Associations***

| | |
|---|---|
| *transform-Function* | The function to be run to determine the reconciled specification. This makes use of the UML uninterpreted data type to refer to the reconciliation specific function. |

## Collaborations for Merged Operations

For operations, constraints and collaborations, all corresponding elements are added to the result. Merge integration specifications may attach a collaboration to specify the order of execution of corresponding operations (Figure 72).

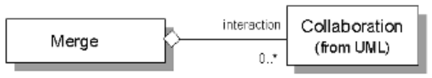


**Figure 72: Collaborations for Merged Operations**

*Merge Meta-class*

***Associations***

*interaction* A collaboration that specifies the order of execution of operations related by a composition relationship.

## 7.3. Well-Formedness Rules

This section lists the well-formedness rules for merge composition relationships. These rules are in addition to the rules specified for composition relationships in general in “5.3. Composition Relationship” on page 113.

*Reconciliation Specification*

[1] Reconciliations attached to a composition relationship apply to all elements *except* operations, constraints and collaborations.

[2] There can only be one of each of the kinds of reconciliation in the ordered set of reconciliations attached to a merge. For example, only one precedent element is possible. Each of the other three kinds (explicit, default and transform function) maintain their own relevant set of explicit, default and transform function specifications, respectively, but only one set of each per merge is necessary.

*Collaboration Specification for Operation Merge*

[3] All operations in a corresponding set must be referenced in any collaboration specifying the order of execution for that corresponding set (see Figure 73). Note, not all operations must be realised by a collaboration. Any operation which is not realised by a collaboration attached to the composition relationship will not exhibit collaborative behaviour. In this way, it is possible to specify that some operations result in the execution of all the corresponding operations, but not necessarily all of those operations have that effect.

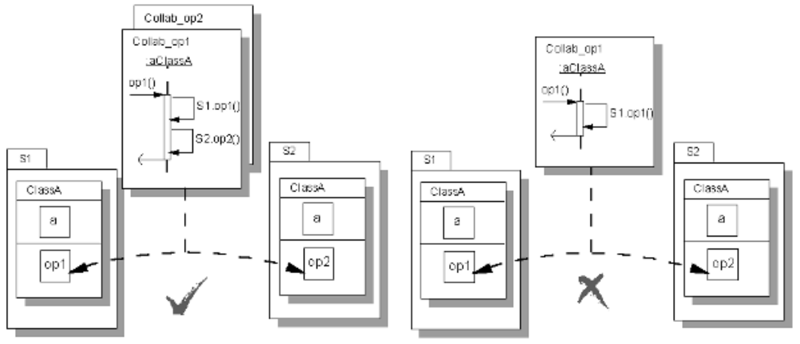


**Figure 73: All corresponding operations referenced in attached collaborations**

## 7.4. Semantics

As stated previously, merge integration is used to merge design specifications in different design subjects. Composition relationships with merge integration indicate which elements in the design subjects are corresponding, and should be considered as one element.

This section first discusses the general semantics of merge in " General Semantics" on page 169. Sections "Impact of Merge on Subjects" on page 170 to "Impact of Merge on Collaborations" on page 195 then consider the impact of merge on each of the different types of supported elements.

### General Semantics

[1] Corresponding elements are identified as described for composition relationships in "Semantics for Identifying Corresponding Elements" on page 122. These semantics apply to composition relationships with merge integration.

[2] For elements not involved in correspondence matching in different subjects, merge integration is a simple union of those elements in the composed subject.

[3] For all corresponding elements except operations, constraints and collaborations, one element representing the corresponding elements appears on the composed result.

[4] Component elements of composites may only be merged if their owning composites are corresponding and therefore, are merged.

[5] Where conflicts exist in the specifications of corresponding elements (except operations, constraints and collaborations) those conflicts are recon-

ciled based on the reconciliation option specified by the composition relationship.

[6] All corresponding operations appear on the merged result, but are merged in the sense that the specification dictates that an invocation of one of the corresponding operations results in the invocation of all corresponding operations. Where ordering is important, a collaboration may be attached to the appropriate composition relationship.

[7] All constraints are added to the result. Where only one representative element of a corresponding set of elements is added to the result, all constraints on the corresponding elements are added to the result for that representative element.

[8] Adding elements to a composed result from different source subjects may not result in name clashing. In the event of name clashes, renaming of clashing elements occurs.

[9] All references to elements in the result that may have changed from the specification in the input subject are changed as described in "Semantics for Forwarding References to Composed Elements" on page 123.

[10] The composed result must conform to the well-formedness rules of the UML.

## Impact of Merge on Subjects

This section discusses what happens to subject specifications as a result of merge (See "Appendix A: Partial Illustrations of UML Metamodel" on page 269 for an illustration of the UML specification of Package, from which Subject is stereotyped). Then, with an example, the following are illustrated:

- How correspondences are established
- The results of merge on corresponding subjects with no conflicts
- The results of merge on corresponding subjects that require specification reconciliation
- Checking the UML Well-Formedness Rules on the results of merge
- Further examples of reconciliation of conflicts in subjects.

The following subsections describe the impact of merge on the example illustrated in Figure 74

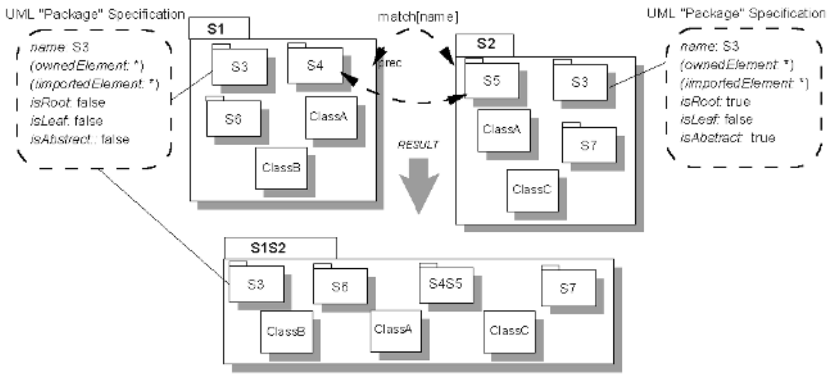


**Figure 74: Impact of Merge on Subjects**

*Correspondences:*

- [Eg7.1] `S1` corresponds with `S2` because of a composition relationship between the two. This relationship is the contextual composition relationship (see "Composition Relationship" on page 113 for details) This relationship specifies matching on name for identification of correspondence between the components
- [Eg7.2] `S1.S3` corresponds with `S2.S3` (Eg7.1)
- [Eg7.3] `S1.S4` corresponds with `S2.S5` (because of the composition relationship between the two.)
- [Eg7.4] `S1.ClassA` corresponds with `S2.ClassA` (Eg7.1)

*Result of Merge*

Elements with correspondences and no conflicts:

- With subjects, the result of the merge is to name the resulting subject by concatenating the names of the input subjects[2]. The specification of the resulting subject is therefore `S1S2` with the values of the other properties copied from one of the input subjects. Since there is no conflict, it is not important which subject's values are copied. This excludes the values for `ownedElements` and `importedElements` as these are components of subjects.
- The specification of the subject resulting from the merge of `S1.S4` and `S2.S5` is named `S4S5`. The values of the other properties are copied from one of the input subjects (since they are the same). The components of both (in `ownedElements` and `importedElements`) are considered

2. When the names of the input subjects are the same, concatenating is still performed (e.g. `S1S1`) to distinguish the result from the input subjects.

separately, with the resulting components contained in `S4S5` in the result.

- The specifications of `S1.ClassA` and `S2.ClassA` are merged in the resulting subject (see section "Impact of Merge on Classifiers" on page 173 for more details on classifiers). The components of `ClassA` are considered separately.

Elements with correspondences and conflicts in their specifications:

- The specifications of `S2.S3` and `S1.S3` are merged. The name of the resulting subject is `S3S3`. However, the values of `isRoot` and `isAbstract` are different, so a reconciliation strategy is required. The composition relationship governing this correspondence (that is, between `S1` and `S2`) indicates that `S1` has precedence in the event of a conflict. Therefore, the values of `isRoot` and `isAbstract` from `S1.S3` are copied to the result. The components of `S3` (in `ownedElements` and `importedElements`) are considered separately.

Elements with no correspondences:

- `S1.S6`, and `S1.ClassB` have no corresponding elements in `S2`. They are therefore added to the resulting subject, unchanged in any way, and without further consideration of their components.
- `S2.S7`, and `S2.ClassC` have no corresponding elements in `S1`. They are therefore added to the resulting subject, unchanged in any way, and without further consideration of their components.

*Check on UML Well-Formedness Rules*

The well-formedness rules for packages are not broken in this example.

*Other Reconciliation Possibilities*

The previous example showed how a subject can be set as the precedent subject, which means that in the event of a conflict between specifications of corresponding component elements, the values from the element in the precedent subject are copied to the result. Figure 75 illustrates the use of other reconciliation strategies.

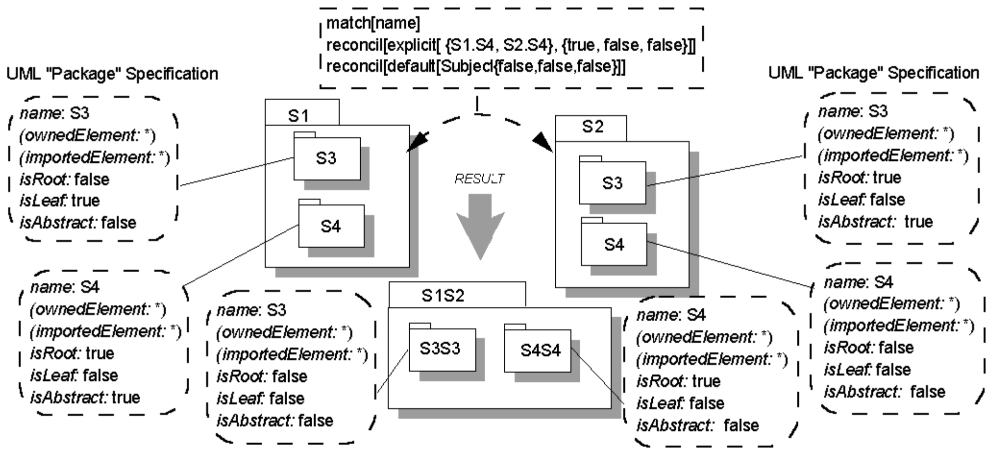


**Figure 75: Reconciling Conflicts in Subject Specifications**

Elements with correspondences and conflicts in their specifications:

- The specifications of `S2.S3` and `S1.S3` are merged. The name of the resulting subject is `S3S3`. However, the values of `isRoot`, `isLeaf` and `isAbstract` are different, so a reconciliation strategy is required. The composition relationship between `S1` and `S2` has two kinds of reconciliation strategies attached. First, a search through the explicit reconciled elements shows that there is no explicit reconciliation for `S3`. However, default values for subjects are included, and so the values of `isRoot`, `isLeaf` and `isAbstract` in the resulting subject are set to the defaults listed. The components of `S3` (in `ownedElements` and `importedElements`) are considered separately.
- The specifications of `S2.S4` and `S1.S4` are merged. The name of the resulting subject is `S4S4`. However, the values of `isRoot` and `isAbstract` are different, so a reconciliation strategy is required. The composition relationship between `S1` and `S2` has two kinds of reconciliation strategies attached. A search through the explicit reconciled elements shows that an explicit reconciliation for `S4` has been defined. Therefore values of `isRoot`, `isLeaf` and `isAbstract` in the resulting subject are set to the explicit values listed. The components of `S4` (in `ownedElements` and `importedElements`) are considered separately.

## Impact of Merge on Classifiers

This section discusses what happens to subject specifications as a result of merge (See “Appendix A: Partial Illustrations of UML Metamodel” on page

269 for an illustration of the UML specification of Classifier). The following subsections describe the impact of merge on the example illustrated in Figure 76.

*Correspon-dences*

- [Eg7.5] `S1` corresponds with `S2` because of the composition relationship between the two. This relationship specifies matching on name for identification of correspondence between the components
- [Eg7.6] `S1.ClassA` corresponds with `S2.ClassA` (Eg7.5)
- [Eg7.7] `S1.ClassB` corresponds with `S2.ClassB` (Eg7.5)
- [Eg7.8] `S1.ClassD` corresponds with `S2.ClassC` (from the relationship between the two)
- [Eg7.9] `S1.ClassD` also corresponds with `S2.ClassD` from (Eg7.5). Recall that composable elements may participate in multiple composition relationships (see "Participation in multiple composition relationships" on page 86). Any correspondence *not* required which occurs implicitly as a result of a matching specification attached to a relationship at a higher level in the subject tree must be explicitly excluded with a composition relationship with a `dontMatch` attachment.

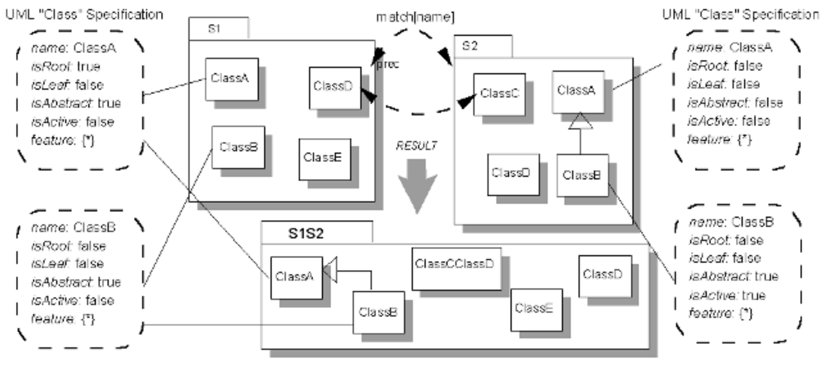


**Figure 76: Impact of Merge on Classifiers**

*Result of Merge*

Elements with correspondences and no conflicts:

- In the result, `S2.ClassC` is merged `S1.ClassD`. Since their names are different, the names are appended with the result class called "`ClassC-ClassD`". The components of `S2.ClassC` and `S1.ClassD` (in `feature`) are considered separately.
- `S1.ClassD` is merged `S2.ClassD`. Their components are considered separately.

Elements with correspondences and conflicts in their specifications:

- The specifications of `S1.ClassA` and `S2.ClassA` are merged. Since the names are the same, the name of the resulting class is `ClassA`. However, the values of `isRoot` and `isAbstract` are different, so a reconciliation strategy is required. The composition relationship between `S1` and `S2` indicates that `S1` has precedence in the event of a conflict. Since this merge applies here, the values of `isRoot` and `isAbstract` from `S1.ClassA` are copied to the result. The components of `ClassA` (in `feature`) are considered separately.
- The specifications of `S1.ClassB` and `S2.ClassB` are merged. Since the names are the same, the name of the resulting class is `ClassB`. However, the values of `isActive` are different, so a reconciliation strategy is required. The composition relationship between `S1` and `S2` indicates that `S1` has precedence in the event of a conflict. Since this relationship applies here, the value of `isActive` from `S1.ClassB` is copied to the result. The components of `ClassB` (in `feature`) are considered separately.

Elements with no correspondences:

- `S1.ClassE` has no corresponding elements in `S2`. It is therefore added to the resulting subject, unchanged in any way, and without further consideration of its components.

*Check on UML Well-Formedness Rules*

The example illustrated in Figure 76 does not result in a breakage of the well-formedness rules of the UML.

However, with a small change as illustrated in Figure 77, it is easy to see where a breakage might occur. The illustration highlights (with a big X) where a breakage of the well-formedness rules of the UML may occur.

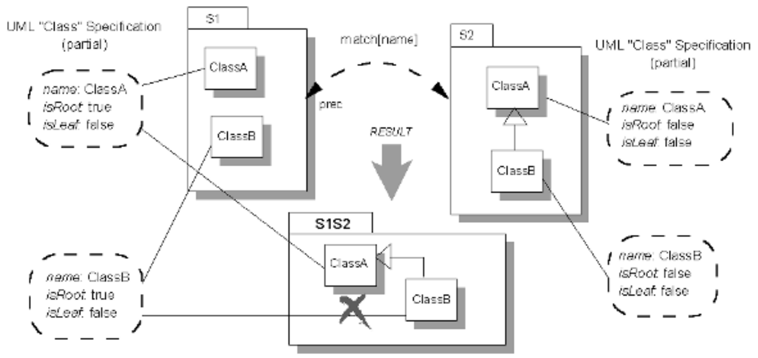


Figure 77: Breaking Well-Formedness Rules for Classifiers

This example results in one breakage of the UML well-formedness rules. Classifier is a subtype of GeneralizableElement (see “Appendix A: Partial Illustrations of UML Metamodel” on page 269), and must conform to the well-formedness rules of all generalizable elements. One rule for generalizable elements states that “A root cannot have any Generalizations” [UML Semantics Guide page 2-53, GeneralizableElement, Rule [1]]. The `S1.ClassB` which has precedence, specifies `ClassB` as being a root class, but `ClassB` in `S2` is generalised to `ClassA` and this generalization is copied to the result.

This application of the general precedence resolution strategy results in a breakage of the well-formedness rules of the model. See “Other Reconciliation Possibilities” on page 176 for how a different reconciliation strategy might have been more appropriate here.

*Other Reconciliation Possibilities*

The previous example showed how a subject can be set as the precedent subject, which means that in the event of a conflict between specifications of corresponding component elements (in this case, Classes), the values from the class in the precedent subject are copied to the result. Figure 78 illustrates the use of other reconciliation strategies.

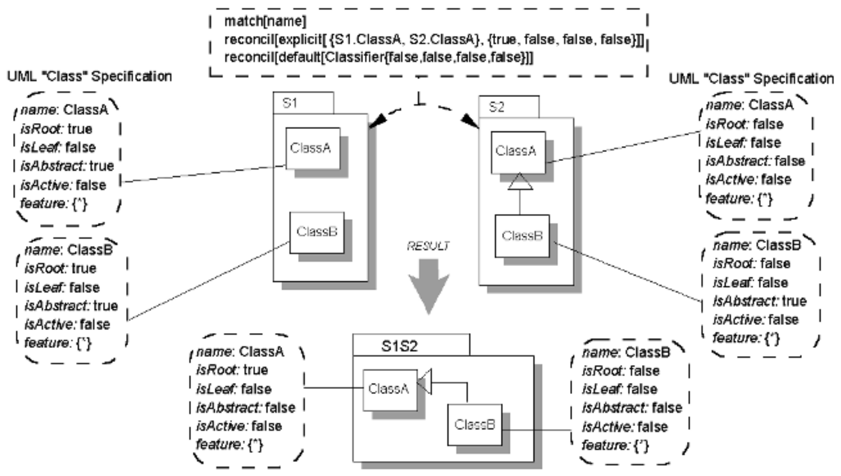


**Figure 78: Reconciling Conflicts in Classes**

Elements with correspondences and conflicts in their specifications:

- The specifications of `S2.ClassA` and `S1.ClassA` are merged. Since the names are the same, the name of the resulting subject is `ClassA`. However, the values of `isRoot` and `isAbstract` are different, so a reconciliation strategy is required. The composition relationship between

S1 and S2 has two kinds of reconciliation strategies attached. A search through the explicit reconciled elements shows that there is an explicit reconciliation for ClassA defined. Therefore values of isRoot, isLeaf, isAbstract and isActive in the resulting class are set to the explicit values listed. The components of ClassA (in feature) are considered separately.

- The specifications of S2.ClassB and S1.ClassB are merged. Since the names are the same, the name of the resulting subject is ClassB. However, the value of isRoot is different, so a reconciliation strategy is required. The composition relationship between S1 and S2 has two kinds of reconciliation strategies attached. First, a search through the explicit reconciled elements shows that there is no explicit reconciliation for ClassB. However, default values for classifiers are included, and so the values of isRoot, isLeaf, isAbstract and isActive in the resulting subject are set to the defaults listed. The components of ClassB (in feature) are considered separately.

*Revisiting Well-formedness Rules:*

The example in the previous section as illustrated in Figure 77 resulted in a breakage of the well-formedness rules of the UML when the reconciliation automatically made the values of elements in S1 take precedence in the event of a conflict. However, the example shown in Figure 78 illustrates how specifying defaults with the most flexible of values avoids problems with well-formedness rules. Here, the values of the defaults for isRoot and isLeaf are both false, which mean that a class with these values may participate as it wishes in generalization relationships.

## Impact of Merge on Attributes

This section discusses what happens to attribute specifications as a result of merge (See “Appendix A: Partial Illustrations of UML Metamodel” on page 269 for an illustration of the UML specification of Attribute).

The following subsections describe the impact of merge on the example illustrated in Figure 79.

*Correspondences*

- [Eg7.10] S1 corresponds with S2 because of the composition relationship between the two. This relationship specifies matching by name for identification of correspondence between the components
- [Eg7.11] S1.ClassA corresponds with S2.ClassA (Eg7.10)

- [Eg7.12] `S1.ClassB` corresponds with `S2.ClassC` (from the composition relationship between the two. This relationship specifies matching on name for identification of correspondence between the components)
- [Eg7.13] `S1.ClassA.a` corresponds with `S2.ClassA.a` (Eg7.10)
- [Eg7.14] `S1.ClassB.a` corresponds with `S2.ClassC.a` (Eg7.12)
- [Eg7.15] `S1.ClassB.f` corresponds with `S2.ClassC.e` (from the composition relationship between the two)
- [Eg7.16] `S1.ClassB.f` also corresponds with `S2.ClassC.f` from (Eg7.12). Recall that composable elements may participate in multiple composition relationships (see "Participation in multiple composition relationships" on page 86). Any correspondence *not* required which occurs implicitly as a result of a matching specification attached to a relationship at a higher level in the subject tree must be explicitly excluded with a composition relationship with a `dontMatch` attachment.

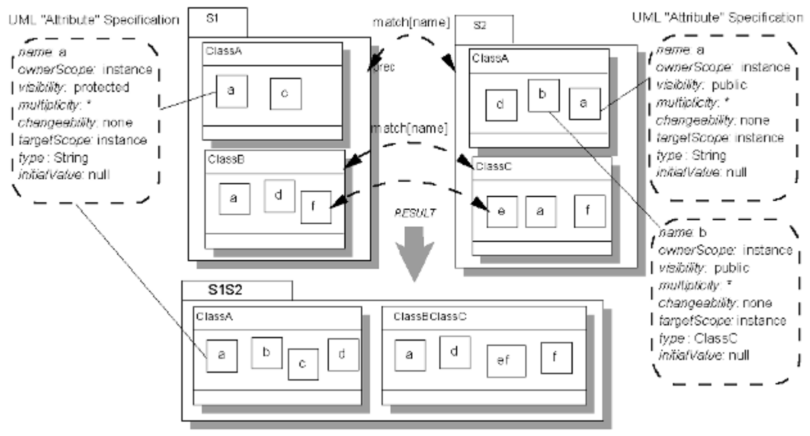


**Figure 79: Impact of Merge on Attributes**

*Result of Merge* Elements with correspondences and no conflicts:

- In the result, `S2.ClassA` is merged with `S1.ClassA`. Since their names are the same, the name of the result class is `ClassA`.
- In the result, `S2.ClassC` is merged with `S1.ClassB`. Since their names are different, the names are concatenated with the result class called "`ClassBClassC`".
- In the result, `S2.ClassB.f` is merged with `S1.ClassC.e`. Since their names are different, the names are concatenated with the result attribute called "`ef`".

- In the result, `S2.ClassB.f` is merged with `S1.ClassC.f`. Since their names are the same, the name of the result attribute is “`f`”.

Elements with correspondences and conflicts in their specifications

- The specifications of `S1.ClassA.a` and `S2.ClassA.a` are merged. Since the names are the same, the name of the resulting attribute is `a`. However, the value of the `visibility` property is different, so a reconciliation strategy is required. The composition relationship between `S1` and `S2` indicates that `S1` has precedence in the event of a conflict. Since this relationship applies here, the value of `visibility` (and all other properties) from `S1.ClassA.a` is copied to the result.

Elements with no correspondences:

- Attributes `S1.ClassA.c` and `S1.ClassB.d` have no corresponding attributes and so are added unchanged to the resulting `ClassA` and `ClassBClassC`.
- Attributes `S2.ClassA.b` and `S2.ClassA.d` have no corresponding attributes and so are added unchanged to the resulting `ClassA`.

Elements requiring change as a result of “forwarding” semantics

- Attribute `S2.ClassA.b` has a type of `ClassC` in `S2`. However, `S2.ClassC` is merged with `S1.ClassB` and, therefore, all references to `ClassC` in `S2` must be changed to its new specification, which is `ClassBClassC`.

*Check on UML Well-Formedness Rules*

The well-formedness rules for attributes are not broken with this example.

*Other Reconciliation Possibilities*

The previous example showed how a subject can be set as the precedent subject, which means that in the event of a conflict between specifications of corresponding component elements (in this case, Attributes), the values from the attribute in the precedent subject are copied to the result. Figure 80 illustrates the use of other reconciliation strategies.

Elements with correspondences and conflicts in their specifications

- The specifications of `S1.ClassA.a` and `S2.ClassA.a` are merged. Since the names are the same, the name of the resulting attribute is `a`. However, the value of the `visibility` property is different, so a reconciliation strategy is required. The composition relationship between `S1` and `S2` has two kinds of reconciliation strategies attached. A search through the explicit reconciled elements shows that an explicit reconciliation for `ClassA.a` has been defined. Therefore the values of `ownerScope`, `visibility`, `multiplicity`, `changeability`, `targetScope`, `type` and `initialValue` in the resulting attribute are set to the explicit values listed.

- The specifications of `S1.ClassA.b` and `S2.ClassA.b` are merged. Since the names are the same, the name of the resulting attribute is `b`. However, the values of the `ownerScope`, `visibility` and `type` properties are different, so a reconciliation strategy is required. The composition relationship between `S1` and `S2` has two kinds of reconciliation strategies attached. First, a search through the explicit reconciled elements shows that there is no explicit reconciliation for `b`. However, default values for attributes are included, and so the values of `ownerScope`, `visibility`, `multiplicity`, `changeability`, `targetScope`, `type` and `initialValue` in the resulting attribute are set to the defaults listed. Note that if no defaults had been listed for attribute, and no other precedence strategy attached to the composition relationship that applied here, then both `b` attributes would be added to the result, renamed by concatenating the subject name to avoid a name clash.

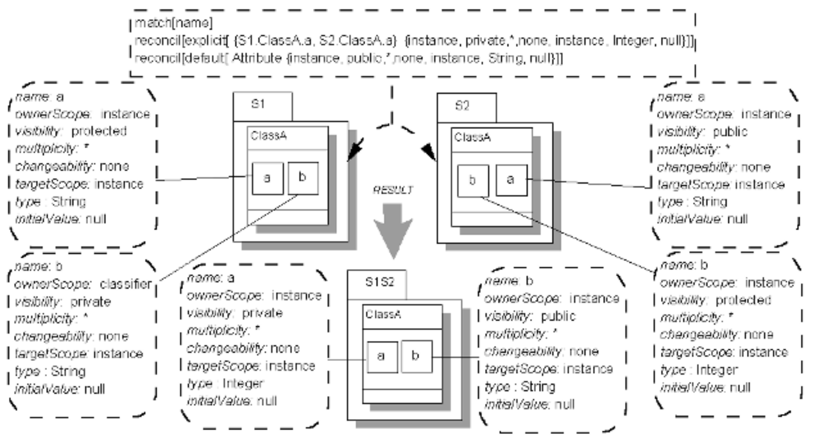


**Figure 80: Reconciling Conflicts in Attribute Specifications**

## Impact of Merge on Associations and Generalizations

This section discusses what happens to association and generalization specifications as a result of merge (See "Appendix A: Partial Illustrations of UML Metamodel" on page 269 for an illustration of the UML specification of Relationship). Then, with an example, the following are illustrated:

- How correspondences are established
- The results of merge on corresponding associations with no conflicts

- The results of merge on corresponding associations that require specification reconciliation
- Further examples of reconciliation of conflicts in associations.
- The results of merge on corresponding generalizations.
- Checking the UML Well-Formedness Rules on the results of merge

*Result of Merge for Figure 81*

The first example, in Figure 81, illustrates the merging of associations with the same name (with name match correspondence specification) but different association ends.

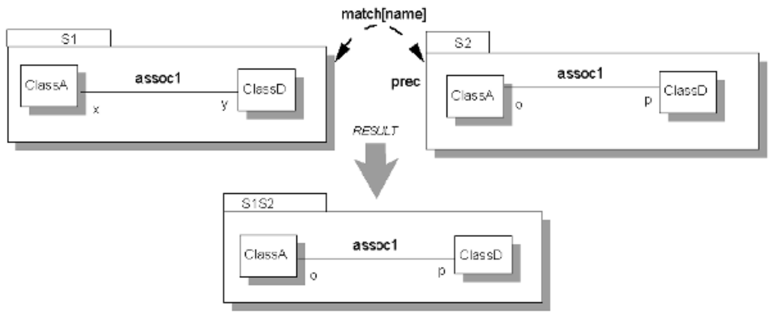


**Figure 81: Example 1: Impact of Merge on Associations**

Elements with correspondences and conflicts in their specifications

- The specifications of `S1.assoc1` and `S2.assoc1` are merged. Since the names are the same, the name of the resulting association is `assoc1`. However, the values of the `name` properties of the association ends are different, so a reconciliation strategy is required. The composition relationship between `S1` and `S2` indicates that `S2` has precedence in the event of a conflict. Since this merge applies here, the values of `name` at both ends (and all other properties) from `S2.assoc1` is copied to the result

*Result of Merge for Figure 82*

As with other elements where reconciliation may be required, defaults may be used to reconcile differences in specifications. In Figure 82, differences in the specifications of the associations in different subjects, and in one of the association ends occur. (Note, for space reasons, all the default properties for reconciliation of association ends are not listed in the diagram).

Elements with correspondences and conflicts in their specifications

- The specifications of `S1.assoc1` and `S2.assoc1` are merged. Since the names are the same, the name of the resulting association is `assoc1`. However, the values of the `isRoot` property of the association, and of the `isNavigable`, `ordering`, `targetScope` and `visibility` properties of the association ends named `x` are different, so a reconciliation strategy is required. The composition relationship between `S1` and `S2` includes defaults for association and association ends in the event of a conflict. Since this relationship applies here, the values of the conflicting properties are taken from the default and copied to the result. There are no conflicts in the specification of the association end `y`, and so the result is copied from either of the subjects. Similarly, explicit values for the association and its association ends may be specified with the composition relationship, which would be used for their reconciliation in the result.

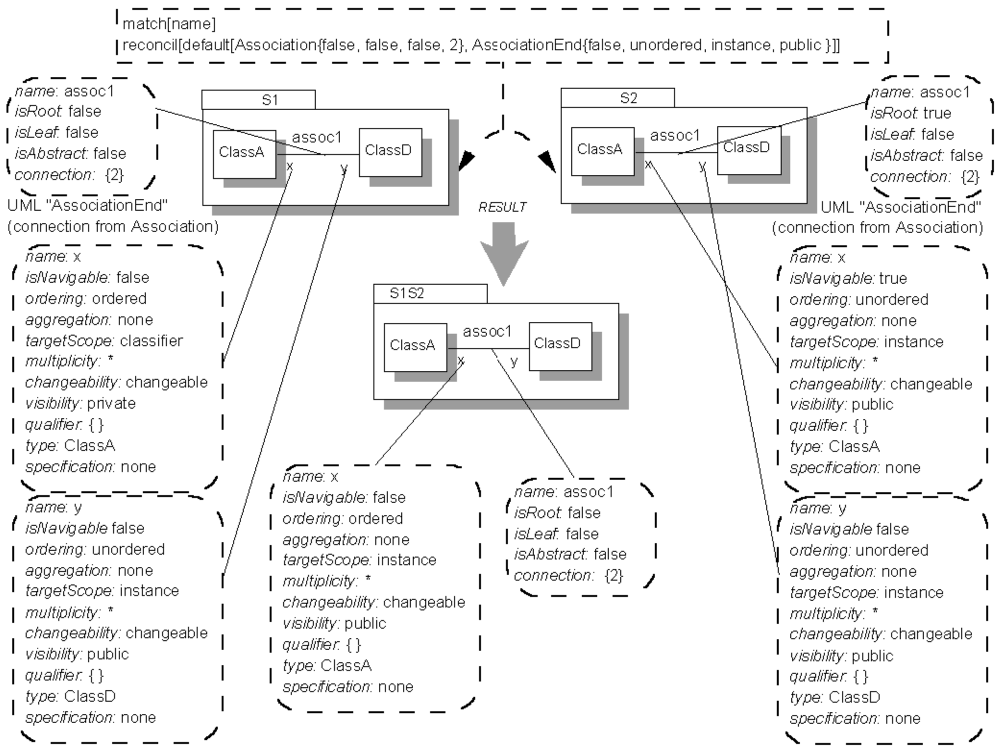


**Figure 82: Example 2: Using Defaults to Reconcile Conflicts in Associations**

*Result of Merge for Figure 83*

As with all elements, associations with no corresponding associations are added to the result (see Figure 83). Like-named associations between different sets of classifiers are deemed *not* to correspond.

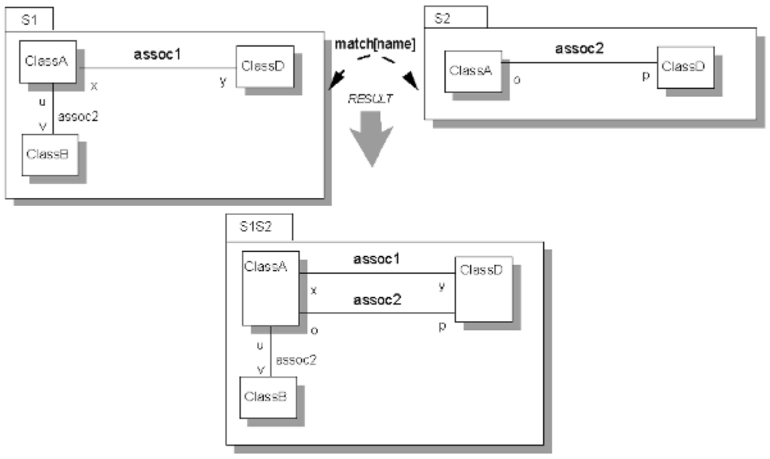


**Figure 83: Example 3: Impact of Merge on Associations**

*Generalizations* A generalization is a relationship between a more general element and a more specific element. A generalization is not a composable element, but this section considers the impact of merge on generalizations. All generalizations in the scope of a merge are added to the result. As illustrated in Figure 84, this may result in a multiple inheritance graph, where single inheritance was specified in the input subjects.

In Figure 84, the resulting `ClassC` is generalised from `ClassF` through two routes – directly, and from `ClassE`. This does not break the well-formedness rules as defined by the UML, but may not be the desired semantics. As with all design effort using generalizations, care should be taken with merge to ensure that the result is as desired.

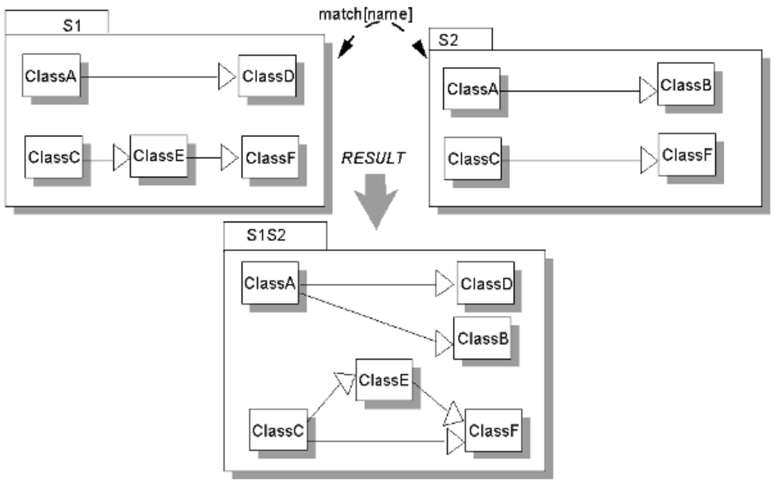


**Figure 84: Example 1: Impact of Merge on Generalizations**

*UML Well-Formedness Rules*

As with all elements, merge may result in breakages to the well-formedness rules for generalizations. In section "Impact of Merge on Classifiers" on page 173, one example was illustrated relating to the specification of root classes. Another example is illustrated in Figure 85 and relates to the well-formedness rule "Circular inheritance is not allowed" [UML Semantics Guide page 2-53, GeneralizableElement, Rule [3]].

As described previously in the semantics for override integration relating to generalizations ("Generalizations" on page 144), ideas described in [Walker 2000] could be incorporated here to eliminate cycles in composed hierarchies. This is added to future work.

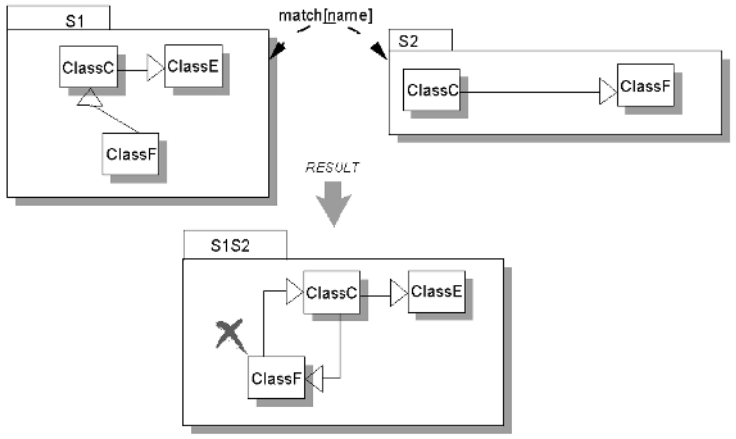


**Figure 85: Example 2: Impact of Merge on Generalizations**

## Impact of Merge on Dependencies

This section discusses what happens to dependency specifications as a result of merge (See "Appendix A: Partial Illustrations of UML Metamodel" on page 269 for an illustration of the UML specification of Dependency). The impact of merge on dependencies is illustrated with an example.

A dependency is a "using" relationship, which states that the implementation or functioning of one or more elements requires the presence of one or more elements. Dependency is not a composable element, but this section considers the impact of merge on dependencies.

In general, all dependencies in the scope of a merge are added to the result. Where there are duplicate dependencies in merging subjects, only one will appear in the result. Duplicate dependencies are of the same kind and stereotype and have the same supplier and client. Figure 86 illustrates an example.

*Result of Merge*

- All dependencies are added to the result.

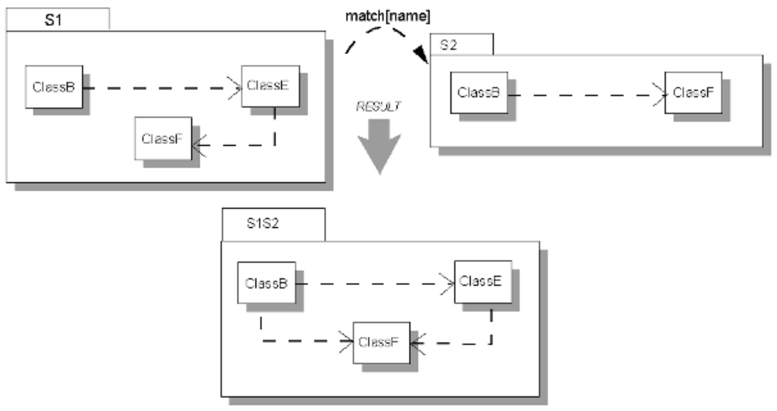


**Figure 86: Impact of Merge on Dependencies**

## Impact of Merge on Constraints

This section discusses what happens to constraint specifications as a result of merge (See "Appendix A: Partial Illustrations of UML Metamodel" on page 269 for an illustration of the UML specification of Constraint). The impact of merge on constraints is illustrated with an example.

A constraint is a boolean expression on an associated element, which must be true for the model to be well formed. Some constraints are predefined in the UML, others may be user defined. All constraints are included in the rule for merge, which states that the resulting model must be well-formed. Predefined stereotypes of constraint are invariant, precondition and postcondition.

Constraint is not a composable element, but this section considers the impact of merge on constraints (invariants). In general, all constraints in the scope of a merge are added to the result. Where there are corresponding elements where only one representative element is added to the result (e.g. classifier, attributes), constraints on those elements are all added to the result, with the effect of a boolean `and` across the constraints that were defined for corresponding elements in the input subjects. Care should be taken when merging constraints to ensure that the semantics of the constraints do not conflict or have unanticipated implications. In some cases, merging of some constraints may break the well-formedness rules of the model.

Pre and post conditions are discussed with operations in "Impact of Merge on Operations" on page 188.

*Result of Merge on Figure 87*

In the first case, user-defined constraints in the separate subjects are added to the merged subject.

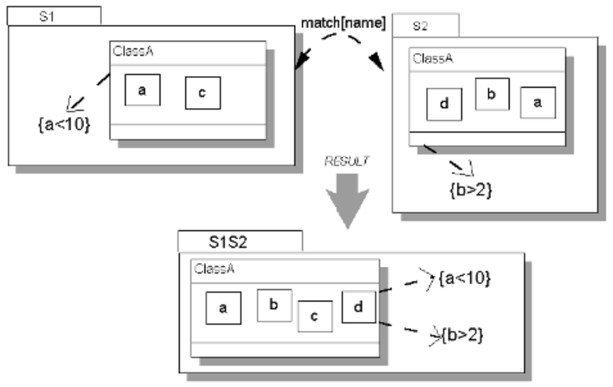


**Figure 87: Example 1: Impact of Merge on Constraints**

- Constraints on attributes `S1.ClassA.a` and `S2.ClassA.b` added to result

*Result of Merge in Figure 88*

As with the direct writing of constraints on a model, care should be taken to ensure the constraints in the result of a merge integration remain as intended. Adding constraints in this manner may result in unanticipated or conflicting implications. For example, in Figure 88, constraints on `ClassA.a` imply that `ClassA.c` must always be negative.

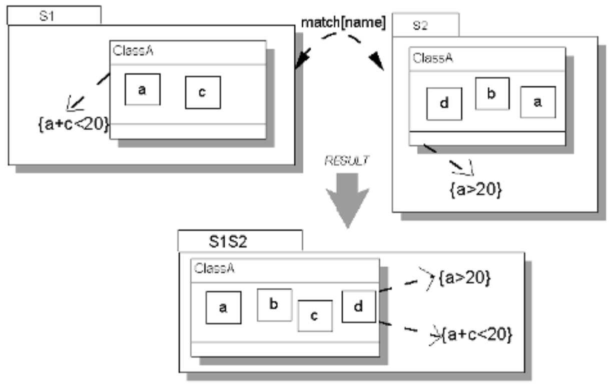


**Figure 88: Example 2: Result of Merge on Constraints**

- Constraints on attributes `S1.ClassA.a+A1.ClassA.c` and `S2.ClassA.a` added to result

*Result of Merge in Figure 89*

Figure 88 illustrated an example of an unanticipated implication of merging constraints. There is also the possibility that merging constraints will result in incorrect and conflicting constraints. Figure 89 illustrates this possibility and highlights the problems with a “?”. The supporting text following the diagram answers the implied question by describing the policy of merge.

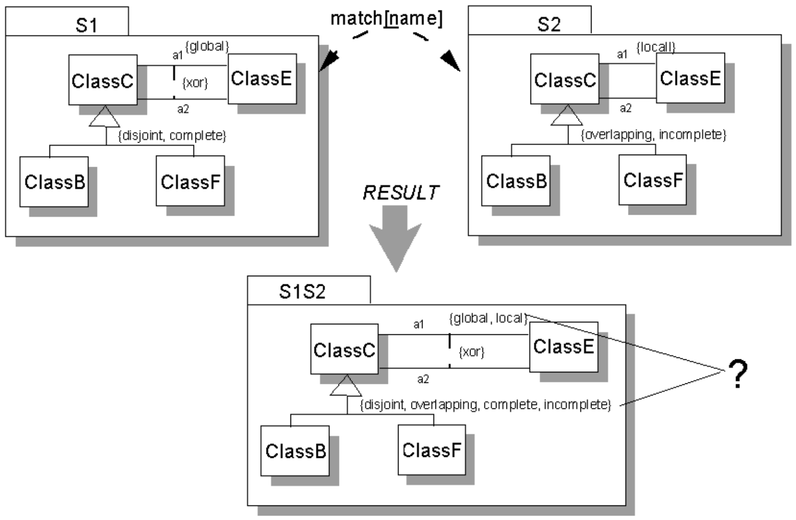


**Figure 89: Example 3: Impact of Merge on Constraints**

- Constraints on the generalizations to `S1.ClassC` and `S2.ClassC` are added to result. However, these constraints now conflict, as a generalization cannot be both `disjoint` and `overlapping`, and cannot be both `complete` and `incomplete`.
- The constraints on the associations `S1.a1` and `S2.a1` are added to the result. However, an association cannot be both a `global` and a `local` association.
- The `xor` constraint between `S1.a1` and `S1.a2` is added to the result. This causes no conflict.

As described previously, the general policy of composition is to perform the composition as specified, and to highlight breakages of the UML well-formedness rules as a result. Unlike classifiers and operations, the policy for merging constraints is to add all specified constraints. Conflicts in, for example, attributes can have reconciliation applied since only one representative attribute of corresponding ones appears in the result. Since this is not the case for constraints, such reconciliation does not apply, and so conflicts may exist in the composed result.

In this case, however, there is a strong temptation to attempt to automatically "fix" the problems that are illustrated in Figure 89. Possible approaches to such fixes might be to automatically add the more flexible constraints in the event of a conflict (e.g. making the generalization `incomplete` and `overlapping`) or perhaps the opposite by adding the more restrictive options. With whatever policy that might be adopted for automating fixes, there remain two fundamental problems:

- *Domain Semantics:* It is not always possible to reason about the intentions of the designer. In this example, it is not possible to decide whether the designer who specified the generalization as `disjoint` and `complete`, and the association `global`, was correct in reflecting the constraints of the domain in `S1`, or the decisions the designer of `S2` made were correct. Possibly, they were both correct for their own subjects. But, what is correct in the merged subject? Since the answer to this question lies in the semantics of the domain, it is therefore safer to highlight the conflict in the result, and ensure that an informed choice is made based on the requirements.
- *Consistency:* Constraints in UML models may be pre-defined by the UML, or user-defined constraints. Where constraints are user-defined, it is more difficult to define an automatic policy to adopt to handle conflicts, and therefore, if there was a policy for those constraints pre-defined for the UML, there would be an inconsistency in the behaviour of composition – some constraint conflicts "fixed" and some not.

*Check on UML Well-Formedness Rules*

Constraints are included in the well-formedness specification of a model.

## Impact of Merge on Operations

This section discusses what happens to operation specifications as a result of merge (See "Appendix A: Partial Illustrations of UML Metamodel" on page 269 for an illustration of the UML specification of Operation).

Then, with an example, the following are illustrated:

- How correspondences are established
- The results of merge on corresponding operations when no collaboration is attached to the merge
- The results of merge on corresponding operations with a collaboration attached.
- Checking the UML Well-Formedness Rules on the results of merge

*Merging Operations with no Attached Collaborations*

Merging operations means that corresponding operations' behaviours are joined together. This means that the execution of any one of the corresponding operations results in the execution of all of the corresponding operations. Specification of this behaviour is achieved within the subject-oriented design model by generating interaction diagrams realising the composed operation as delegating to each of the corresponding input operations on invocation of the composed operation. Input operations may be renamed to avoid a name-clash. Re-naming is achieved by pre-pending the name of the input subject, followed by an underscore, to the operation name. Input operations are also given protected visibility in the output.

In Figure 90, examples of merging corresponding operations are illustrated, showing:

- The re-naming of corresponding input operations and the creation of operations used to specify the behaviour of merged operations - that is, that all corresponding operations are executed when any one of them is executed. See "Composition relationship with No Attached Collaboration" on page 161 for a discussion on different solutions considered here.
- Use of a primitive composition relationship to indicate correspondences between particular operations.
- Correspondences between operations are only established within classifiers that correspond.
- Collaborations are generated to specify the combined behaviour of corresponding operations.

*Correspondences:*

- [Eg7.17] `S1` corresponds with `S2` because of the composition relationship between the two. This relationship specifies matching on name for identification of correspondence between the components
- [Eg7.18] `S1.ClassA` corresponds with `S2.ClassA` (Eg7.17)
- [Eg7.19] `S1.ClassB` corresponds with `S2.ClassB` (Eg7.17)
- [Eg7.20] `S1.ClassA.op1` corresponds with `S2.ClassA.op1`, and `S1.ClassB.op4` corresponds with `S2.ClassB.op4` (Eg7.17)
- [Eg7.21] `S1.ClassA.op3` corresponds with `S2.ClassA.op2` (from the composition relationship between the two)

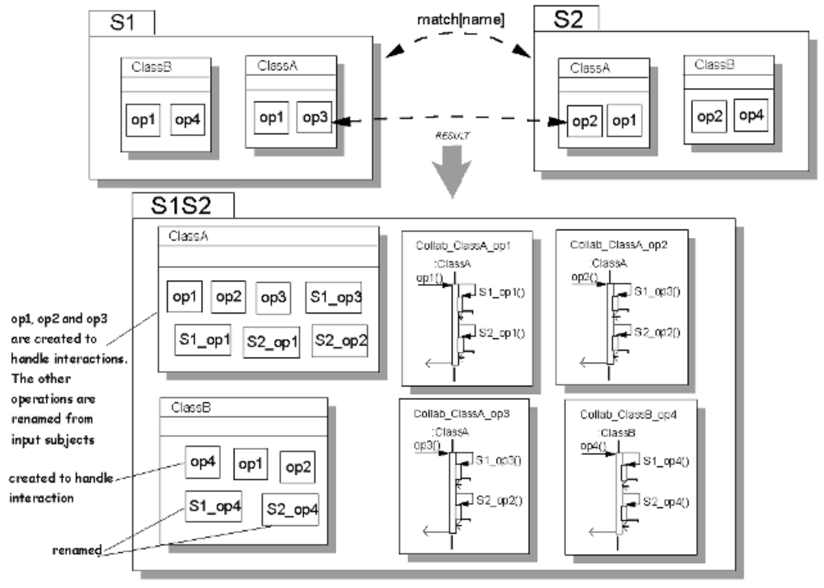


**Figure 90: Impact of Merge on Operations**

*Result:*

- After renaming, `S1.ClassA.op1` and `S2.ClassA.op1` are added to the result. A new `op1` is created and added to the result, realised by a new collaboration which is also created. This collaboration indicates that on execution of an `op1` operation, both `S1_op1` and `S2_op1` are executed. `S1_op1` and `S2_op1` have protected visibility.
- `S1.ClassA.op3` and `S2.ClassA.op2` are renamed and added to the result. Two new operations `op2` and `op3` are created, realised by two new collaborations which are also created. These collaborations indicate that on receipt of either an `op2` or an `op3` message, both `S1_op3` and `S2_op2` are executed. `S1_op3` and `S2_op2` have protected visibility.
- After renaming, `S1.ClassB.op4` and `S2.ClassB.op4` are added to the result. A new `op4` is created and added to the result, realised by a new collaboration which is also created. This collaboration indicates that on receipt of an `op4` message, both `S1_op4` and `S2_op4` are executed. `S1_op4` and `S2_op4` have protected visibility.

*Operations involved in Multiple Compositions*

The composition of designs model allows for composable elements to participate in multiple composition relationships (see “Participation in multiple composition relationships” on page 86). For merging operations, this has the potential to cause some ambiguity. For example, in Figure 91 the operation

S1.ClassA.op3 corresponds with two different operations. One is as a result of an explicit composition relationship between S1.ClassA.op3 and S2.ClassA.op2, and the other is as a result of the matching by name criteria specified in the composition relationship between S1 and S2. The semantics of merging operations states that the execution of any one of a corresponding set of operations means the execution of each of the operations in the corresponding set. However, since there are two corresponding sets of operations for op3, there is ambiguity as to which interaction is appropriate. As with specifying composition relationships in general, care should be taken to ensure that the behaviour in the output is as required, though this ambiguity can be resolved by attaching additional collaborations to the composition relationship.

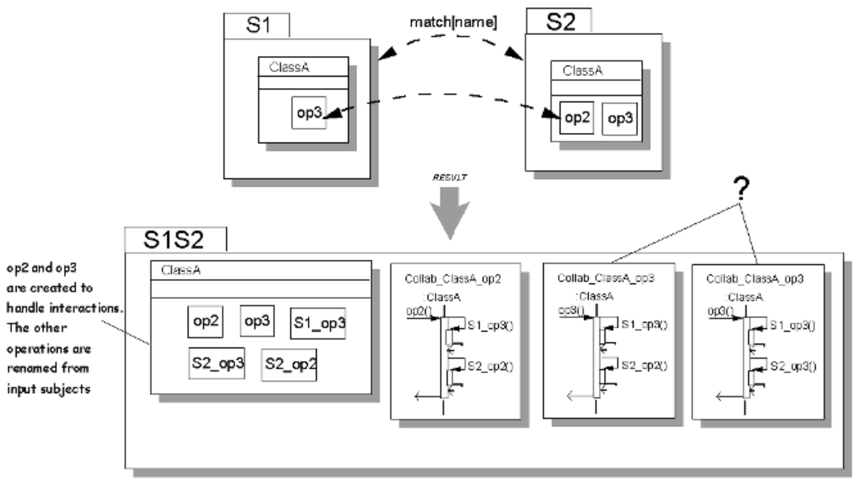


**Figure 91: Operations involved in Multiple Compositions**

*Merging Operations with Attached Collaborations*

When the order of execution of corresponding operations is important, a collaboration(s) specifying this order should be attached to the composition relationship. In this case, the attached collaboration is added to the merged subject as the specification of the behaviour of corresponding operations.

*Result of merge in Figure 92:*

- S1.ClassA.op3, S2.ClassA.op1 and S2.ClassA.op2 are corresponding and are renamed and added to the result. The three collaborations attached to the composition relationship are added to the result indicating that on execution of an op1 or an op2 or an op3 operation, S2_op1, S1_op3 and S2_op2 are executed in that order.

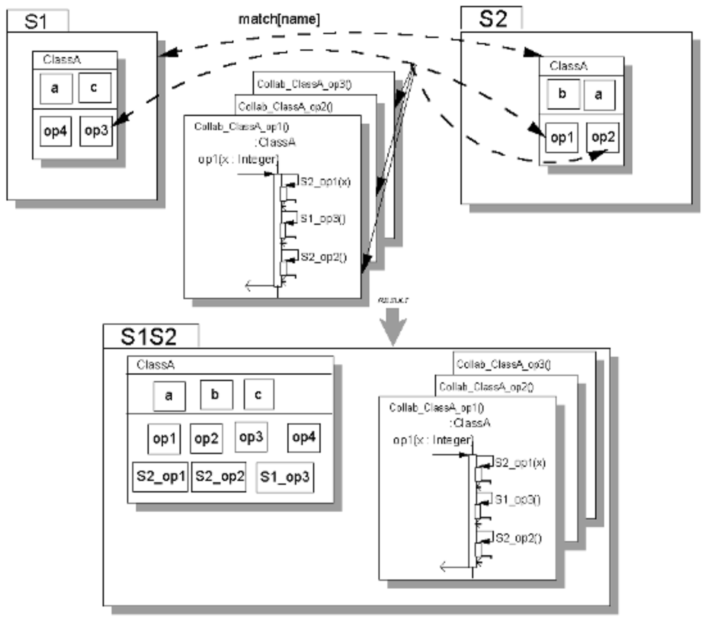


**Figure 92: Merging Operations with Attached Collaborations**

Where an operation is part of a corresponding group of operations, and is *not* realised by a collaboration attached to the composition relationship, a call to that operation does not result in delegation to all of the operations in the corresponding group. This is only the case where *at least one* collaboration is explicitly attached to the composition relationship. Where no collaboration is attached, then collaborations are generated for all of the operations. This behaviour supports the designer excluding a specific operation in the corresponding group as always resulting in all of the operations being executed.

*Conflict Rules for Merging Operations*

There are various ways in which the specifications of operations may be different, and this section looks at the impact of merge when the specifications of operations defined as corresponding are different.

*Conflicting Parameter Lists:*

The general rule relating to merging operations is that they must have the same parameter list. On execution, values input to the composed operation may then be used in the calls to each of the corresponding operations.

One exception to this rule is included. 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. Without such a collaboration, the operations will be deemed to conflict, therefore treated as non-corresponding, and will not be merged. Figure 93 illustrates how a designer may merge operations with conflicting parameter lists.

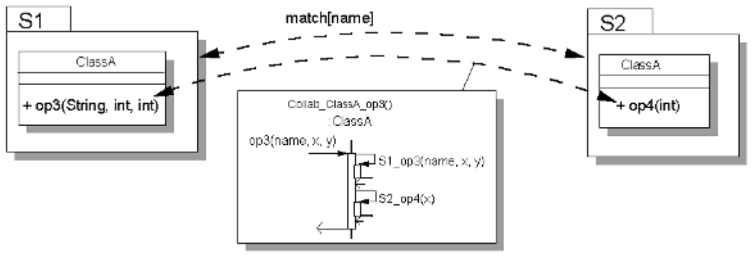


**Figure 93: Merging Operations with Different Parameters**

*Other conflicting properties:*

There is other potential for apparently conflicting properties in operations that have been specified as corresponding. For example, in Figure 94, `op1`, `op2` and `op3` are `private`, `protected` and `public` respectively. Other differences are illustrated for each of the other properties of operation. It is the policy of merge integration that operations with conflicting properties are deemed to be non-corresponding. In this case, they are treated as any non-corresponding elements, and not merged.

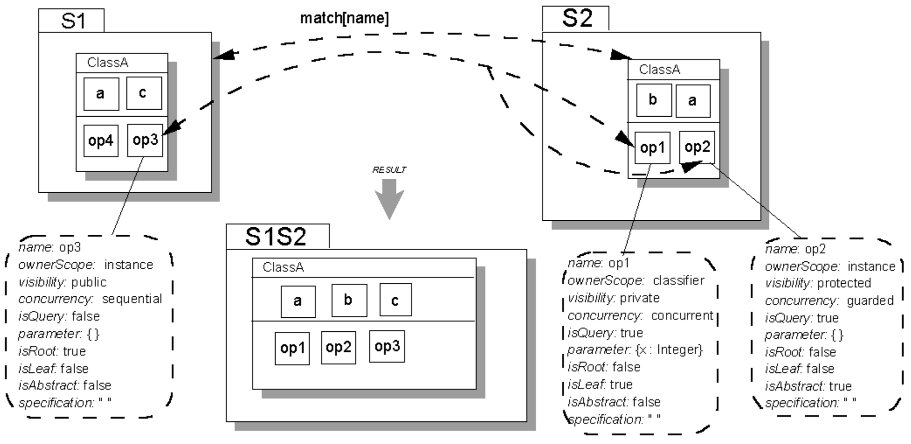


**Figure 94: Merging Operations with Other Conflicting Properties**

Concerns with the rigidity of this approach are discussed in “Incompatible Elements” on page 100. Here, it is concluded that a taxonomy of rules to guard against integration of truly incompatible elements, but allow some possibilities, is the best approach. This is added to future work for composition of design models.

*Pre and Post conditions for corresponding operations:*

As with constraints in general, each pre and post condition for each corresponding operation is added to the result, which may have unpredictable results. In the example in Figure 95, the only time that `op2()` will execute is if `op3()` changes the value of `a` to be `> 50`. This may or may not be what is required. The general advice for constraints applies here. Care must be taken when merging operations with pre and post conditions, that the combination, if not disjoint, makes sense.

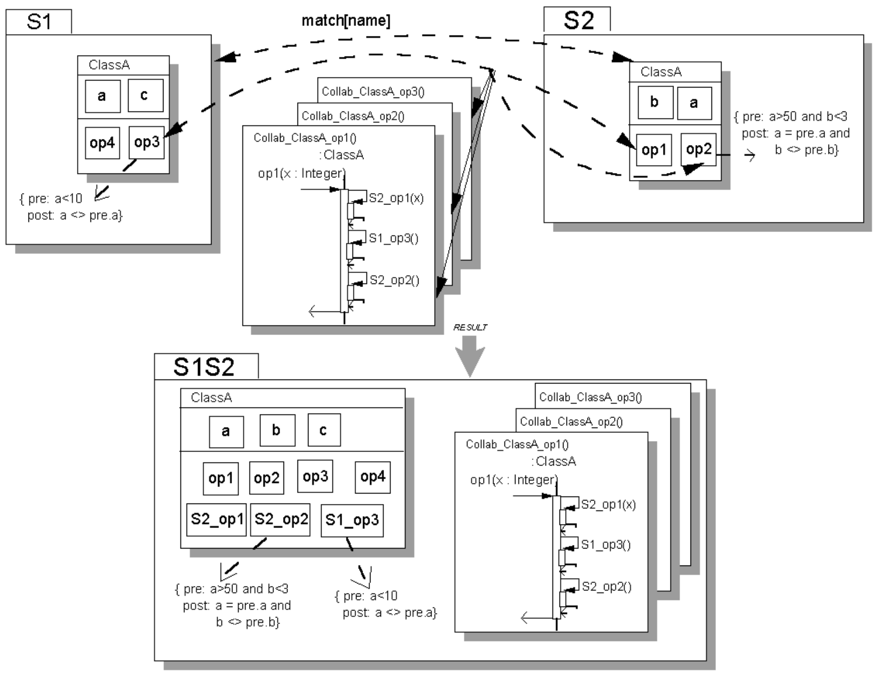


**Figure 95: Merging Operations with Pre/Post conditions**

*Merged Operations with Return Types*

Where corresponding input operations each have a return type, what type should the composed operation return? The subject-oriented programming domain, as described for Hyper/J in [Tarr & Ossher 2000], supports what they call *summary functions*, which synthesise the return values of each of the methods to return a value appropriate for the collaborating methods. A summary function, defined by the developer, takes as input an “array of values” that were returned by the composed methods, and uses them to compute a single return value. Where a summary function is not defined, the default behaviour is that the value returned by the *last* of the methods executed is the one returned by the composed method.

This is also an issue within the subject-oriented design domain. Further research is required to assess the feasibility of a "summary function" equivalent solution. This may require an additional attachment to a composition relationship, but should be examined further to define the best solution. Currently, behaviour similar to the default behaviour defined in Hyper/J is defined within the subject-oriented design model. The value returned by the last input operation executed is the value returned by the composed operation. Which operation this is may be manipulated by the composition designer with a collaboration attached to the composition relationship specifying which operation is executed last.

*Merged Operations and Forwarding of References*

References to operations input to merge integration are forwarded to the output operation that delegates to the corresponding set of operations. These operations are the ones with the same signature as the input operations, created to be realised by interaction models defining the delegating semantics.

There is potential here for reducing the number of operations that need to be created to be realised as delegating to each of a set of corresponding input operations. For example, in Figure 90 on page 190, two operations (and interaction specifications) are created to define the delegation to both `S1.ClassA.op3()` and `S2.ClassA.op2()`. Here `S1.ClassA.op3()` forwards to `S1S2.ClassA.op3()` in the result, and `S2.ClassA.op2()` forwards to `S1S2.ClassA.op2()` in the result, each of which is realised by a collaboration. Since each defines the same behaviour, there is some repetition here. Research is required to assess the potential for extending this semantics to all multiple input operations forward to a single delegating operation.

## Impact of Merge on Collaborations

Since all corresponding operations are added to the result, so also are all collaborations added to the result. Re-naming may be required in some cases where collaborations have a name clash. Figure 96 illustrates the result of merging collaborations.

*Result of Merge*

- After renaming, `S1.ClassA.op1` and `S2.ClassA.op1` are added to the result. A new collaboration is created and added to the result indicating that on execution of `op1` , both `S1_op1` and `S2_op1` are executed.
- After renaming to avoid a name clash, `S1Collab1` and `S2Collab1` are added to the result. The changed names of `S1_op1` and `S2_op1` are reflected in the added collaborations for the two operations.

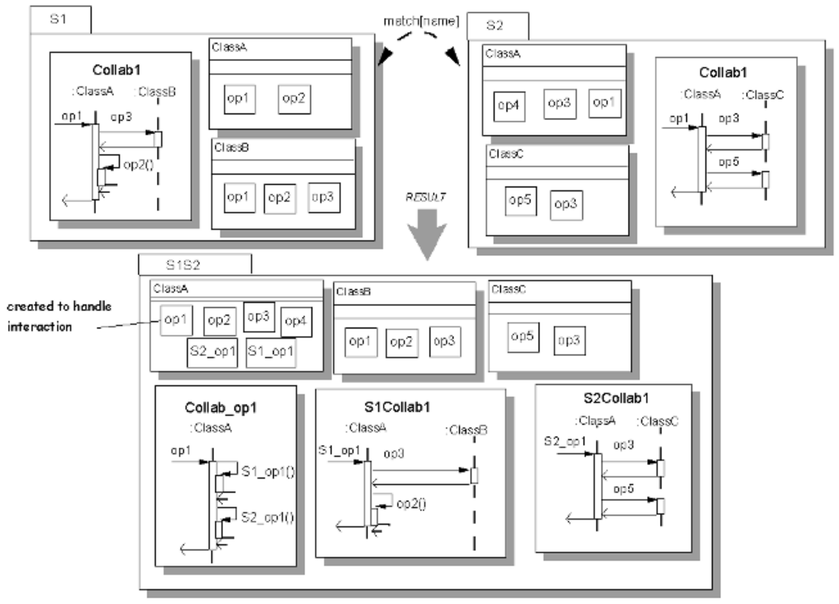


**Figure 96: Impact of Merge on Collaborations**

## 7.5. Chapter Summary

This chapter defines the syntax and semantics of composition relationships with merge integration. Changes to the UML metamodel to support the syntax are illustrated as an extension to the composition relationship metamodel as described in “Composition Relationship” on page 113. Well-formedness rules for composition relationships with merge integration are given. These rules are primarily related to the specification of reconciliation strategies for conflicting elements, and the attachment of collaborations to composition relationships. All rules for general composition relationships, defined in “Well-Formedness Rules” on page 117, apply for the relationships with merge integration.

The semantics for merge integration are defined by illustrating the impact of merging each of the design elements currently supported in the thesis. First, general semantics for merge are defined. For some elements (for example classifiers and attributes) one element, representative of all corresponding elements, is copied to the output. In this case, it is important to assess whether there are any conflicts in the properties of the corresponding elements. It is illustrated and described how different kinds of reconciliation strategies may be used to resolve any conflicts.

The semantics for merging operations is different in that all corresponding operations are added to the output, because execution of an operation in the output means that all corresponding operations are executed. This behaviour is specified by the creation of an interaction for inclusion in the output. The order may be controlled by attaching an interaction to the appropriate composition relationship, which is then copied to the output.

In general, in order to fully define the semantics, the impact of merge on each construct is examined, with any change from the general semantics highlighted as appropriate.

The next chapter looks at the kinds of requirements that may impact multiple classes in multiple different design models. The manner in which their behaviour impacts these different models is similar in every case, and therefore can be seen as *patterns*. The notion of *composition patterns*, supporting the capture of patterns of cross-cutting behaviour into a separate design model, is described. It is illustrated that the design of such a requirement may be achieved without explicit reference to any class it may impact. Composition patterns are based on merge integration semantics, and on UML templates.

# Chapter 8: Composition Patterns

One of the benefits of subject-oriented design is that a requirement that has an impact across multiple classes in the system design, i.e., a cross-cutting requirement, may be decomposed into a separate design model. “Chapter 7: Merge Integration” on page 155 discusses the semantics of merging different design subjects. This chapter discusses how patterns of composition may occur, and presents a solution based on a combination of the subject-oriented design merge integration model and UML templates. Patterns of composition occur when a design subject with cross-cutting behaviour is likely to be merged with other design subjects in the same manner each time. Specification of such a design subject is deemed to be a *composition pattern*.

As discussed throughout this thesis, some kinds of requirements may have an impact on multiple classes in a design model. For example, a requirement for an audit trail of operation execution has an impact on all operations in a model. In this case, if the audit trail requirement states that an operation’s execution entry should be logged and its execution exit should also be logged, then the specification of this logging behaviour is the same for all operations. Similarly to any requirement, logging functionality may be designed separately in a subject; in such a subject, operations are likely to be included to handle the logging before execution, and to handle the logging after execution. One approach to merging this subject with any other subject is to design collaborations to be attached to a composition relationship (as described in “Attaching a Collaboration to a composition relationship” on page 162) that specify the appropriate order for execution for each operation to be logged. While this would work, it is a cumbersome solution to a merge integration that is the same in every case – every operation would need its own collaboration specifying the same order of execution with the logging operations. Where a merge like the logging one described is the same for every merge case, it is considered to be *pattern* of cross-cutting behaviour.

## 8.1. Composition Patterns Model

Patterns of cross-cutting behaviour may be abstracted and designed separately from the other design elements this behaviour may impact. Within this thesis, such separated designs of patterns of behaviour are called *composition patterns*. Composition patterns make use of template parameters from the UML, and combine them with merge integration semantics. This section looks at how this is achieved.

**Merge Integration**

The subject-oriented design composition model essentially takes a set of input subjects and integrates them according to the strategy defined by a (set of) composition relationship(s), producing an output subject. Different integration semantics define how elements specified as corresponding are composed. The particular integration strategy relevant for composition patterns is merge. Merge integration effectively joins the input subjects, reconciling differences in element specifications (except for operations) based on specified reconciliation strategies. Merged operations combine the behaviours realized by each corresponding operation. This is achieved with the generation of an interaction model realizing the composed operation as delegating to each of the corresponding input operations.

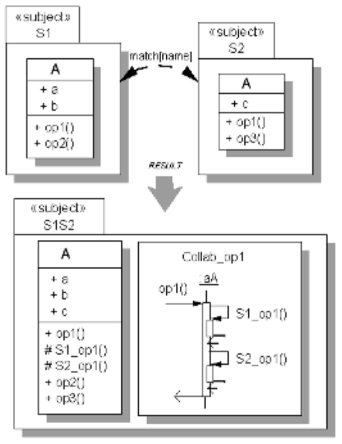


**Figure 97: Merge Integration Example**

For example, Figure 97 illustrates two subjects, each with one class. The composition relationship between the two specifies that the subjects are to be merged (denoted by arrowheads at each end of the arc) and that elements with the same name correspond to each other (denoted by `match[name]` attachment to the relationship).

In the result in Figure 97, the classes `S1.A` and `S2.A` are merged because they have the same name. Each of those classes has an `op1()` specification which are deemed to be corresponding. Merge semantics defines the output `op1()` as delegating to the two input specifications of `op1()`, which have been re-named to avoid a name-clash. Renaming is by pre-pending the name of the input subject, followed by an underscore, to the operation name. An interaction diagram is generated to define the delegation behaviour. Where the order of execution is important, the designer may attach an interaction to the composition relationship defining the required order. See "Chapter 7: Merge Integration" on page 155 for details of the semantics of merging subjects.

## UML Templates

Template parameters may be seen as dummy model elements that are designed to be replaced by "real" model elements as needed. The UML defines a template as a parameterized 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 restricts the binding of arguments to template parameters as one-to-one for instantiation. Parameterized collaborations are supported to capture the structure of a pattern, where the base classifiers are templates. This, however, does not cater for combining patterns of behaviour with behaviour in replacing classifiers – in other words, combining patterns of cross-cutting behaviour with the behaviour it cross-cuts.

## Combining the Two: Composition Patterns

A composition pattern is a design subject in which at least one pattern class (a class that is a placeholder to be replaced by a real class element) has been specified. Composition patterns harness the strengths of both the subject-oriented design merge composition model and UML templates. Using composition patterns, patterns of collaboration may be defined for cross-cutting behaviours. Within pattern classes, both template parameter elements, and non-template elements may be defined. Merge integration semantics, with a `bind[`*`params`*`]` attachment to the composition relationship, specify the replacement elements for template parameters, and how they are integrated. In the remainder of this section, the following parts of the composition pattern model are described:

- Composition Pattern Specification: Here, how a designer specifies a composition pattern is described.

- Composition Binding Specification: Here, how a designer defines which elements replace all pattern template elements is described – i.e., how a composition pattern should be composed with (an)other design subject(s).
- Composition Output: Here, the result of a composition process involving a composition pattern is described.

## Composition Pattern Specification

As discussed previously, composition patterns are based on subject-oriented design merge semantics, and UML templates. This combination requires some extensions to template specifications as defined by the UML, and also requires the composition designer to be more aware of the details of delegation and renaming of merged operations semantics than is required of a standard model composition designer. For example, in Figure 97, the designer simply indicated, with the composition relationship `match[name]` specification, that the `op1()`s corresponded, and the composition process took care of the re-naming and delegation specification. In this section, how a designer can harness this semantics to define reusable cross-cutting behaviour is described. First though, how does a designer specify templates within a composition pattern?

### *Specifying Templates*

As with any object-oriented design, the design of a cross-cutting requirement may require multiple classes and operations to support its design. A cross-cutting requirement may also impact different kinds of classes in different ways. Therefore, a composition patterns designer needs to be able to specify any number of classes within the composition pattern subject that contain properties to be merged with any replacing class. These are *pattern classes*. The designer also needs to be able to specify that there are operations within a pattern class that are expected to be replaced on composition because the composition pattern has defined behaviour to be merged with these operations. These are *template parameters*. Both of these are analogous to the pattern classes and template parameters within the UML.

The UML represents template parameters in a template box on the template class, ordered to support a Binding relationship. Since a composition pattern is a subject with potentially multiple pattern classes, 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 pattern class (each class grouped by <> brackets). Similarly to templates in the UML, ordering of pattern class groups, and template parameters within the pattern class groups, is important to support composition specification.

This chapter uses the observer pattern [Gamma et al. 1994] for the purposes of demonstrating the composition patterns model. This pattern defines a "one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated immediately". Such behaviour may be considered as "cross-cutting", as behaviour defining that the change of state of one object (a subject) initiates the notification and update of its dependent objects (observers), affects both subjects and observers. In addition, this behaviour is not specific to any business domain, but is relevant for any domain.

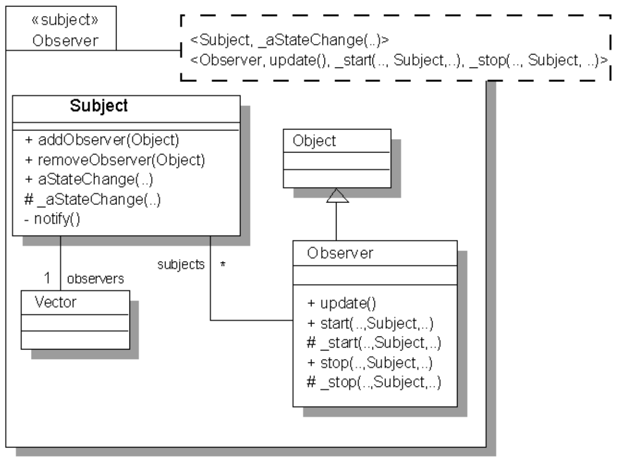


**Figure 98: Specifying Templates in a Composition Pattern**

Figure 98 illustrates a composition pattern supporting the observer pattern. There are two pattern classes, `Subject` and `Observer` defined, the first to represent subjects whose changes in state is observed, and the second to represent any classes observing a subject's state. Two standard classes are also defined, `Vector` and `Object`.

As defined in the template box, the template parameter for pattern class `Subject` is operation `_aStateChange(..)`. Within the pattern, this template is utilised to represent any state-changing operation within a subject class. The "`..`" specification of the parameters denotes that any operation signature may replace `_aStateChange(..)` (see "Template Scope" on page 204 for details).

As also defined in the template box, the template parameters for pattern class `Observer` are operations `update()`, `_start(..,Subject,..)` and `_stop(..,Subject,..)`. Within the pattern, `update()` represents the operation to be called to update observers as a result of a state change in the observed subject. `_start(..,Subject,..)` and `_stop(..,Subject,..)` represent the triggers that begin and end, respectively, an observer's interest in a subject's state. The "`..,Subject,..`" specification of the parameters denotes that this template operation requires a parameter of type `Subject` somewhere in the pattern list (see "Template Scope" on page 204 for more details).

Pattern classes need not specify additional templates within the pattern class, as the pattern class may simply specify elements to be merged into a substitution class.

*Utilising Operation Merge Semantics*

As discussed previously, where an operation's behaviour cross-cuts operations in a different design subject, a composition relationship specifies that these operations are corresponding in order to merge their behaviours. To achieve this, merge integration produces an output operation realised by a collaboration specifying delegation to each of the corresponding (re-named and protected) input operations (see Figure 97 on page 199).

A composition patterns designer needs to be able to explicitly define how the cross-cutting behaviour collaborates with merged behaviour, and that this collaboration is appropriate for all compositions with the pattern subject. To achieve this, the semantics for merging operations can be utilized. Using interaction diagrams, the composition pattern designer may explicitly refer to the output and input 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 (see Figure 99). The generated output operation is referenced with the same name, but without the pre-pended underscore.

As specified by the composition pattern in Figure 99 for pattern class `Subject`, execution of any operation that replaces `_aStateChange(..)` will, in the output subject, result in the execution of `notify()` after the execution of the replacing operation. Note, `_aStateChange(..)` was also given protected visibility as defined by merge integration (see Figure 98 on page 202). Similarly, `addObserver(Subject)` will be executed after any operation replacing `_start(..,Subject,..)`, and `removeObserver(Subject)` will be executed before any operation replacing `_stop(..,Subject,..)`.

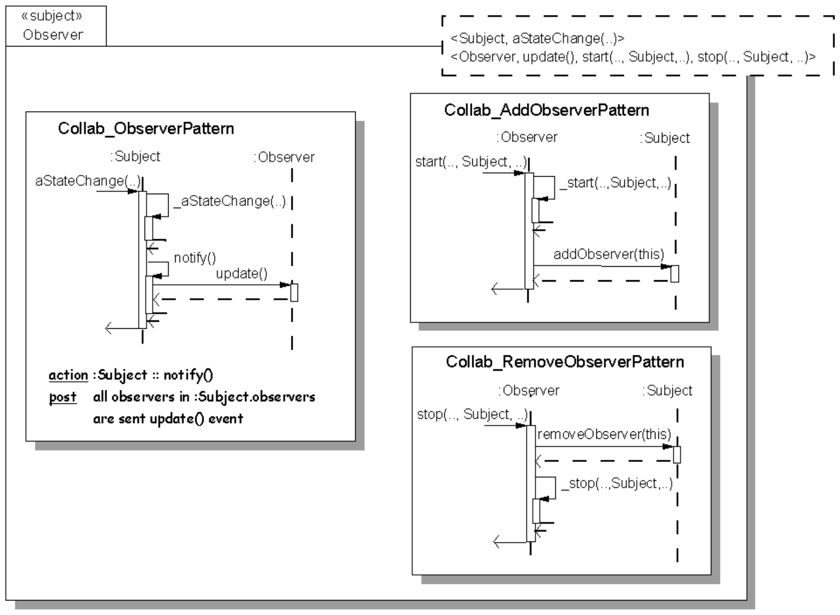


**Figure 99: Specifying Patterns of Cross-Cutting Behaviour**

Where no additional behaviour is required for a template operation, use of an additional protected operation pre-pended with an underscore is not required. One example of this here is the `update()` operation.

*Template Scope*

When specifying template operations, we have seen different kinds of parameter possibilities defined for those operations. The different possibilities relate to the scope within which the replacing operation is executed. For example, in Figure 99, the active period of the execution of `aStateChange(..)` defines the scope for this operation, and any parameters defined may be used within this scope. There are three possibilities for this specification as follows:

| Parameter | Usage |
|---|---|
| `opTemp()` | In this case, the replacing operation must have no parameters. This is used when the replacing operation is called within a pattern interaction, but it is not possible to ensure that any required parameters are possible to supply when executed within the pattern interaction. |
| `opTemp(..)` | In this case, the replacing operation may have any parameters defined. Here, the pattern interaction is defined entirely within the scope of the replacing operation. |

**Table 2: Template Parameters Scope**

| | |
|---|---|
| `opTemp(.., TypeName,..)` | In this case, the replacing operation may have any parameters defined, but one of the parameters must be of type `TypeName`. Here, the pattern interaction is defined entirely within the scope of the replacing operation, but an operation call is made to a `TypeName` instance which must be supplied. Where there is more than one parameter of type `TypeName`, the first is used. |

**Table 2: Template Parameters Scope**

*Further Potential for Template Rule Specification*

In the current composition patterns model, the properties of pattern classes and template operations are entirely replaced by classes and operations replacing them (i.e., those properties whose impact is considered for integration semantics -- see "Impact of Merge on Classifiers" on page 173 and "Impact of Merge on Operations" on page 188). For example, a template operation whose visibility is defined as `private` will not impose a `private` visibility in a replacing operation whose visibility is not defined `private`. The visibility of the replacing operation (and all other properties) take precedence in the result.

This, however, is an area where an examination of the feasibility of extending the capabilities of composition patterns is appropriate. Further research is required to explore extensions to this model. For example, a composition designer could specify constraints on the kinds of elements that may replace templates, and the conditions under which different kinds of elements may replace templates.

## 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 composition patterns model combines the two notions by extending standard composition relationships with a `bind[]` attachment that defines the elements that replace the templates within the composition pattern. The 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. The possibilities for parameters to the `bind[]` attachment are as follows:

| Parameter | Usage |
|---|---|
| `<` *`params`* `>` | Parameters to the `bind[]` attachment are grouped by pattern class. For each pattern class specified in the composition pattern, a set of parameters defining replacements for that pattern class and any of its template operations are grouped in <> brackets. |
| `<{className},` *`params`*`>` | The first parameter within a pattern class set is the name of the class that replaces the pattern class. This may also be a comma-separated list of class names, bounded by `{}` to denote a set. |
| `{className.` `opName}` | For each template operation defined for the pattern class, a replacing operation may be defined with the operation's name. Where this may be ambiguous - for example, when there are multiple classes replacing the pattern class, and there are some operations of the same name within those replacing classes - the operation name may be supplemented with its class name. Replacements for each template operation may also be a comma-separated list of operation names, bounded by `{}` to denote a set. |
| `{*}` | When specified as a replacement for a pattern class, this denotes that all classes within the input subject are replacements for the pattern class. When specifed as a replacement for a template operation, this denotes all operations within each replacing class are replacements for the template operation. |
| `{meta:` *`metatest`*`}` | The `meta` keyword, used inside `{}` denoting a set, denotes that a test against the metaproperties of elements determines their eligibility to replace the template. When specified as a replacement for a pattern class, class properties of every class within the input subject are examined against the test criteria. When specifed as a replacement for a template operation, operation properties of every operation within each replacing class are examined against the test criteria. In both cases, valid metaproperties and valid values for those properties, must be defined, as specified by the UML semantics. |

**Table 3:** `bind[]` **parameters**

There is considerable potential for further work in extending the capabilities of the parameters to the `bind[]` attachment. Sophisticated matching criteria for selection of replacement candidates for pattern classes and template operations are possible. This work should be done in conjunction with the work extending the rules for template specification previous discussed in "Further Potential for Template Rule Specification" on page 205.

We now look at an example of defining a binding specification for the observer composition pattern. As illustrated in Figure 100, the binding specification is:

- `S1.ClassA` is a replacement for pattern class `Subject`, with every operation that is a non-query operation replacing template parameter `_aStateChange(..)`.
- `S1.ClassB` replaces pattern class `Observer`
- `S1.ClassB.op2()` replaces `update()`
- `op3(ClassA)` and `op4(ClassA)` from `S1.ClassB` are supplemented with the pattern behaviour specified for `start(..,Subject,..)` and `stop(..,Subject,..)`, respectively.

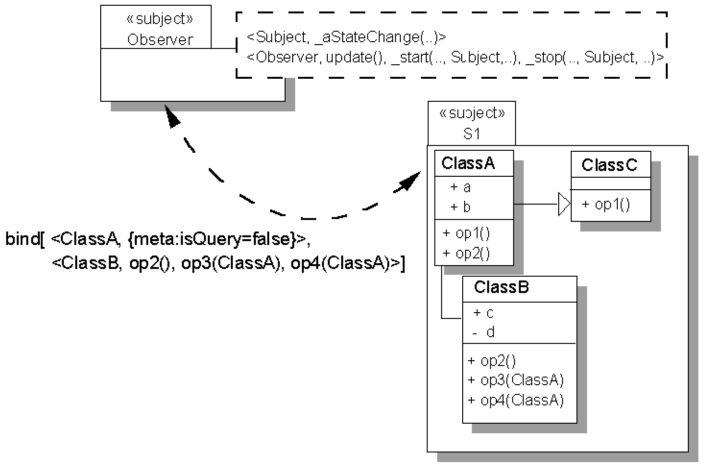

«subject»
Observer
<Subject, _aStateChange(..)>
<Observer, update(), _start(.., Subject,..), _stop(.., Subject, ..)>
«subject»
S1
ClassA
+ a
+ b
+ op1()
+ op2()
ClassC
+ op1()
ClassB
+ c
- d
+ op2()
+ op3(ClassA)
+ op4(ClassA)
bind[ <ClassA, {meta:isQuery=false}>,
<ClassB, op2(), op3(ClassA), op4(ClassA)>]

**Figure 100: Specifying Binding for Composition**

**Composition Output**

As illustrated in Figure 100, 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 pattern classes, each replacement class is supplemented with the properties (and behaviour) of the pattern class in the output subject. For example, in Figure 101, classes `ObserverS1.ClassA` has `Observer.Subject`'s properties. Where a pair of operations has been defined (e.g., `aStateChange(..)` and `_aStateChange(..)`) and referenced within the same scope in a composition pattern (that is, inside the same pattern class), and one is a template parameter for that class, composition applies merge operation semantics. For each operation substituting the template parameter, each reference to `_aStateChange(..)` is replaced by the suitably re-named substituting operation, and a new `aStateChange(..)` operation is also defined. Each operation's delegation semantics is realised by a new collaboration as specified within the composition pattern. Each pattern class referenced within an interaction diagram is also re-named as appropriate.

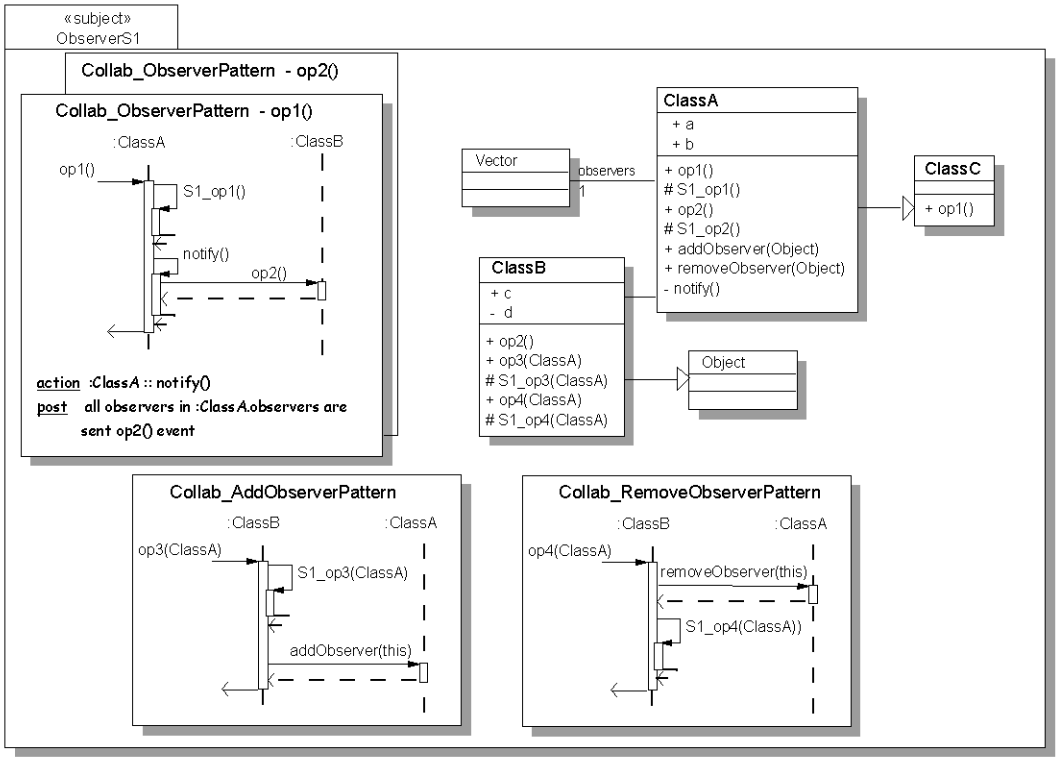


**Figure 101: Output from Composition with Pattern Subject**

In this example, the result of the composition is:

- Two non-pattern classes, `Vector` and `Object`, are defined in subject `Observer`, and are added unchanged to the result.
- `ClassA` is a replacement class for `Subject`, and so all non-template properties of `Subject` are added to `ClassA` in the result. Therefore, operations `addObserver(Object)`, `removeObserver(Object)` and `notify()` are added unchanged to `ClassA`. If there had been (an)other class(es) replacing `Subject`, then each of these operations would also be added to each replacing class.
- Within `ClassA`, the `{meta:isQuery=false}` bind parameter indicates that the set of operations replacing the `aStateChange(..)` template operation is selected by examination of the `isQuery` meta-property of all operations within `ClassA`. Those whose value for `isQuery=false` are added to the replacing set. In this case, operations `op1()` and `op2()` are non-query operations, and therefore both are added to the replacing set. Where multiple replacements are specified for

operations, each operation is supplemented with the behaviour defined within the pattern subject. This is specified in the result with both `op1()` and `op2()` having interaction diagrams realising the supplementary behaviour, with re-naming of the replacement operations in line with the delegation of corresponding operations semantics.

- Since `ClassA` is a `Subject` class, it also has an `observers` association with `Vector` added. Every class replacing `Subject` has such an association with `Vector` added (in this example, this is just `ClassA`).
- `ClassB` is a replacement class for `Observer`, and so all non-template properties of `Observer` are added to `ClassB` in the result. Since all properties of the `Observer` pattern class used in the `Observer` composition pattern are templates, this does not add any additional properties to `ClassB`.
- `ClassB.op2()` is defined as the replacing operation for `update()`. This means that it is added unchanged to the result, and called from the `notify()` operation in the observer pattern for both `ClassA.op1()` and `ClassA.op2()`.
- `ClassB.op3(ClassA)` and `ClassB.op4(ClassA)` are defined as the replacing operations for `start(..,Subject,..)` and `stop(..,Subject,..)`, respectively. These are valid replacements as `ClassA` is a `Subject`, and therefore the pattern has a valid `Subject` to work with. The interaction diagrams for `start(..,Subject,..)` and `stop(..,Subject,..)` are updated to realise the replacement operations.

Additional composition relationships between levels lower in the subject tree may be specified within a composition pattern, though only one subject in a particular composition context may have templates defined. Further research is required to assess the impact of merging multiple subjects where more than one subject contains template elements. It may be possible to relax this restriction, but without fully assessing the impact, the results are not defined, and therefore the restriction is applied in this thesis.

## 8.2. Composition Patterns Metamodel

The composition patterns model, at the specification level, differs from the UML templates model in two primary ways:

- Templates within a composition pattern are centred around pattern classes within a subject first, which (may) have additional template parameters defined.
- 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.

These differences are demonstrated at the meta-level in Figure 102, which is an extension to the metamodels defined in "5.3. Composition Relationship" on page 113, and "7.2. Merge Integration Syntax" on page 164.

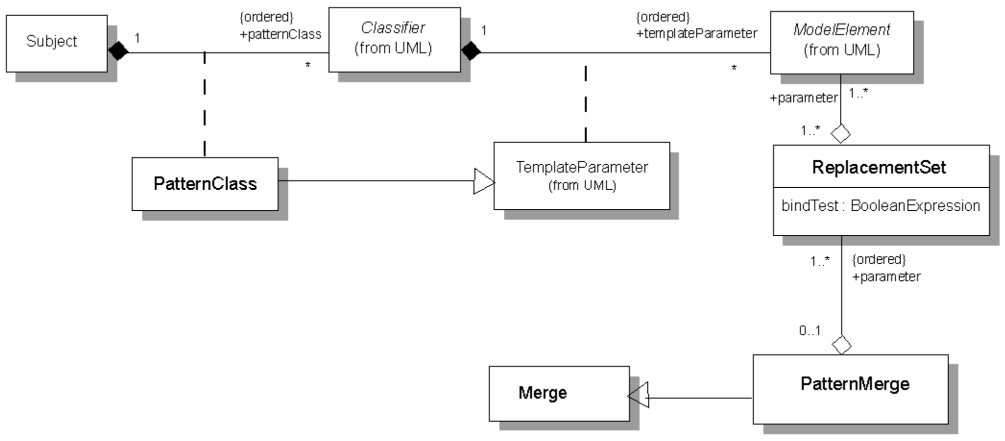


**Figure 102: Composition Patterns Metamodel**

*Subject Metaclass*

***Associations***

| | |
|---|---|
| *patternClass* | An ordered list of classifiers which are deemed to be pattern classes. Each parameter is a dummy classifier designated as a placeholder for (a) real classifier(s) to be substituted during composition. |

*PatternClass Metaclass*

The `PatternClass` metaclass defines the relationship between a subject and its pattern classes. By definition, a subject with a relationship to at least one pattern class is a composition pattern.

*PatternMerge Metaclass*

A pattern merge is a kind of merge integration that also handles merging elements with template specifications. It is sub-classed from the `Merge` metaclass because it also conforms to merge integration semantics.

***Associations***

| | |
|---|---|
| *parameterSet* | The replacement set specification for elements that replace the corresponding template elements. |

*Replacement-Set Metaclass*

The composition patterns model supports the specification of multiple replacements for each pattern class and template parameter. The `ReplacementSet` metalass defines a multiple replacement set.

***Attributes***

| | |
|---|---|
| *bindTest* | Boolean expression checking values of properties of each input element (that is, each element contained in the subject composed with the pattern, or contained in the subject's classes) to decide whether that element is to be substituted for the template element. Any combination of properties of elements may be used for test purposes. This attribute may be null, where an explicit list of elements is defined for the replacement set. |

***Associations***

| | |
|---|---|
| *parameter* | A list of elements that replace a template. This list may be based on the specification within the replacement set, or an explicit list of elements. |

**Well-Formedness Rules**

[1] Only one subject involved in a single contextual composition relationship may contain template elements.

[2] Only a contextual composition relationship may have pattern match integration defined - that is, when the composition relationship is between two or more subjects.

[3] All templates must have at least one replacement defined.

[4] Replacements defined for pattern classes must be contained within the subject input to the composition.

[5] Replacements defined for a template operation must be contained within a replacement for its owning pattern class.

## 8.3. Chapter Summary

This chapter describes how patterns of cross-cutting behaviour can be decomposed into a separate design model, and designed without explicit reference to any design elements the behaviour may cross-cut. This is achieved using a combination of merge integration semantics and UML templates, in what is described in this chapter as *composition patterns*. A composition pattern is a design subject with at least one pattern class defined. Within a pattern class, template operations may be referenced. These template operations represent operations that replace them at composition time, and that specify behaviour to be supplemented with pattern behaviour as defined in the composition pattern (that is, cross-cutting behaviour). A composition relationship with merge integration may be defined between a composition pattern and subjects requiring the pattern behaviour. A `bind[]` attachment to such a composition relationship defines the replacement elements for the pattern classes and template operations.

Composition patterns require extensions to the subject-oriented design metamodel which are described here. These extensions are based on specification of pattern classes and template parameters within composition patterns, and on specifying the replacement classes and operations for the templates with a composition relationship.

Having described in detail the syntax and semantics of the subject-oriented design model in the previous chapters, the next chapter applies the model to the motivating example from "Chapter 2: Motivation" on page 11. This example was based on the design of a simple software engineering environment (SEE) for programs consisting of expressions. This problem is redesigned using subject-oriented design, demonstrating improvements to the problems motivating the work. A further example of the application of subject-oriented design is demonstrated and evaluated in "Chapter 10: Case Study and Evaluation" on page 225.

# Chapter 9: Applying the Subject-Oriented Design Model

This thesis proposes a new approach to object-oriented design based on providing additional means of decomposing design artefacts by matching the structure of features and other user-level concerns, encapsulating those concerns within the design. The approach addresses the structural misalignment between requirements, designs and code that is the cause of considerable difficulties with the use of design as described in "Chapter 2: Motivation" on page 11. At the core of this model is composition specification, that allows differences in views of overlapping concepts within different design subjects to be identified and resolved, and supports the understanding of the design as a whole by integration.

In this chapter, the small, motivating example described in "Chapter 2: Motivation" on page 11 is re-designed based on the subject-oriented design model. By applying the model to the construction and evolution of the expression SEE, it is illustrated that the design addresses the misalignment problem, and achieves better, more flexible system design.

## 9.1. SEE System Design, Version 1.0

Revisiting the motivating example, the requirements specification stated: "The required SEE supports the specification of simple expression programs. It contains a set of tools that share a common representation of expressions. The initial tool set should include an *evaluation* capability, which determines the result of evaluating expressions; a *display* capability, which depicts expressions textually; and a *check* capability, which optionally determines whether expressions are syntactically and semantically correct. The SEE should permit optional logging of operations". For further details of the grammar and abstract syntax tree implementation of expressions, see "2.2. Example: Software Engineering Environment" on page 19.

To align design with requirements, a design subject per feature identified in the requirements specification is defined. Thus, as illustrated in Figure 103, there is a *Kernel* subject supporting the representation of expressions; an *Evaluate* subject; a *Check* subject; a *Display* subject; and a subject, *Log*, responsible for logging of operations.

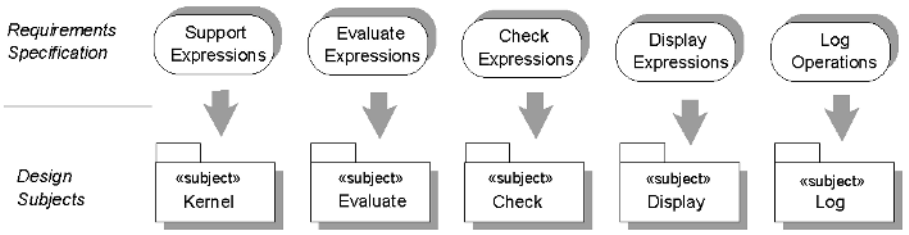


**Figure 103: Design Subjects for SEE**

**Design Subjects**

Each of the structure diagrams for the chosen design subjects is now illustrated[1].

*Kernel*

The Kernel subject is illustrated in Figure 104. As in the original design, expressions are represented as abstract syntax trees. Notice, however, that the kernel design subject only defines the AST classes and their primitive accessor methods - it does not tangle support for any of the required SEE features with the expression representation.

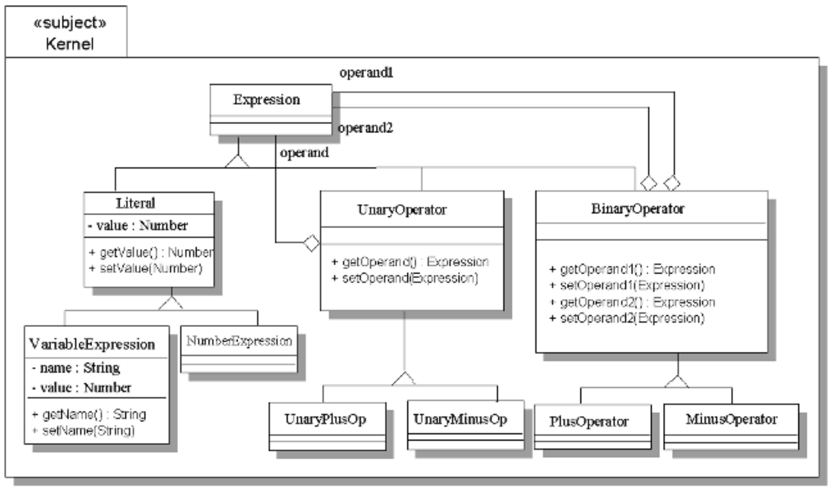


**Figure 104: Kernel Subject Class Diagram**

1. The interaction diagrams which complete the design of each subject are not illustrated, but are assumed to exist. As described in “Impact of Merge on Collaborations” on page 195, all interaction diagrams are added to the output, unchanged.

*Check* The design for the check subject, illustrated in Figure 105, maintains a view of expressions relevant for checking purposes, with only those properties that are required by the checking behaviour.

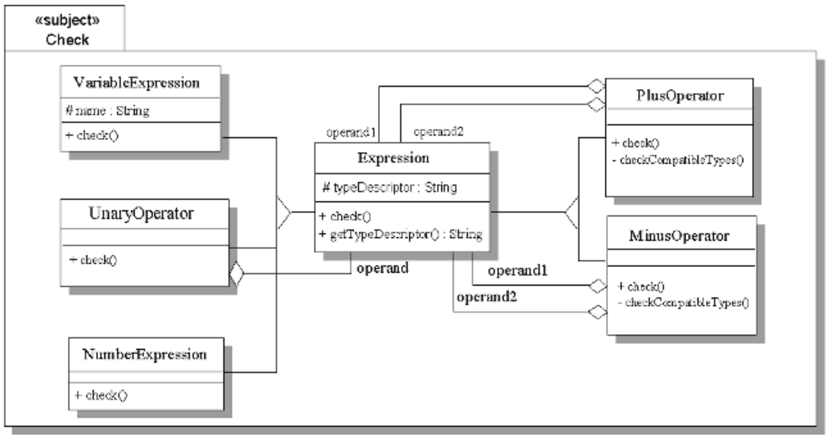


**Figure 105: Check Subject Class Diagram**

*Evaluate* The design for the evaluate subject, illustrated in Figure 106, maintains a view of expressions relevant for evaluation purposes, with only those properties that are required by the evaluation behaviour.

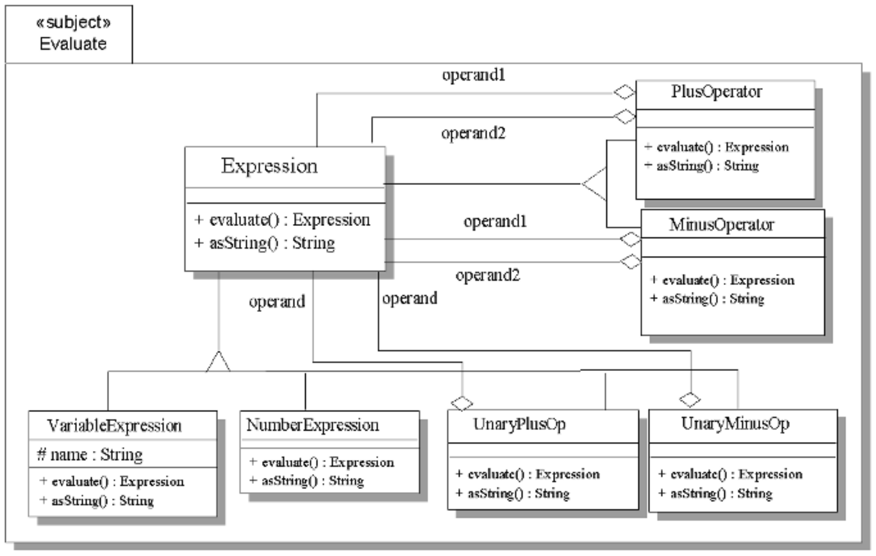


**Figure 106: Evaluate Subject Class Diagram**

*Display* The design for the display subject, illustrated in Figure 107, maintains a view of expressions relevant for display purposes, with only those properties that are required by the display behaviour.

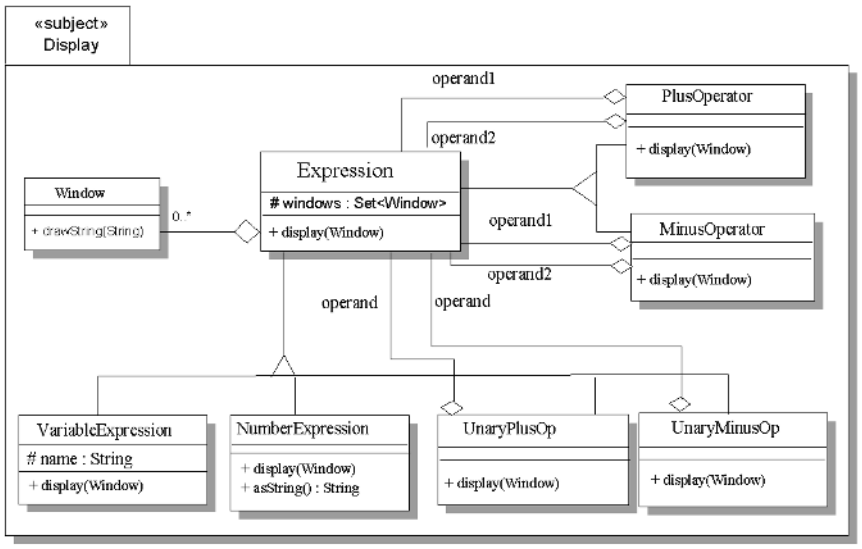


**Figure 107: Display Subject Class Diagram**

*Log*

The design for the logger subject, illustrated in Figure 108, can be designed independently of the operations to be logged, as the ability to log operations is not particular to expressions or ASTs. An interaction specification for logging behaviour is included in this subject, as its specification is central to how the subject will be composed with other subjects. This is not the case with interactions in the other SEE subjects. Logging behaviour impacts all operations in the SEE, and therefore, the Log subject is designed as a *composition pattern* (see “Chapter 8: Composition Patterns” on page 198).

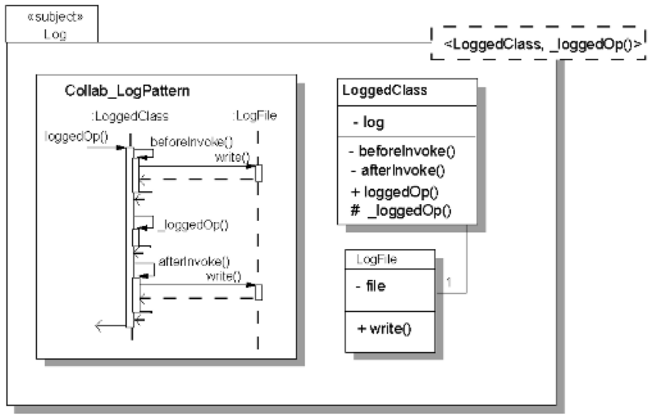


**Figure 108: Log Subject Design**

**Characteristics of SEE Design Subjects**

These design subjects illustrate some important characteristics of subject-oriented design.

*Features are Encapsulated*

First, the kernel, check, evaluate and display subjects realise and encapsulate their respective SEE tools in a standard object-oriented manner, with appropriate properties - attributes and operations - in each of the AST classes. Given that requirements are structurally different to object-oriented designs, and each subject is designed using an object-oriented language, there is unavoidable scattering of tool support across classes within each subject. Nonetheless, encapsulation is achieved by each subject *as a whole*. This provides clear alignment of the design to the requirements, as each subject represents the design of a particular feature in total, and contains no reference to any other feature; all cross-feature interactions are specified by means of composition relationships. Encapsulation of the logger feature also avoids tangling of logger functionality with the rest of the design.

*Features have Different Views*

A second important feature of this subject-oriented design approach is that each of the subjects specifies its own view of overlapping design elements. For the SEE, the AST structure of an expression is manifested in each subject, except the Logger subject. Yet each subject defines a slightly different view of the AST class hierarchy; for example, the Check subject does not define the `BinaryOperator`, `UnaryPlusOp`, and `UnaryMinusOp` classes in its hierarchy, as they are not affected directly by the checking methods. Similarly, the Evaluation subject and the Display subject do not include the `BinaryOperator` and `UnaryOperator` classes. The designers of the individual subjects need not be concerned about these differences, as identification and resolution of any differences is supported by composition relationships. This increases the amount of concurrent design that is possible. It also enables each subject to include whatever model of AST it finds most appropriate to its task, rather than requiring commitment to a single AST definition. This property helps to improve the individual subjects, to insulate each designer from the effects of changes in other subjects, and to eliminate coupling across subjects.

*Cross-Cutting Feature Designed Independently*

The Logger subject illustrates another interesting feature of subject-oriented design. The SEE requirements specification imposed a requirement for optional logging of operations. The ability to log operations is not particular to expressions or ASTs, however, so the Logger subject can be designed independently of the operations to be logged (see Figure 108). Composition relationships will establish connections between the SEE subjects (or any others) and Logger, thereby specifying exactly when logging is to take place. This approach has the advantage of separating design of logging from that of the

SEE, addressing the tangling problem that manifests itself primarily in the behavioural specifications for operations that are to be logged (see Figure 8 on page 25). It also results in a subject that is generally reusable for any application that requires logging of operations.

## Composition Relationships for Design Synthesis

Taken together, the collection of design subjects described in the previous section defines a *family* of SEEs. That is, the set of features encapsulated in the individual design subjects can be integrated in a number of different combinations - e.g. some versions of SEEs might include the evaluation feature, but not the checking feature, and some might include logging while others might not. This ability to "mix-and-match" features is another benefit of subject-oriented design. It requires only the specification of composition relationships among whatever design subjects are to be included in any given member of the SEE family. For example, Figure 109 illustrates the composition relationships required to define a SEE that includes the features display, check and evaluation. The composition relationship with merge integration specified between the kernel, check, evaluate and display subjects indicate `match[name]` correspondence.

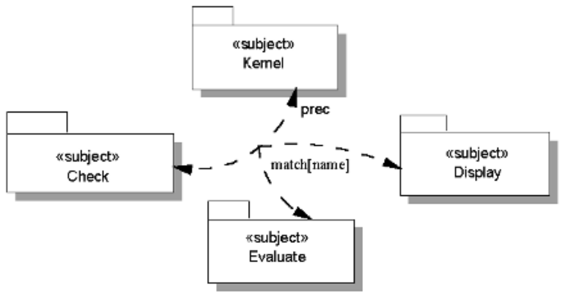


**Figure 109: Composition Relationship for Merging SEE Subjects**

A `match[name]` correspondence with *merge* integration means that in a composed design subject, classes and attributes having the same name in different design subjects would appear only once, and operations having the same name would be aggregated. The composition designer has also deemed that, should a conflict occur in any corresponding elements, the Kernel subject contains the specifications that should appear in the result. This composition relationship is complete and sufficient to specify the composed design as illustrated in Figure 110 and Figure 111.

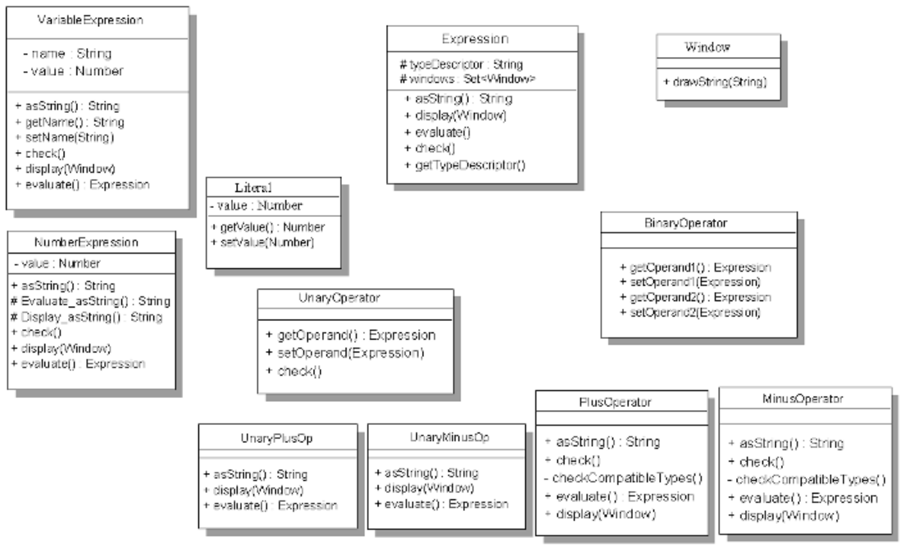


**Figure 110: Composed SEE Design (Class Details Only)**

This composition to a complete design is useful, particularly to a developer attempting to understand the full semantics of a composed design and all the ramifications of a set of composition relationships. In this case, there is an example of how the one simple composition relationship illustrated in Figure 109 has some undesirable behaviour. Merging corresponding operations means that on execution of any one of the operations, each of the corresponding operations is executed. Because of the matching by name specification of the composition relationship, the `asString()` operation from `Evaluate.NumberExpression` corresponds with the operation of the same name from `Display.NumberExpression.` Both of the operations are added to the result, and both are executed when there is a call to `asString()`. However, these operations provide the same service in that a string representation of the class is returned, and so it does not need to be executed twice. To avoid this behaviour, an additional composition relationship may be added within the context of the relationship in Figure 109, indicating that one of the `asString()` operations overrides the other.

As illustrated in Figure 111, all of the associations and generalizations are added to the result. Where there are associations of the same name, they are deemed to be corresponding, and therefore only one representative association is added to the result. Nonetheless, there are some redundant associations and generalizations as a result of the differing generalization hierarchies designed for each of the different subjects. For example, the `Evaluate` subject did not generalize the `PlusOperator` and `MinusOp-`

erator to a BinaryOperator class, as the evaluation behaviour is different for both operators. In Evaluate, two associations each were added relating to the Expression class for the operands of plus and minus. However, in the Kernel subject, these were generalized to a BinaryOperator where these associations were added once. Since Evaluate and Kernel are merged in this example, all the associations are added. Similarly for the generalization relationships in this case, each of the generalization relationships are added to the result. which means that, for example, PlusOperator inherits from Expression both directly, and through BinaryOperator. This is a design equivalent to *flattening* behaviour in merging code subjects in subject-oriented programming. In subject-oriented programming, each class is fully expanded to include all the elements from its superclasses prior to integration. Instead of flattening the design elements, the subject-oriented design model adds all the generalization (and association) relationships to the result. Flattening the output of design composition yields the same result as flattening the output of code composition as in subject-oriented programming.

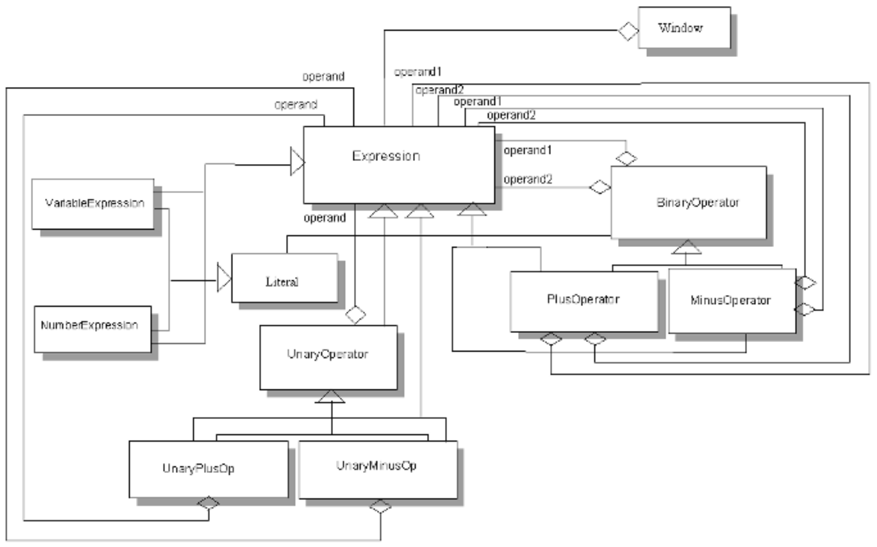


**Figure 111: Composed SEE Design with Relationships**

Not surprisingly, the fully composed design has the scattering and tangling characteristics of the original SEE design depicted in Figure 3 on page 21. All of the requirements are scattered across the design, and it is difficult to identify the exact elements that support a particular requirement. A single design class has support for multiple requirements tangled up within it. It is therefore considered to be useful only for the designer who needs to understand the design as a whole to work with the composed design. In general, it is simpler to work with, understand and explain the input subjects that support a single requirement.

Producing an SEE that excludes any of the features is equally simple to producing an SEE with all the features - the subject supporting the excluded requirement is therefore excluded from the composition relationships. Because each requirement is encapsulated in a separate subject, removal of a feature does not impact the design of any other feature.

## Composition Pattern

The "generally reusable" property of the Logger subject presents a good scenario for the use of composition patterns. Figure 108 illustrates the design of logging functionality using UML templates as placeholders for any operation requiring logging. An example of merging this subject with another subject (that is, a small extract from the `Kernel` subject with one operation) is illustrated in Figure 112.

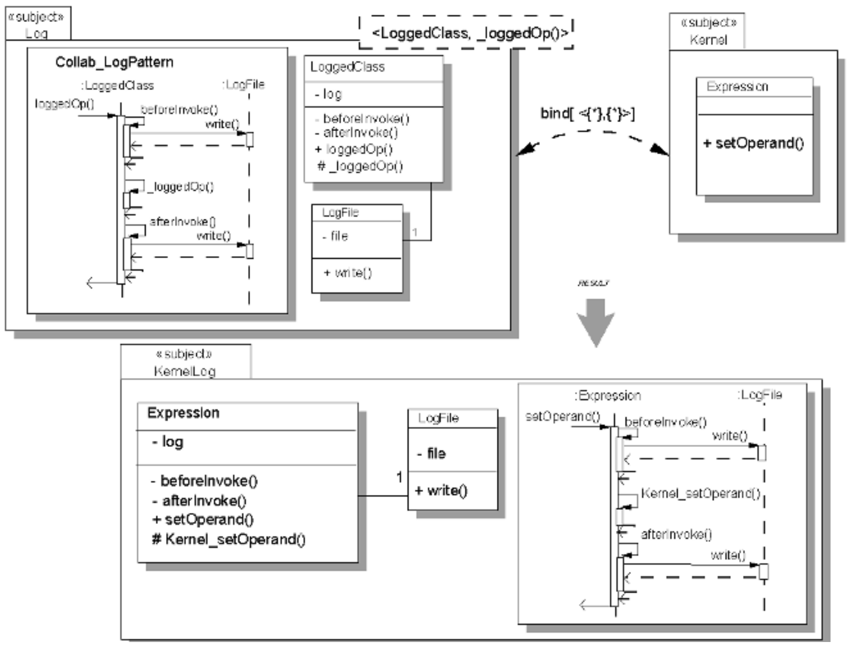


**Figure 112: Applying Composition Pattern for Logging**

In this example, the (partially illustrated) `Kernel` subject is merged with the `Log` subject containing a pattern class with an operation template. The `<{*},{*}>` parameters of the `bind` annotation to the composition relationship indicate that all classes in the merging subject, and all operations within those classes, should (separately) replace the pattern class and template operation, respectively. A collaboration is added for each operation indicating the changed behaviour as a result of the merge with logging functionality. This is illustrated for the `setOperand()` operation in Figure 112. In the output subject, the new interaction specifies that a call to the `setOperand()`

operation means that `beforeInvoke()` is executed before execution of `setOperand()`, and `afterInvoke()` is executed immediately afterwards.

An interesting example of the usefulness of separate design and composition of subjects is in the design of the logger. In the original logger design, two methods, `turnLoggingOn()` and `turnLoggingOff()`, had to be included to support this feature. The approach to the optional nature of logging is to include or exclude the Logger subject from compositions depending on whether or not logging is required. This approach has the benefit of not requiring any modifications to the design subjects.

For full details of the composition patterns model see "Chapter 8: Composition Patterns" on page 198.

**Producing Code from the Design**

This chapter has shown how subject-oriented design aligns with requirements. There are two approaches to aligning this design with code. The first approach is to code each individual design subject as a code subject in the subject-oriented programming paradigm, and then compose the code subjects with a *composition rule* [Ossher et al. 1996] derived from the composition specifications in the design. The second approach is for the designers to construct an integrated design, and then write standard object-oriented code based on it. In either case, however, the two-way alignment of subject-oriented design supports the realisation of one of software design's primary purposes - to bridge the gap between requirements and code. The first approach is preferred, however, because it results in code that is directly aligned with requirements, and that therefore has the same properties of traceability, and especially, evolvability, described earlier for subject-oriented designs.

## 9.2. Evolving the SEE System Design

The design of the SEE from "SEE System Design, Version 1.0" on page 22 suffered from the problem that what appeared to be simple, additive changes ended up being pervasive and invasive - See "Evolving the SEE System Design" on page 29. Specifically, clients requested the inclusion of different forms of optional checking, thus rendering the check feature a "mix-and-match" capability. The solutions considered either resulted in combinatorial explosion of classes (using a non-invasive, sub-classing approach), or required invasive changes to all of the AST classes (retrofitting design patterns). The subject-oriented design avoids all of these problems. Each different kind of checking is designed in a separate subject. Effecting the change request simply requires the definition of two new subjects: one to support the design of a def/use checker, and one to support verifying conformance to local naming conventions.

Selective use of composition relationships permits designers to decide what kind(s) of check(s) are to be performed in any particular system produced from the design. For example, in Figure 113 all of the checking subjects (partially represented) are included in the composition, with the resulting behaviour specification indicating that any `check()` operation results in each of the three kinds of checking being executed.

This example illustrates the general point that subject-oriented design facilitates *additive* rather than *invasive*, changes, significantly increasing the ease of system evolution.

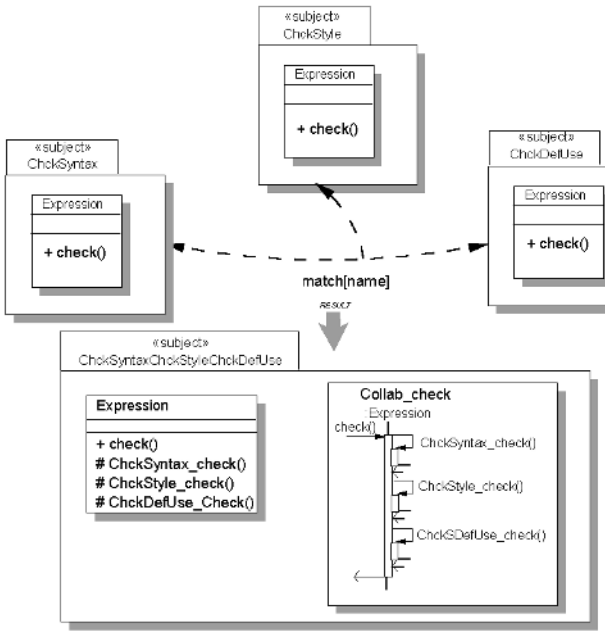


**Figure 113: Evolving SEE with New Check Requirements**

## 9.3. Chapter Summary

This chapter revisits the design of the SEE, using the subject-oriented design model. The approach illustrates how the structural misalignment between requirements, design and code can be solved by the encapsulation of features in design subjects. In comparison with the design of the SEE illustrated in "Chapter 2: Motivation" on page 11, the subject-oriented design demonstrates how scattering and tangling properties have been removed. Individual design subjects encapsulate the design of their own requirement, and may have different specifications of common concepts.

Support for integration of overlapping concepts, even where there are differing specifications is achieved with the specification of composition relationships. Composition relationships supporting the composition of the SEE design subjects are illustrated and discussed. A composition relationship with merge integration between the kernel, evaluate, display and check subjects is illustrated. The output of this composition illustrates that only one other composition relationship is required to handle the duplication of the `asString()` operation, which appears in two different subjects. This illustrates how an analysis of the output of the composition assists the designer in verifying composition relationships. In many cases, where the designer is familiar with the details of input design subjects, the initial composition may include the exceptions to a contextual composition relationship that governs the composition of all the components of the input subjects.

It is also illustrated how generally reusable subjects may have composition patterns defined, simplifying the process of composition specification. A composition pattern to support logging functionality illustrates how logging operations may be designed with reference to template operations, as opposed to the actual operations to be logged. This supports the simple composition of multiple subjects with operations to be logged, as illustrated.

Scattering of the design for requirements across a full system design, and tangling of the design for multiple requirements in a single design element have been illustrated to be the root of many of the difficulties with standard object-oriented designs. These properties make the designs difficult to understand, difficult to change and difficult to reuse. Removal of scattering and tangling properties therefore eases the difficulties that they cause. Even in a small example such as the SEE, separation of the support for different requirements makes it easier to trace the design for each of the requirements. In particular, the design of logging functionality without reference to any expression operations makes this subject reusable in any domain where the design includes operations.

# Chapter 10: Case Study and Evaluation

This chapter demonstrates the use of the subject-oriented design model using a Library Management System case study. Throughout, any decision that is available uniquely because of the application of the subject-oriented design model is highlighted. Differences with possible alternatives using standard UML are evaluated.

The case study, though relatively small compared with most software products, nonetheless illustrates the capabilities of the subject-oriented design model. An initial system is designed using different design subjects for different requirements. Both functional and cross-cutting requirements are included, with a demonstration of how their composition may be specified, and the output of composing different subjects. The chapter then shows how changes to the borrowing rules, that demonstrate the evolution of the library system, may be designed separately and composed with the existing system. Functional holes in the system design, the kind likely to be found during system test, are encountered and may also be designed separately and composed with the existing system. The case study demonstrates the strengths of the subject-oriented design model, but also highlights some interesting weaknesses.

## 10.1. Requirements Specification

A library management systems manages the resources within a university library, and the activities relating to those resources. The subset of such a system examined in this case study is the management of books and periodicals. This is essentially managing their ordering and physical locations within the library, and managing their borrowing and return. A full library management system would be a far larger system, probably including, for example, management of client and vendor information and history. Architecturally, the portion of the system included in the case study may be seen as the business model layer, in a "layered architecture" ([Shaw & Garlan 1996]) separating the user interface from the objects that support the base library concepts.

**Functional Requirements**

A library's resources are multiple copies of both books and periodicals. Users of the library are librarians and borrowers, but only librarians use the library management system. The actors and their uses of the library management system are:

| | |
|---|---|
| Actors | Librarians, Staff, Students, Public |
| Library resources: | Multiple copies of books and periodicals |
| Uses of system: | Add/remove library resource |
| | Order library resource |
| | Search for library resource |
| | Borrow/return library resource |
| | Pay late return fine |

*Add library resource*

The librarian may add library resources to the library management system (LMS). These may be additional copies of an existing title, or copies of a new title. The following information is maintained by the LMS:

- The ISBN, title, author(s) and publisher information of the title
- The staff member(s) and course number(s) that use the title
- The library-assigned numbers and physical locations of all copies

*Remove library resource*

The librarian may remove all copies of a title from the LMS. This is only possible if all borrowed copies of the title have been returned.

*Order library resource*

The librarian may record an order for multiple copies of a resource through the LMS. The following information is maintained:

- The ISBN, title, author(s), and publisher information of the title
- The number of copies ordered
- The vendor information and date of ordering

*Search for library resource*

All users of the library may search for the physical location of copies of a particular title. The search may be on ISBN, title or author information. Wildcard searches are required, which may result in multiple items returned from the search.

*Borrow library resource*

The only library resources which may be borrowed are copies of books. Restrictions exist for different kinds of borrowers:

- Librarians may borrow any number of books
- Staff may borrow up to ten books
- Postgraduate students may borrow up for eight books
- Undergraduate students may borrow up to four books
- Members of the public may borrow up to two books

*Return library resource*

The librarian may record when borrowers return books. If the on-loan period is greater than the allowed period for the type of borrower, then a fine is imposed as follows:

- Librarians may borrow their books for a period of two months, staff for two months, postgraduate students for six weeks, undergraduate students for two weeks, and members of the public for one week.
- Some titles have their own time restrictions on amount of time copies may be borrowed which take precedence over the period restrictions for type of borrower.

*Pay late-return fine*

The librarian may record the payment of fines by the borrower.

**Technical Requirements**

It is required that the services for managing resources are available concurrently. However, those services that change the objects (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.

## 10.2. Design with Structural Matching to Requirements

This section discusses the options for decomposing the design of the library problem domain for potentially different design teams. The structural mismatch of the requirements specifications with object-oriented specifications of the library concepts is illustrated. A design of the system using the decomposition capabilities provided by subject-oriented design is presented.

**Decomposition**

This thesis does not include a discussion on how the subject-oriented design model impacts the software development process, but recognises that this is an important area for future work. Therefore, for the purposes of this case study, some assumptions are made as to the "process" a development project manager may follow to assign tasks to different people/teams.

Without the benefit of subject-oriented design, a project manager must look at the design domain as well as the requirements domain in order to carve up the design domain area appropriately. Given that only one person/team may work on an object-oriented class at one time, it is reasonable to assume that a project manager would attempt to group classes with group(s) of requirements as much as possible. To achieve this, it is likely that a project manager and lead designer(s) would meet (with, possibly, white board aids) to attempt a high-level assessment of a workable division of classes. Such an effort is likely to result in the information illustrated in Figure 114.

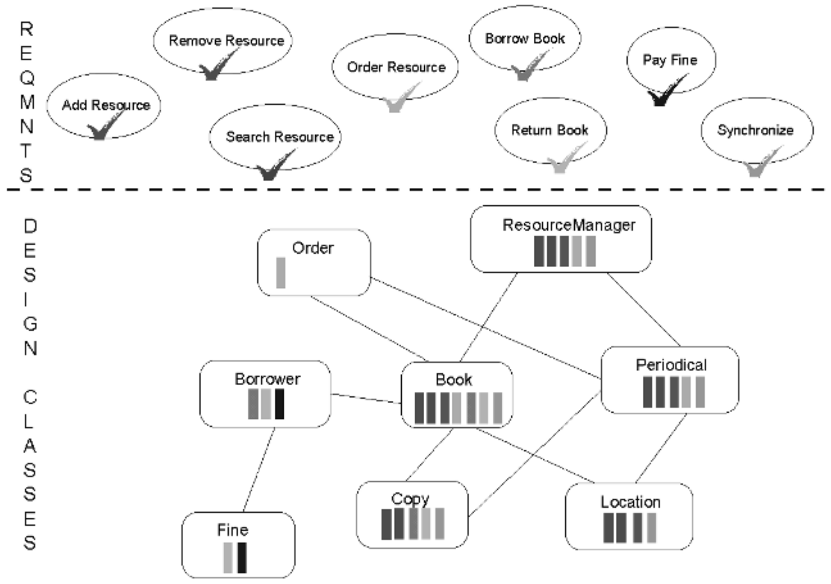


**Figure 114: Initial Assessment of Project Classes and Tasks**

The efforts of a development project manager and lead designer(s) as illustrated in Figure 114 demonstrate the scattering and tangling properties that are at the core of the motivation for the research described in this thesis. Any attempt to divide up the work by requirement leads to overlapping usage of classes, requiring complicated scheduling and critical path management. Any attempt to divide up the work by classes leads to a need for designers to communicate for the purposes of clear interface definitions. Where any requirement "colour" (Figure 114) touches multiple classes, the interface between those classes must be clearly defined for that requirement. Communication between designers costs time.

On the other hand, with subject-oriented design, it appears at this stage that a clean division of work may be achieved by assigning one design subject for each requirement.

**Design Subjects**

A project development manager using the subject-oriented design model need not attempt to anticipate the internals of the design for the purposes of division of the tasks. So again, making some assumptions as regards "process", the manager may decide on a one-to-one structural matching of the requirements with the design models, yielding the separate design subjects illustrated in Figure 115.

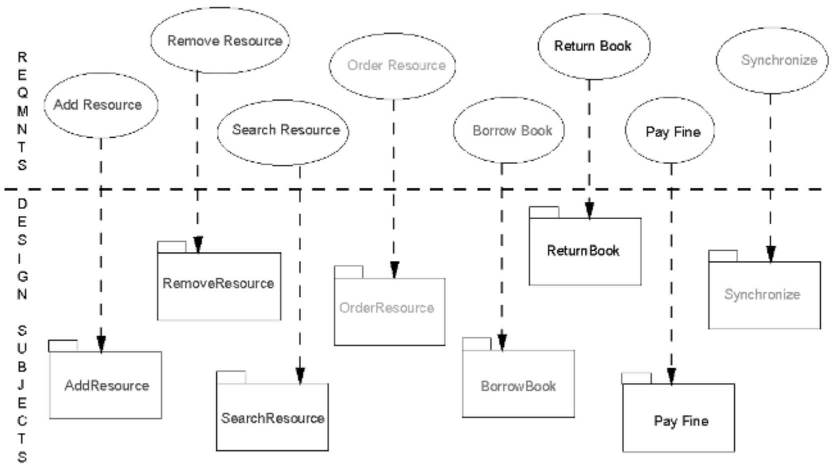


**Figure 115: Division of Tasks into Design Subjects**

It is, however, likely that the development manager would meet with the senior designer(s) for the purposes of estimating the size of the task of designing each subject. This information is likely to impact the size of teams working on each one. For the purposes of this case study, we assume that separate teams work on different subjects, and that the number of designers in each team is not relevant for the purposes of assessing the subject-oriented design model.

The following subsections illustrate some details of the designs of each of the design subjects. It is not, however, the intent of this chapter to discuss detailed motivation for choosing and naming particular design elements, or to discuss individual design decisions for each subject. It is assumed that standard design practices and decision-making processes apply inside each individual design subject. The following subsections will, however, point out any interesting decisions that may impact subsequent composition of those subjects. A further assumption with this case study is that the "Actor" management is catered for outside this case study. That is, information relating to library staff, academic staff, postgraduates, undergraduates and members of the public is maintained outside the library management business model.

*Add Resource* The `AddResource` subject handles the structural and behavioural implications of storing books and periodicals in the library. In the structural design illustrated in Figure 116, the commonalities of `Book` and `Periodical` are abstracted to a `Resource` class, from which each of them inherits. The designer of this subject deems that it is appropriate for `Resource` class to be abstract.

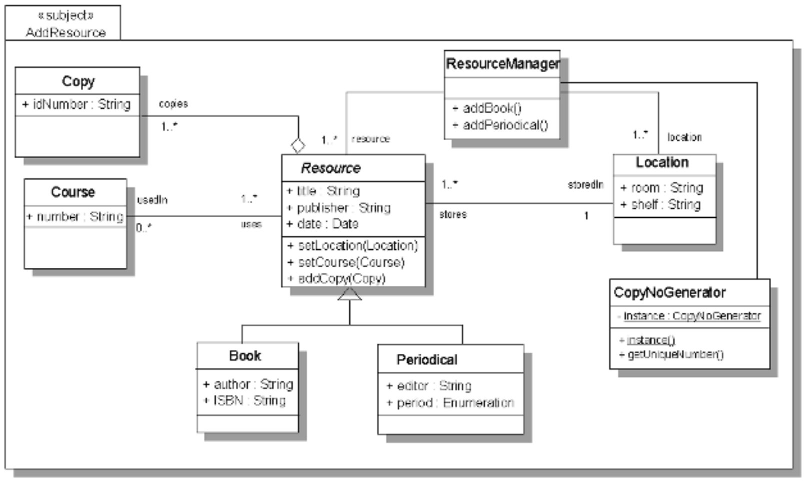


**Figure 116: Add Resource Class Diagram**

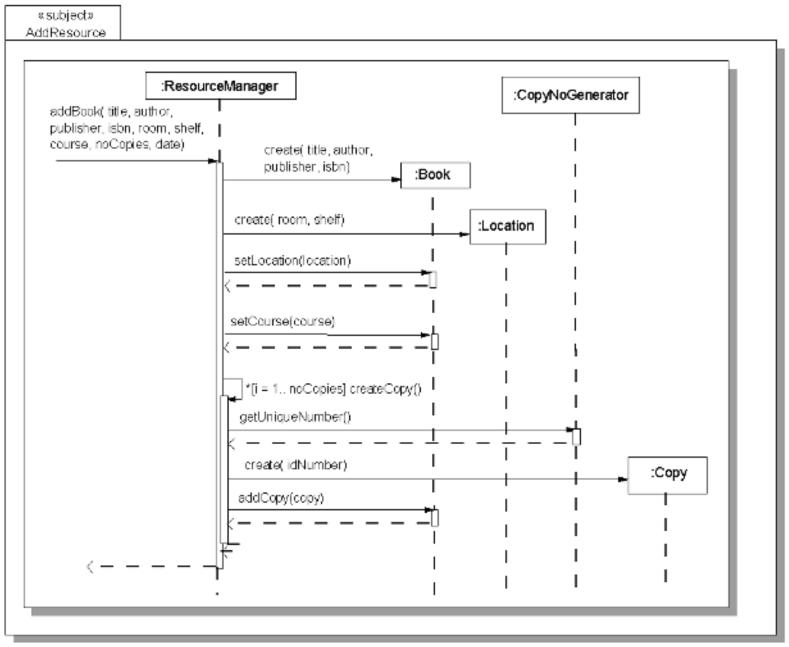


**Figure 117: Add Resource Interactions**

The behavioural interactions of adding a resource to the system is illustrated in Figure 117. This design illustrates the interactions for adding a `Book` instance. The interactions for adding a periodical are similar and since they provide no additional points of interest to the design, they are not illustrated.

*Remove Resource*

The `RemoveResource` subject handles the structural and behavioural implications of removing books and periodicals from the library. Figure 118 illustrates the structural and behavioural design. Removing a book and removing a periodical are the behaviourally the same, and so the designer of the `RemoveResource` subject need only reference and use the `Resource` class. This is a feature of the subject-oriented design model, where a designer need only specify details of elements that are relevant for the particular requirement under design.

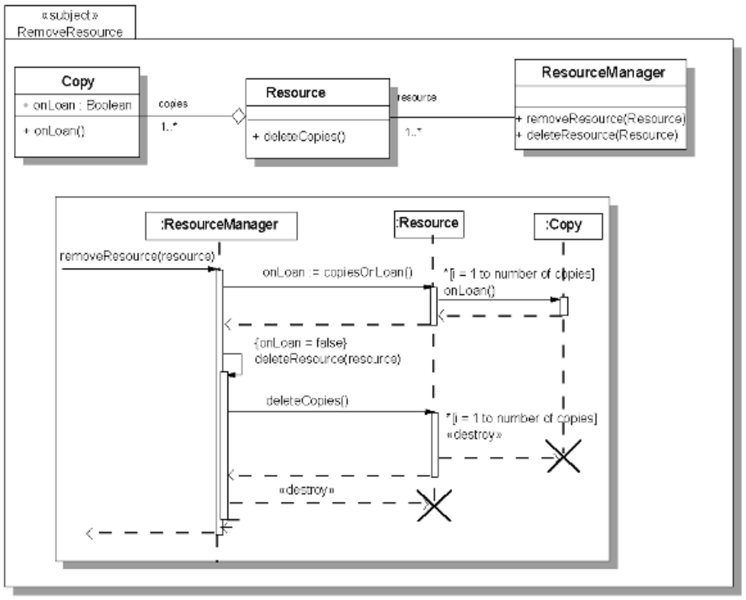


**Figure 118: Remove Resource Class Diagram and Interactions**

Here we can also see a difference in the specifications of the `Resource` classes in the `AddResource` and `RemoveResource` subjects. In the `AddResource` subject, `Resource` was defined as being abstract, while here in the `RemoveResource` subject, it is not. Here, the designer has no reason to set the `Resource` class as being abstract. Designers working independently will not communicate this difference of opinion, and therefore, in a composition of these two subjects, the details of the `Resource` classes will clash, requiring reconciliation. The subject-oriented design model provides the means to resolve this conflict, discussed in "Composing Resource Management Subjects" on page 237.

*Order Resource* Figure 119 and Figure 120 illustrate a structural and an interaction design, respectively, for ordering resources for the library. The designer works with only those elements that are relevant for ordering resources.

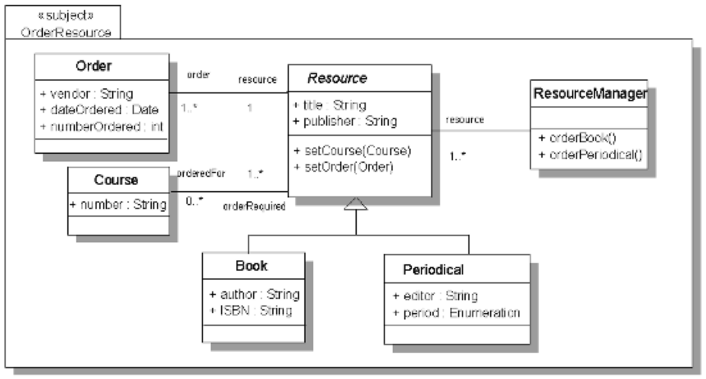


**Figure 119: Order Resource Class Diagram**

However, comparing this design with the design of the `AddResource` subject highlights a weakness with the subject-oriented design model. The `AddResource` class diagram states that a resource must be stored in one location. However, the `OrderResource` design uses the `Resource` class to store the on-order information as well, and therefore it is not stored anywhere until it has been received. This designer does not even consider locations as they are not relevant for ordering. This is an example of where knowledge of the domain is required to assess the impact of joining constraints from different models. The impact of this on composition is discussed in “Composing Resource Management Subjects” on page 237.

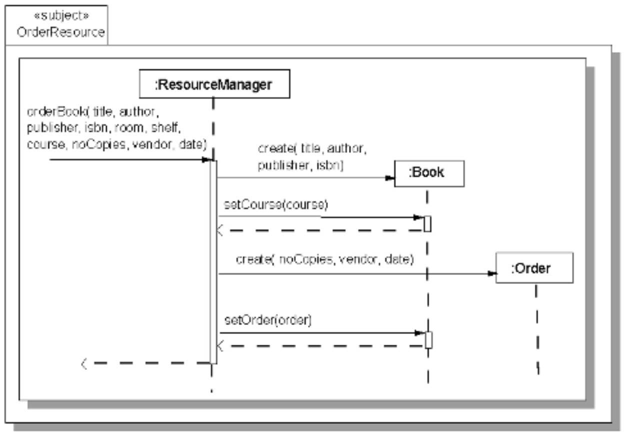

«subject»
OrderResource
:ResourceManager
orderBook( title, author, publisher, isbn, room, shelf, course, noCopies, vendor, date)
create( title, author, publisher, isbn)
:Book
setCourse(course)
create( noCopies, vendor, date)
:Order
setOrder(order)

**Figure 120: Order Resource Interactions**

There is one more interesting point to note with the `OrderResource` subject, that the subject-oriented design model does cater for. The interaction diagram illustrated in Figure 120 shows a `setCourse()` operation that sets the `orderedFor` relationship with `Course`. A look at the `AddResource` subject shows that here also is a `setCourse()` operation that sets the `usedIn` relationship with `Course`. Here are two operations with the same name that are essentially different operations, and therefore the composition designer must cater for this. How this is achieved is discussed further in "Composing Resource Management Subjects" on page 237.

*Search for Library Resource*

A structural design for searching for library resources is illustrated in Figure 121. This design does not highlight any additional interesting points for the subject-oriented design model.

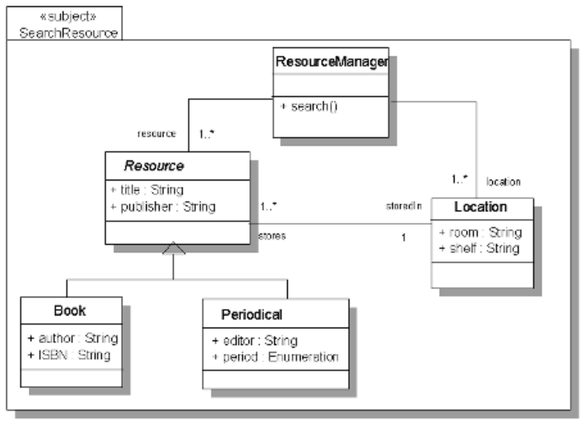


**Figure 121: Search Resource Class Diagram**

*Borrow Library Book*

Figure 122 and Figure 123 illustrate a structural and an interaction design, respectively, for borrowing library books. Since only books may be borrowed, the designer need only reference and include elements relating to books. This design does not highlight any further additional interesting points for the subject-oriented design model.

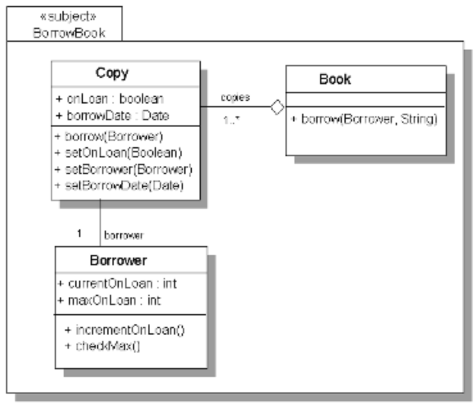


**Figure 122: Borrow Book Class Diagram**

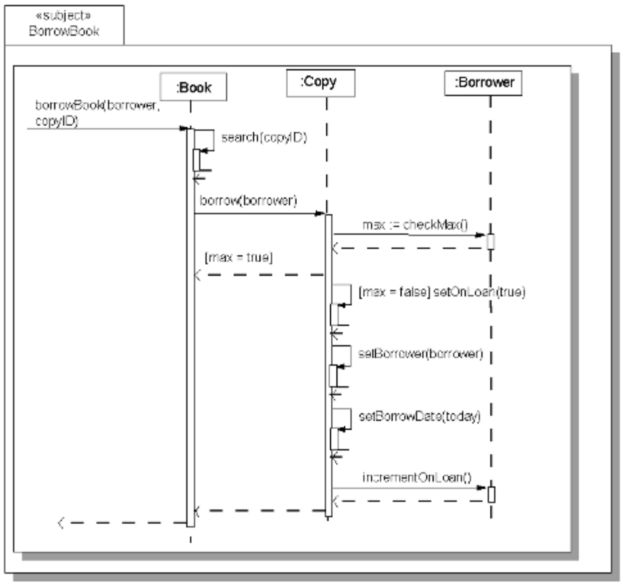


**Figure 123: Borrow Book Interactions**

*Return Library Book*

Figure 124 and Figure 125 illustrate a structural and an interaction design, respectively, for returning library books. This includes a calculation of the appropriate fine for late return.

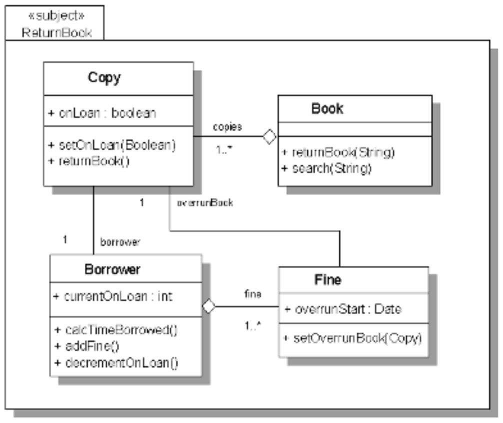


**Figure 124: Return Book Class Diagram**

From a subject-oriented design model perspective, this design highlights another interesting issue. The `ReturnBook` subject has two operations that also appear in the `BorrowBook` subject. These are `search()` and `setOn-Loan(boolean)`, and are calls to the *same* operations in both cases. From an integration perspective, the subject-oriented design model described in this thesis has discussed merge and override. Merging operations means that all merged operations are executed when any one is. Overriding operations means that one operation's specification is overridden by another. Conceptually, neither merge nor override applies. For example, it is not appropriate to call the `setOnLoan(boolean)` operation twice if these operations are merged. On the other hand, conceptually, overriding does not apply, as neither specification is an updated version of the other. “Composing Resource Management Subjects” on page 237 discusses a work-around using override integration, where one of the operations is arbitrarily chosen as the overridden operation and the other as the overriding one. However, the subject-oriented design model should include a mechanism for stating the operations are not just corresponding, but are the same, and therefore only one should appear in the result.

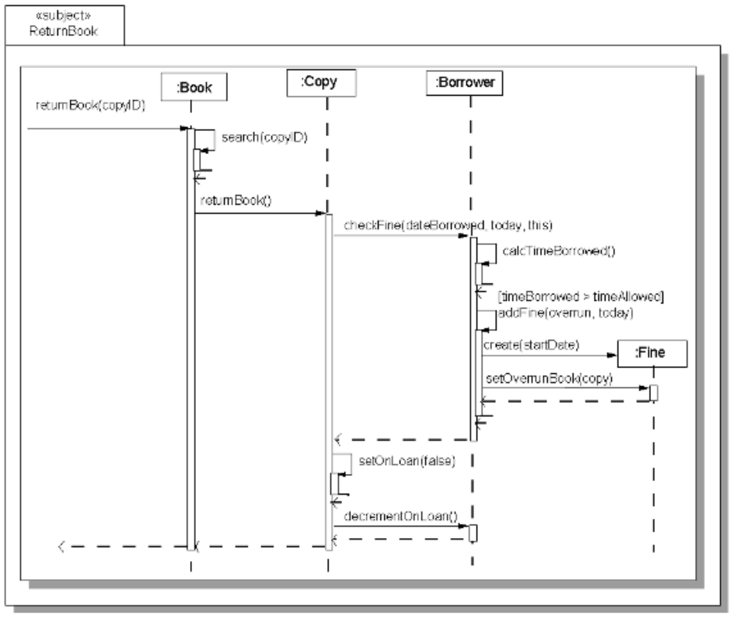


**Figure 125: Return Library Book Interactions**

*Pay Late-return Fine*

Figure 126 illustrates a design for recording the payment of fines. This design does not highlight any further additional interesting points for the subject-oriented design model.

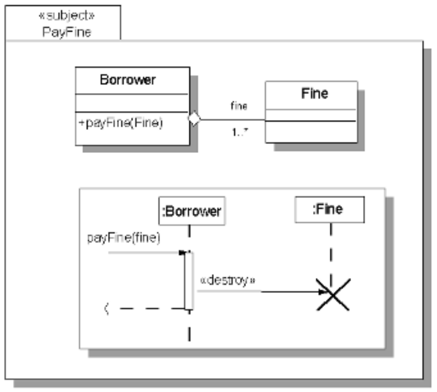


**Figure 126: Pay Fine Class Diagram and Interactions**

*Concurrency* The requirement for concurrency outlines specific library services which should run concurrently, and how they should be synchronized. The designer assigned to this subject could explicitly provide a concurrent design for just those services that are specified. However, this designer recognises that concurrency is not a requirement that is specific to any service in a library, but that it potentially applies outside the library management system, and indeed, to other services within the library management system. This requirement is therefore better designed as a composition pattern, where it can be re-used both inside and outside the current library system under design.

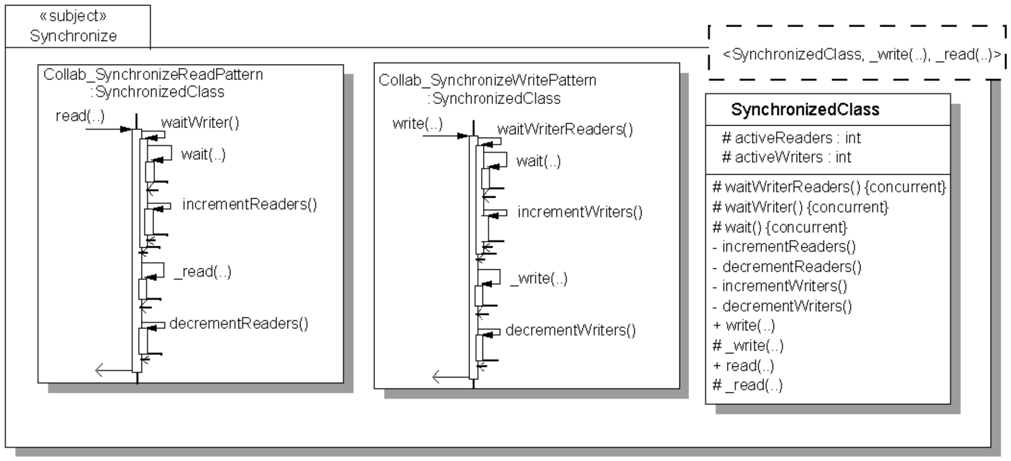


**Figure 127: Synchronize Pattern Classes and Interactions**

The `Synchronize` composition pattern illustrated in Figure 127 has one pattern class, `SynchronizedClass`, representing any class requiring synchronization behaviour. Within this pattern class, two template parameters are defined, called `_read(..)` and `_write(..)`, to represent reading and writing operations. Synchronization behaviour introduces a number of elements, both structural and behavioural, to synchronized 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. In this example, and as described in “Chapter 8: Composition Patterns” on page 198, the designer utilizes operation merge semantics by representing the actual replacing read and write operations with an “_” pre-pended to the template parameter name – that is, using `_read(..)` and `_write(..)`. In this way, when the actual operation is executed in the context of synchronization, the required behaviour is clearly defined within the interactions.

## Composition

This section discusses the use of composition relationships to specify how to compose the different library management system design subjects, and examines the output of such composition. With multiple, independent design subjects, there are multiple possibilities for choosing which ones to compose at any particular time. Research into a supporting design process for the subject-oriented design model should define guidelines to aid this choice. For the purposes of illustration, and to aid discussion, the composition task is divided up into: the composition of subjects specific to resource management; the composition of subjects specific to borrowing and returning books; the composition of the `Synchronize` pattern where appropriate.

### *Composing Resource Management Subjects*

The design subjects appropriate to managing resources are `AddResource`, `RemoveResource`, `SearchResource` and `OrderResource`. Merge integration is appropriate for composing these subjects, as all of the structure and behaviour for each subject is required in the composed subject. In addition, a look at the separate designs shows that each designer generally used names from the requirements specification, and so, generally, the same names were used for the same base concepts. Therefore, a `match[name]` attachment is appropriate for establishing correspondence between elements.

The issues and interesting points discussed within the design sections for each subject were:

- The `RemoveResource` subject defines the `Resource` class as non-abstract while the other subjects define it as abstract.
- The `AddResource` subject specifies that a `Resource` instance must be `storedIn` one `Location`. The `OrderResource` subject uses the `Resource` class for ordering information, and does not consider the implications of its relationship with `Location`, as it does not concern ordering.
- `AddResource` and `OrderResource` both have operations called `set-Course()`, that are different.

Each of these issues may or may not have been noticed by the composition designer. For the purposes of this study of the subject-oriented design model we assume that the likelihood of differences in the specifications of elements has been considered. To cater for it, the composition designer assigns a `prec` attachment to the composition relationship, as illustrated in Figure 128. This specifies that the specifications of elements in `AddResource` take precedence in the event of a conflict. The remaining two issues are discussed in the examination of the output of this composition specification.

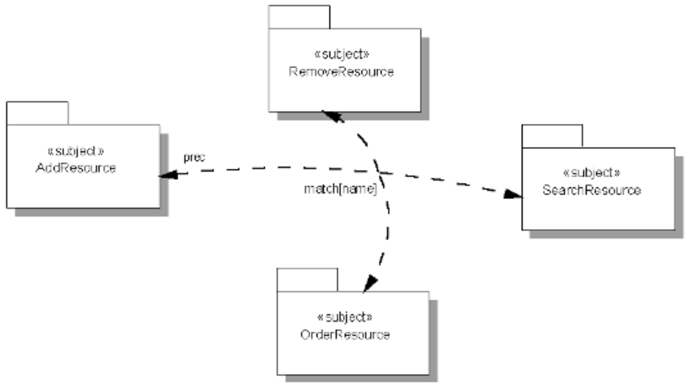


**Figure 128: Specify Composition of Resource Management Subjects**

The composition relationship defined in Figure 128 states that the `AddResource`, `RemoveResource`, `SearchResource` and `OrderResource` subjects are to be merged. Elements with the same name are corresponding, and element specifications in `AddResource` take precedence in the event of a conflict. This specification yields the output illustrated in Figure 129.

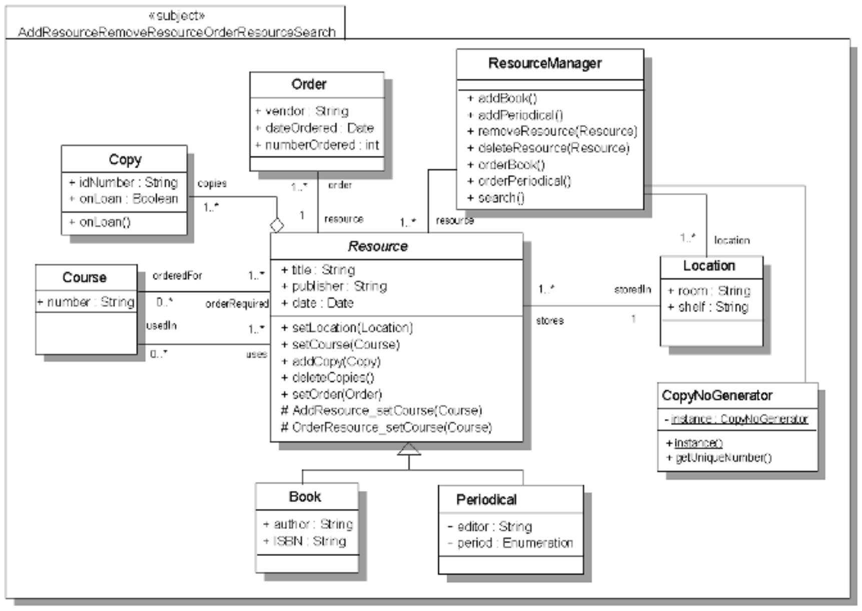


**Figure 129: Output of Composition of Resource Management Subjects**

An examination of this output shows that the operations `setCourse()` have been merged. That means, they have been deemed to correspond (based on the `match[name]` attachment to the composition relationship) and therefore an execution of either one results in the execution of both. The composition process specifies this behaviour with the interaction diagram illustrated in Figure 130.

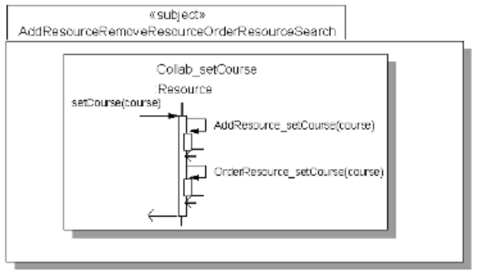

«subject»
AddResourceRemoveResourceOrderResourceSearch
Collab_setCourse
Resource
setCourse(course)
AddResource_setCourse(course)
OrderResource_setCourse(course)

**Figure 130: Generated Interaction**

As discussed previously, however, these operations are different, and should not be considered to correspond. The subject-oriented design provides a means to specify exceptions to a general name-matching correspondence specification. It can be achieved by adding a relationship, with a `dontMatch` attachment, between the two operations. This is illustrated in Figure 131.

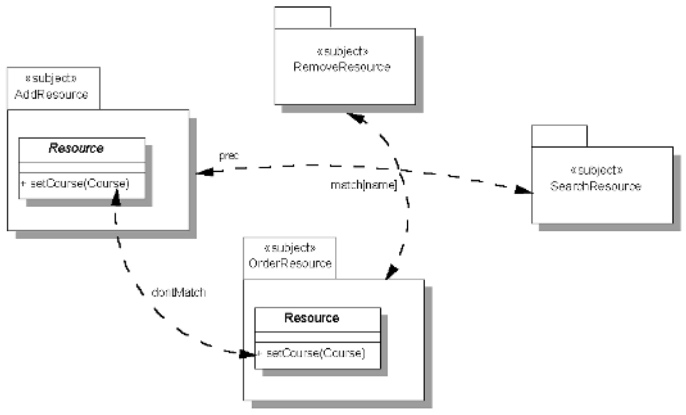


**Figure 131: Specifying Exception to General Matching**

The final previously identified issue relates to the cardinality constraint between `Resource` and `Location`, specified in the `AddResource` subject, that states that a resource must be `storedIn` one location. This constraint causes a problem when `AddResource` is composed with `OrderResource`. It is not appropriate that such a cardinality constraint is put on resources that are only on order. This problem currently cannot be solved using composition relationships, and requires domain knowledge to identify. In such a case, the designer must solve the problem as appropriate in the output subject. Alternatively, the designer may design a separate subject defining the appropriate association between `Resource` and `Location`, and override the association in the composed subject in Figure 129.

A final observation may be made on the output of this composition. The semantics for merging subjects states that the name of the output subject is the concatenation of each of the input subjects. In this case, the output subject's name of `AddResourceRemoveResourceOrderResource-Search` is not ideal. While it is possible to see at a glance which subjects were included in the composition process, it is nonetheless a very long name to work with. This may not be an issue for some domains, but research is required to assess whether it is appropriate to provide a facility to the composition designer to specify the name of the output subject.

*Composing Borrowing Subjects*

The design subjects appropriate to borrowing and returning books are `BorrowBook`, `ReturnBook` and `PayFine`. Merge integration is appropriate for composing these subjects, as all of the structure and behaviour for each subject is required in the composed subject. In addition, a look at the separate designs shows that each designer generally used names from the requirements specification, and so, generally, the same names were used for the same base concepts. Therefore, a `match[name]` attachment is appropriate for establishing correspondence between elements.

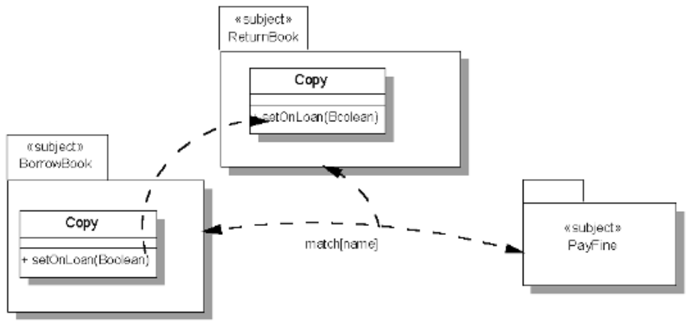


**Figure 132: Specify Composition of Borrowing Subjects**

One interesting issue for the subject-oriented design was previously raised in the description of the design of these subjects. The `ReturnBook` subject and the `BorrowBook` subject both reference `setOnLoan(boolean)` operations, which are the *same*. As previously discussed in "Return Library Book" on page 234, neither merge integration nor override integration is conceptually appropriate as the integration strategy. However, the end result the composition designer wants is one `setOnLoan(boolean)` operation in the output. This end result can be achieved using a composition relationship with override integration as illustrated in Figure 132. The composition designer arbitrarily nominates one of the operations as the one to be overridden, and the other as the overriding one. As required, and as illustrated in Figure 133, only one `setOnLoan(boolean)` operation appears in the result.

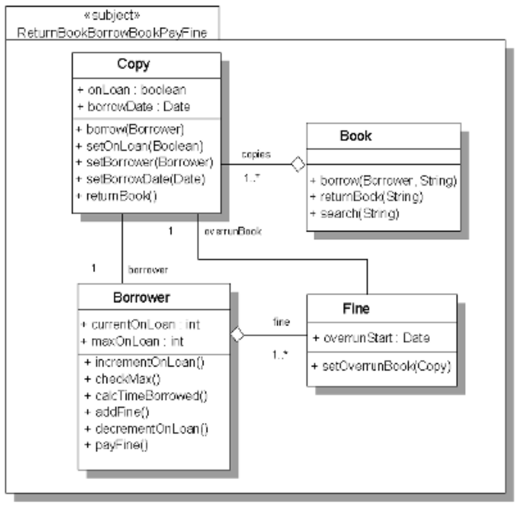


**Figure 133: Output of Composition of Borrowing Subjects**

While the required end result has been achieved, this nonetheless highlights a gap within the subject-oriented design model which reinforces the need for additional integration strategies.

*Composition with Synchronization Pattern*

The output of a composition process is itself a design subject, and so the output of the composition of the resource management subjects may be composed with the `Synchronize` composition pattern subject. For convenience, the output from the resource management composition is named `ResourceMgmt` in this section.

Specifying how to compose the `ResourceMgmt` design subject with the `Synchronize` composition pattern is achieved with the definition of a composition relationship between the two. The `bind` attachment denotes which class(es) are to be supplemented with synchronization behaviour, and which read and write operations are to be synchronized. As illustrated in Figure 134, `ResourceMgmt`'s `ResourceManager` class replaces the pattern class in the output, `addBook()`, `addPeriodical()` and `removeResource()` operations are defined as write operations, and `search()` is defined as read.

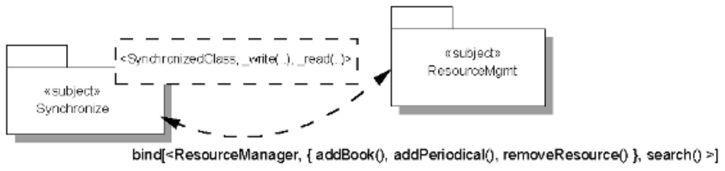

<SynchronizedClass, _write(..), _read(..)>
«subject»
Synchronize
«subject»
ResourceMgmt
bind[<ResourceManager, { addBook(), addPeriodical(), removeResource() }, search() >]

**Figure 134: Specify Composition with Synchronization**

In the output subject illustrated in Figure 135, only the class impact by synchronization is illustrated. All the other classes and relationships are added unchanged.

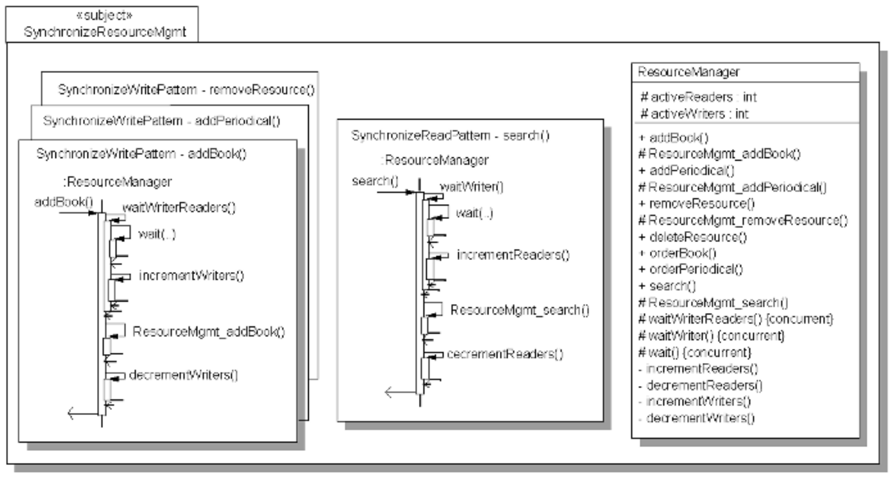


**Figure 135: Output of Composition with Synchronization**

This section has demonstrated how the initial design of a system may be decomposed based on the requirements specifications, and how each design model may be designed separately, and composed later. “10.4. Evaluation” on page 247 assesses this design based on the criteria for assessing design techniques used to motivate this work, described in “Chapter 2: Motivation” on page 11. We now look at how to use the subject-oriented design model for designing changes to a system.

## 10.3. Evolving the LMS

One of the benefits of using the subject-oriented design model stated in this thesis is that its use eases the extensibility of software designs. In this section, we examine the impact of extension requirements on the library management system, and assess the assertion that the subject-oriented design model eases their inclusion into the software design. One requirement is received as a result of the change to the business process associated with borrowing rules. The second requirement arose as a result of a problem with the existing design identified in system test.

Business process change:

- The rule relating to the borrowing of books is changed. In the current design, there is a maximum number of books each borrower may borrow. A change to this rule states that, in addition to the maximum limit, a borrow may not borrow a book if there is a fine outstanding from the previous loan of a book to that borrower.

System test problem:

- During system test, it is found that there is no defined behavioural relationship between the adding of resources to the system, and the maintenance of order information. Once a resource is received and added to the system, a check against the order information of that resource should be made, with receipt of the order recorded.

**Subjects**

As supported by the subject-oriented design model, the project manager may decide to assign the two new requirements to different design teams, working on different design subjects. The `CheckBorrow` subject handles the new rules for borrowing books. The `OrderReceived` subject handles updating order information.

*Changed Rules for Borrowing*

The `CheckBorrow` subject defines a new operation called `check()` to handle checking that the borrower has not reached its maximum limit, and that there are no fines outstanding. This is illustrated in Figure 136. There are no interesting issues relating to the subject-oriented design model.

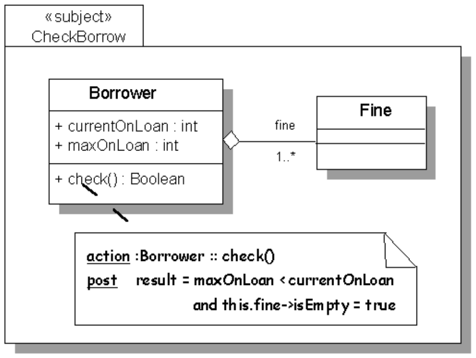


**Figure 136: Updating rules for borrowing**

*Update Order Information*

Figure 137 illustrates the design for updating order information based on information based on the receipt of resources.

This design does not explicitly refer to any of the add resource properties, but knowledge of the subsequent composition of this subject with the design for adding resources does have some influence. In particular, merge integration semantics for integrating operations applies, and therefore, the scope of the lifeline of adding resources is relevant for the specification of the parameters to the `bookReceived()` operation. As described in "Impact of Merge on Operations" on page 188, merging operations with parameters is only possible with compatible parameter lists. In other words, in order for `bookReceived()` to execute, the information it requires through its parameters must be available from the operation first called in the execution combination, which, in this case, is `addBook()`. More details are illustrated in their composition specification in Figure 140.

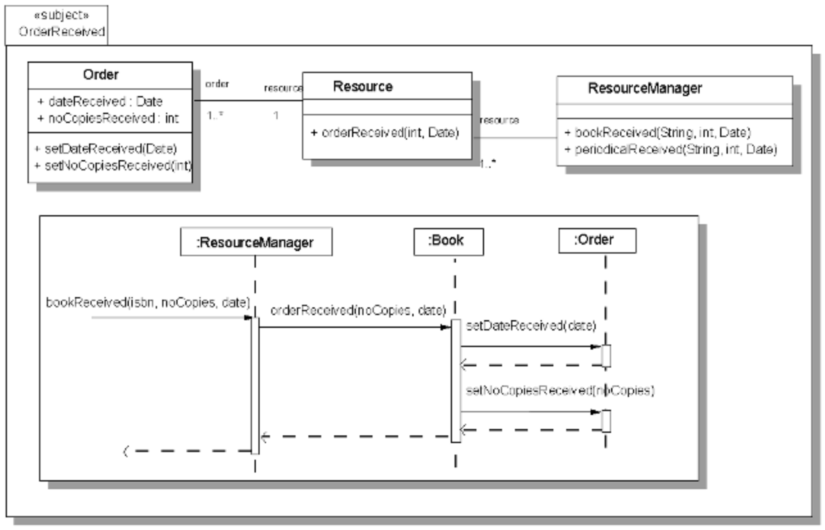

«subject»
OrderReceived
Order
+ dateReceived : Date
+ noCopiesReceived : int
+ setDateReceived(Date)
+ setNoCopiesReceived(int)
order
resource
1..*
1
Resource
+ orderReceived(int, Date)
resource
1..*
ResourceManager
+ bookReceived(String, int, Date)
+ periodicalReceived(String, int, Date)
:ResourceManager
:Book
:Order
bookReceived(isbn, noCopies, date)
orderReceived(noCopies, date)
setDateReceived(date)
setNoCopiesReceived(noCopies)

**Figure 137: Order Received**

**Composition** The design subjects to be composed to include the new rules for borrowing books are the `CheckBorrow` subject and the `ReturnBookBorrow-BookPayFine` subject. Override integration is appropriate for composing these subjects, as the `ReturnBookBorrowBookPayFine` subject contains design which is now obsolete because of the new requirement, and the `CheckBorrow` subject contains a design for the new requirement. A `match[name]` attachment specifies how to identify corresponding elements. One exception to this is that the new `check()` operation is designed to override the old `checkMax()` operation, and this must be explicitly specified with a composition relationship. This composition specification is illustrated in Figure 138.

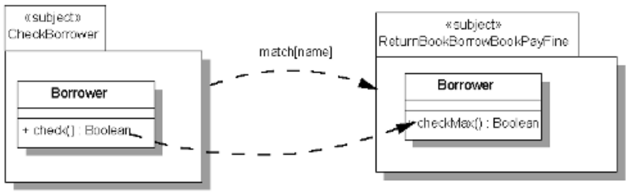


**Figure 138: Specify Composition of Borrow Checking Update**

The ability to simply override one operation with another operation with a different name depends on the forwarding semantics discussed throughout this thesis. The `checkMax()` operation forwards to `check()` in the output, and therefore any references to now `checkMax()` reference `check()`.

The output from this composition, illustrated in Figure 139, is the same as output from Figure 133 except for its name, and that the `checkMax()` operation has been overridden. The interaction calling `checkMax()` is changed to call `check()` (not illustrated), as defined by forwarding semantics.

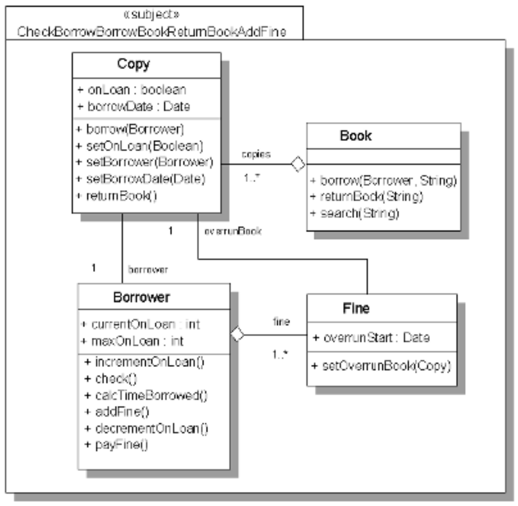


**Figure 139: Output of Composition of Borrow Checking**

The design subjects to be composed to include the updating of order information on addition of resource information are the `OrderReceived` subject and the composed resource management subjects. As before, and for convenience, the output from the resource management composition is named `ResourceMgmt` in this section.

Specification of how to compose the `OrderReceived` subject with the `ResourceMgmt` subject is achieved with a composition relationship with merge integration (see Figure 140). Merge is chosen as the integration strategy as this is additional behaviour, designed to enhance already existing behaviour. A `match[name]` attachment specifies how to identify corresponding elements.

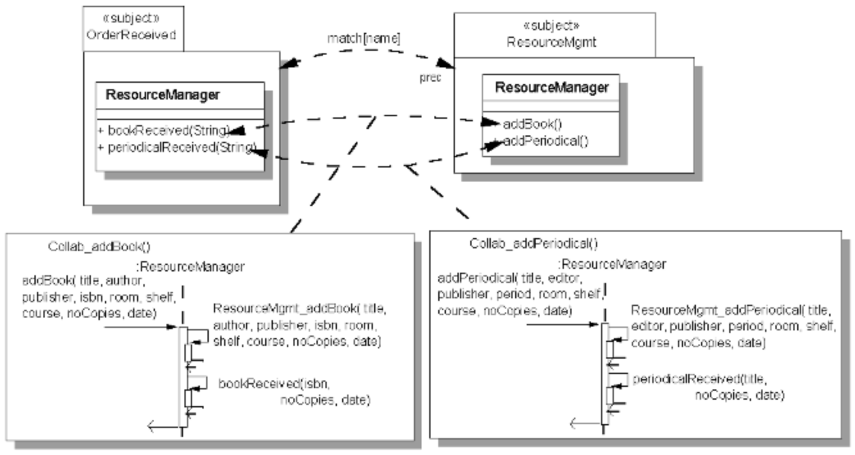

«subject»
OrderReceived
ResourceManager
+ bookReceived(String)
+ periodicalReceived(String)
match[name]
prec
«subject»
ResourceMgmt
ResourceManager
addBook()
addPeriodical()
Collab_addBook()
:ResourceManager
addBook( title, author, publisher, isbn, room, shelf, course, noCopies, date)
ResourceMgmt_addBook( title, author, publisher, isbn, room, shelf, course, noCopies, date)
bookReceived(isbn, noCopies, date)
Collab_addPeriodical()
:ResourceManager
addPeriodical( title, editor, publisher, period, room, shelf, course, noCopies, date)
ResourceMgmt_addPeriodical( title, editor, publisher, period, room, shelf, course, noCopies, date)
periodicalReceived(title, noCopies, date)

**Figure 140: Specify Composition with Receiving Orders**

There are two exceptions to this general matching case. The `bookReceived()` operation and the `addBook()` operation are considered corresponding as they as to be executed together. Similarly for the `periodicalReceived()` operation and the `addPeriodical()` operation. The composition designer dictates the order of execution of these two corresponding sets by attaching interactions to the appropriate composition relationships. This order conforms to the rules associated with merging operations of different signatures described in “Impact of Merge on Operations” on page 188. The calling operation must have the information to support the calls to subsequent operations in the corresponding set.

The output of the composition specification in Figure 140 is illustrated in Figure 141.

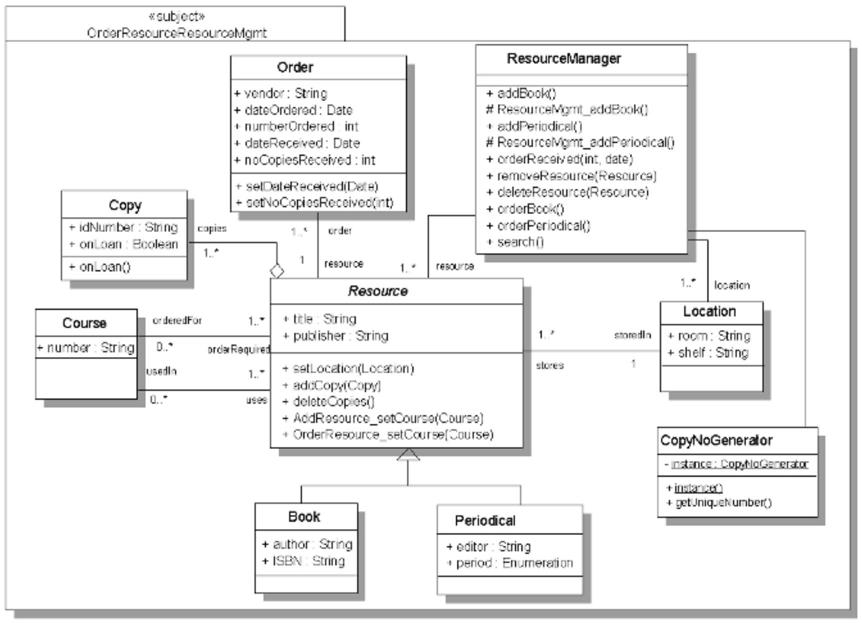


**Figure 141: Output of Composition with Receiving Orders**

This section has demonstrated how the subject-oriented design model supports the evolution of existing software designs. Changes may be designed independently in separate design models, and subsequently composed with the existing designs. We now evaluate the model against the criteria motivating this work described in "Chapter 2: Motivation" on page 11.

## 10.4. Evaluation

The criteria motivating this research described in "Chapter 2: Motivation" on page 11 were: product flexibility; comprehensibility; and managerial concerns. We now look at the experience of designing and evolving the library management system case study against these criteria.

**Product Flexibility**

As discussed in "Chapter 2: Motivation" , product flexibility is the *"possibility of making drastic changes to one part of the system, without the need to change others"*. Here, it was illustrated that scattering and tangling of design elements within traditional object-oriented models was an impediment for ease of change. In this case study, the subject-oriented design model's support for decomposition based on structural matching with requirements showed itself to considerably reduce the negative effects scattering and eliminate tangling entirely. From a scattering perspective, support for a requirement still needs a design across multiple classes and design elements. This is the nature of object-oriented design. However, the negative impact of scattering, where it is difficult to find all the appropriate design elements for a particular requirement, is reduced. This is because all the design elements in a particular subject are pertinent for the requirement under design, and all the design elements required to support that requirement are contained in the particular subject. This is the case for each of the design subjects in the library management system. From a tangling perspective, this property is eliminated, as for each of the library design subjects, only one requirement's design is contained in that subject. This is the case even where one of the requirements, the concurrency one, impacts other requirements. The use of composition patterns, such as the synchronization composition pattern in the library management system, supports the clean separation of such cross-cutting behaviour.

As regards traceability and evolvability, the ability to decompose design models to structurally match requirements makes this easier. For each of the library management system design subjects, it is clear which requirement is supported. For each requirement, it is clear which design subject supports it. The changes to the library design proved no more difficult to design separately than did the original requirements. However, the case study did illustrate that the composition designer needed to be careful when merging corresponding operations. Merging the recording of order receipt information with the adding of resources to the system (Figure 140 on page 246) required careful specification of the order of execution of corresponding operations.

**Comprehensibility**

As discussed in "Chapter 2: Motivation" , comprehensibility is the *"possibility of studying one part of the system at a time. The whole system can therefore be better designed because it is better understood"*. The subject-oriented design model does not guarantee that a design will be easy to understand. Where a requirement is complex, it is likely its design will be complex, and any designer not familiar with the details of such a requirement may find its design details difficult to understand. What *has* been achieved with the subject-oriented design model, as illustrated in the library management system, is that the design can be studied "one part at a time". The reduction of the negative impact of scattering, and the removal of tangling, both support the study of the system one requirement at a time.

**Managerial**

As discussed in "Chapter 2: Motivation" , 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"*. This case study has not proved that the length of development time using the subject-oriented design model is less than the length of development time using traditional object-oriented approaches. To do this requires timing different teams, of similar design experience, and with similar levels of familiarity with the library management domain, creating two separate designs. What the case study *has* illustrated though, is that "different groups can work on different parts of the system with little need for communication". Without subject-oriented design, the project manager is faced with the situation illustrated in Figure 114 on page 228, where designer access to classes must be managed, requiring communication amongst designers. As illustrated in the case study, each of the design teams may work independently of the others, without communication.

**Comment**

Not surprisingly, the subject-oriented design model performs well against the stated criteria, since it was designed to do exactly that. However, this case study identified some problems outside these criteria. First, conflicting constraints are not readily recognisable, and cannot be handled with composition relationships. As illustrated in Figure 129 on page 238, the cardinality constraint imposing one location for each resource conflicts with resources only on order, which do not yet have a location. As discussed, the designer must notice this in order to fix it. It is likely that using traditional object-oriented methods this problem would not occur. Whether it was the designer adding orders to resources after the location was associated, or the designer associating locations to resources after the orders were associated, in either case, the problem is likely to have been resolved. Further research is required to assess whether this problem can be ameliorated with subject-oriented design.

Another problem identified is the limitations in the integration strategies currently supported. As illustrated in Figure 132 on page 240, there are times when neither override nor merge is appropriate. In this particular case, a workaround is achieved within the current subject-oriented design model, but it is likely that other cases might not be so readily worked around. This possibility has been identified and catered for in the metamodel for subject-oriented design described in "Chapter 5: Composition Relationship: An extension to the UML Metamodel" on page 109, where the `Integration` metaclass is abstract, supporting its extension by additional integration strategies.

In addition, we must recognise that the approach has not been applied to a large project, and therefore, unforeseen difficulties are possible. For example, what might be the sociological impact of separating teams? Will teams welcome the narrowing of design focus to a single requirement, as it may be less challenging? Is it reasonable to assume that composition relationship designers will have sufficient skill to assess the impact of composing design subjects? All the implications of using subject-oriented design will only become clear with its application to a large project.

Notwithstanding these issues and uncertainties, the benefits against the specified criteria are sufficiently encouraging for continuing research into this area, and extending the model as described in “11.2. Future Work” on page 253.

## 10.5. Chapter Summary

This chapter illustrates and evaluates the design of a library management system using the subject-oriented design model. Decomposition into design subjects is based on a one-to-one mapping with the requirements specifications. This approach to identifying design subjects is taken both for the initial system, and for the new requirements received after the design of the initial system is in place. Composition specifications using composition relationships are demonstrated, with the output of each illustrated.

The design of the case study is evaluated against the criteria motivating this research: product flexibility; comprehensibility; and managerial concerns. Subject-oriented design structurally matches design models with the structure of requirements specifications. As a result, it is illustrated that each criteria benefits from the considerable reduction in the negative impact of scattering properties, and from the removal of tangling properties. However, some issues are raised with the model. Composing separate design models may lead to the existence of conflicting constraints in the composed design model. This problem is currently not solvable within the subject-oriented design model, and so the designer must be vigilant in investigating and solving such problems. In addition, the currently available integration strategies are not sufficient to cater for all possibilities. This possibility was addressed in the specification of the metamodel for subject-oriented design discussed in “Chapter 5: Composition Relationship: An extension to the UML Metamodel” on page 109.

# Chapter 11: Summary, Conclusions and Future Work

This thesis has addressed a number of issues relating to the current limitations with object-oriented design techniques. While there are benefits to the approach described as it is in this thesis, much work remains to be done. This chapter summarises the research to date as defined in this thesis, draws conclusions as to its benefits and limitations, and details the current view of remaining work in this area.

## 11.1. Summary

This thesis described a new approach to object-oriented design, which addresses issues relating to the modularisation and composition capabilities of existing approaches.

First, the thesis illustrates and highlights the problems caused by limitations in the existing modularisation capabilities of the current object-oriented design paradigm. At the root of the problems is the fundamental structural difference between the way requirements are specified and the way object-oriented designs are specified. Because of this structural difference, design for a single requirement is *scattered* across the design elements of an object-oriented model, and a single design element is *tangled* with support for multiple requirements. This leads to difficulties in model comprehension, and difficulties relating to the ease of extensibility and re-use of object-oriented design models.

The thesis then proposed a new approach to designing object-oriented systems that removes the structural mismatch with requirements by extending the decomposition capabilities of object-oriented models. This extension supports the direct decomposition of object-oriented models to match the structure of a requirements specification. In other words, design models may be defined separately for each requirement in a requirements specification. The thesis illustrates how potential overlaps in the design of core concepts for different requirements are catered for. Cross-cutting requirements are also supported within the model.

Decomposition in this manner requires supporting composition capabilities. Therefore, the thesis defined a new kind of design relationship, called a *composition relationship* that supports the specification of how design models may be composed. With composition relationships, areas of overlap in different design models to be composed may be identified, along with specifying how models should be integrated. The syntax and semantics of composition relationships relative to the UML Metamodel are defined in detail. This is achieved with meta-class models illustrating the constructs associated with composition relationships, well-formedness rules denoting constraints on the specification of composition relationships, and a detailed description of the semantics of composition as defined by composition relationships.

The composition relationship metamodel is designed to support seamless addition of integration strategies. The thesis illustrates how this may be achieved by defining two integration strategies within the context of the composition relationship metamodel. These strategies are *override* integration and *merge* integration.

The impact of override integration on the UML design elements supported in this thesis is described in detail. Override integration essentially replaces elements in one design model with corresponding elements in another design model. Merge integration is also defined in detail, and entails the composition of design models where all of the design elements are relevant for inclusion in the composed model.

For merge integration, the thesis also demonstrates how sophisticated merging of behaviour is possible by enabling the attachment of interaction diagrams to a composition relationship. In this manner, the behaviour of corresponding operations may be explicitly defined as part of the composition specification. The thesis further expounds on this theme by supporting the specification of *patterns* of composition, based on and extending the notions of templates and binding that is already supported within the UML, combined with the power of composition as defined within this thesis. The thesis illustrates how composition patterns support the specification of how *cross-cutting* behaviours, which impact design elements in a uniform manner, may be composed wherever required. Merge integration also requires strategies for reconciling possible conflicts between design elements. This thesis defines a number of different possible strategies - subject precedence, default specification, explicit value specification, and transform functions - and includes these strategies in the context of the UML metamodel.

The thesis then illustrated how the subject-oriented model changes the design for the motivating example, and asserts that the design is easier to understand, illustrates the ease with which it may be extended, and asserts that the design subjects are easier to reuse in different compositions.

## 11.2. Future Work

The work described in this thesis represents the initial “proof-of-concept” of the subject-oriented design model. For a subset of the constructs in one design language (the UML), the subject-oriented design model proves itself to be valuable against some standard software engineering quality criteria - comprehension, extensibility and reuse. However, much work remains to be done to make the subject-oriented design model a formally sound and commercially viable option for large projects. This section categorises the areas where work is required as follows:

- *Supporting Technologies:* This section examines what is required for tool support, and alignment with other technologies at the programming level
- *Additional Features and Rules:* This section considers additional features which would extend the capabilities of the subject-oriented design model.
- *Software Development Process Support:* This section discusses how some work into examining the impact of the availability of capabilities such as those defined within the subject-oriented design model might change a software development process.
- *Formal Foundations:* The description of the semantics of the subject-oriented design model is non-formal. This section discusses the possible need for a more mathematical foundation for the model.

### Supporting Technologies

There are two main areas in which supporting technologies are required to make use of the subject-oriented design model viable for large projects: supporting CASE tool environments at the design level; and automation of a link from this design approach to supporting programming models.

First, CASE tool support for the design phase. Ideally, in order to make the subject-oriented design model a commercially viable option, support would need to be included in the major commercial CASE tools - for example, Rational Rose or Together. It currently seems unlikely that this will occur unless the extensions to the UML described in this language become part of the standard language. Therefore, future work in this area will be focused on including support for the model in an open source CASE tool, called Argo/UML. Argo/UML was originally developed by a small group of people as a research project, and this group now provides the source code for Argo/UML publicly on the internet for review and customisation. The UML metamodel is directly supported, and therefore, we intend to include extensions to the tool to support subject-oriented design in a public manner that conforms to the vision and publication standards of any other extension to the tool.

Secondly, links to supporting technologies at the programming level should be considered. The most closely related programming model to subject-oriented design is the subject-oriented programming model currently implemented in a tool called Hyper/J [Tarr & Ossher 2000]. There are two parts to linking the design model described in this thesis with the subject-oriented programming model: programming the individual design subjects into separate Java code subjects; and generating *composition rules* (the means for specifying how programs should be composed) from composition relationships. Programming code subjects from design subjects is the same process as standard programming from a design. Generating composition rules from composition relationships requires some investigation to determine the differences between composition relationships and composition rules, and to assess the exact mapping from composition relationship constructs to composition rules. An actual generation implementation is also required. Generation of composition rules from composition relationships should be implemented within the context of the Argo/UML tool.

Another programming approach that is related to the subject-oriented design model is the work on aspect-oriented programming currently implemented in a tool called AspectJ [Kiczales & Lopes 1999]. Aspect-oriented programming supports the separate implementation of cross-cutting requirements from base programs implementing the core problem domain. In AspectJ, aspect programs contain the implementation of methods for the cross-cutting requirement, and an indication of the places within the base programs where these methods should be included. Composition is achieved with an aspect weaver that adds the cross-cutting methods to the base program as appropriate. An interesting piece of future work is the extent to which composition patterns, as defined in the subject-oriented design model, may be used as a means to design cross-cutting aspects. It is conceivable that the combination of a design subject containing placeholders for corresponding design elements, and composition relationships binding other subjects to a cross-cutting subject specification (i.e. the combination that defines a composition pattern) may be used to design aspect programs.

### Additional Features and Rules

The most important extension required to the subject-oriented design model as described in this thesis is to analyse and include support for all UML design models. The scope of the work for this thesis was essentially class and interaction models. Support for object, state, activity, use case, component and deployment models is required.

Another interesting area that could extend the subject-oriented design model is consideration of different kinds of relationships between design subjects. These relationships could constrain the kinds of composition relationships possible. For example, if two subjects support two mutually exclusive requirements, then this relationship could be specified between the subjects, thereby constraining their composition - that is, only one of the two subjects may be involved in a particular composition context. As described in “Feature Interaction Problem” on page 106, investigation into this area might yield interesting results in how to support the design of features whose interactions are constrained. Relationships between subjects may also necessitate that compositions are *ordered* in a particular way - that is, it is appropriate for one set of subjects to be composed prior to composition with another (set of) subject(s). This area needs to be investigated further, and if required, support for ordering of compositions included in the model.

From an integration perspective, some additional features could be included to extend the capabilities of the model. For example:

- Override integration, as currently specified, replaces (some) design elements in one subject with corresponding design elements in another. In some cases, there may also be design elements in the overridden subject that are no longer required, but are not explicitly replaced by corresponding elements in the overriding subject. An additional feature to support this requirement is to provide a means to identify elements in the overridden subject that are to be deleted as a result of composition - that is, *not* explicitly integrated with corresponding elements, but nonetheless not appearing in the output of the composition.
- Two kinds of integration strategies are defined in this thesis - override and merge. There may be other kinds of integration strategies that are useful for composing models. For example, a *select* integration strategy, where a dynamic selection of the appropriate design elements from different subjects is made based on the values of environment variables, is an

interesting additional feature which should be considered. A complete investigation into requirements for different integration strategies is an interesting area for future work.

- An area not considered in this thesis is the possibility of *additional* properties arising for the output of the composition. These are not defined in any input subject, but arise *as a result* of the composition itself. This area has not been investigated, but is included in future work.
- In both override integration and merge integration, it is possible that cycles may be created in the output subject. Currently, this is treated as a breakage of the well-formedness rules, and must be fixed by the designer. A more helpful approach may be possible, using ideas from [Walker 2000].

The composition patterns model, discussed in “Chapter 8: Composition Patterns” on page 198, also presented interesting opportunities for development. These are:

- In the current model, a composition designer specifies pattern classes and template parameters that are fully replaced on composition with those elements defined for replacement on the composition relationship. As discussed in “Further Potential for Template Rule Specification” on page 205, there is considerable scope for extending the capabilities of the composition patterns designer in the area of specifying constraints on the replacing elements.
- Related to the previous item, there is also scope to broaden the capabilities of the composition relationship defining the elements that replace templates with its `bind[]` attachment. For example, sophisticated wildcard matching is possible.
- In the current model, there is a restriction that only one of the subjects in a single composition context is a composition pattern. Further investigation into whether there is a need to remove this restriction is required. If it should be removed, an examination of the impact of its removal on the model must be done.

From a more detailed perspective, there are other areas within the subject-oriented design model’s features and rules that may be extended to expand the usefulness of the model. Those areas are:

- In “Forwarding of References” on page 96, there is a discussion on how references to elements which may have changed as a result of composition may be forwarded to refer to the changed elements in the output subject.

Within the current model, a single specification of forwarding covers all elements within a particular subject. An area worth investigating is the need for, and usefulness of, supporting separate forwarding options for individual elements.

- Also related to forwarding, there is a discussion, in “Merged Operations and Forwarding of References” on page 195 , of how the process for creation of operations to define the delegation to corresponding, merged operations might be refined to only require one such operation, to which all the input corresponding operations forward. This area requires investigation to determine any possible impact on the semantics of forwarding in general.
- In “Incompatible Elements” on page 100, there is a discussion on how the current subject-oriented design model restricts composition of operations with *any* conflicting properties (excluding parameter lists). Further work is required to define a full set of appropriate rules guarding, on the one hand, against loss of any input subject constraints in the composed model, while not being overly restrictive.
- In “Merged Operations with Return Types” on page 194, there is a discussion relating to the difficulties associated with return values from merged operations. Support, similar to that provided in Hyper/J, for allowing a designer to work with the return values of all executed operations to provide the most appropriate result should be included in the subject-oriented design model.
- Within the current subject-oriented design model, a rule has been defined restricting corresponding elements to being of the same type. An interesting area for future work is to analyse whether this rule is too restrictive. Within the database schema integration field, some different kinds of fields may be integrated. An analysis of the impact of removing this rule on integration of subjects is included in future work for the subject-oriented design model.
- More flexible means for general specification of matching for corresponding elements needs to be included in the model. Currently, general matching is supported based on a name-match of elements. Other possibilities need to be examined, and if appropriate, included in the model.

**Software Development Process Support**

The impact of the subject-oriented design model on the software development process has not been explored in this thesis. This is an important area requiring examination. Some of the areas in which a software process may aid the subject-oriented design model are:

- in the initial selection of the appropriate design subjects based on the requirements specification. For example, further guidelines beyond "one requirement, one subject" may be appropriate as to whether there should be a separate subject designed for a particular problem versus whether the design should be included as part of another subject.
- in the decision as to whether to design a change/update to a particular subject as a separate subject in its own right (and use composition with override integration), or whether to simply change the subject directly. Work into assessing the impact of maintaining multiple subjects versus making some small changes directly will assist in the development of a set of guidelines to assist such a decision.
- in the decision as to the extent of the autonomy of separate design teams for separate overlapping subjects. Where there is no communication between teams on overlapping elements, conflicts may require complex composition relationships for the specification of composition. Where there is some communication, composition relationships may be less complex. Guidelines to find the most appropriate balance for a particular project are required.

A complete assessment of the impact of the subject-oriented design model on the full software development process is required.

Another major area that has not been addressed in this thesis is the possibilities associated with the "harvesting" of design subjects from design models not designed using the subject-oriented design model. Object-oriented design has been around for some time, and therefore there may be many models which contain the design for problems/requirements that could potentially be reused elsewhere. An interesting area for future work is to analyse whether it is possible to extract design subjects from legacy design models, that contain the complete design for only one problem/requirement.

**Formal Foundations**

The specification of the subject-oriented design model in this thesis is informal, and therefore it has not been proven that it is mathematically sound. A formal, mathematical foundation for the model might therefore be useful. Work in this area will align itself with any formalisation of the UML itself.

An interesting extension to such a formal foundation is the scope for defining an algebra relating to subject composition. This could include the specification of a composition *operator*, on which properties such as associativity, commutativity, and transitivity might be defined.

## 11.3. Conclusions

The objective of this thesis was to realise more of the benefits of object-oriented software design than are currently evident with existing approaches. For small scale examples, with a subset of the UML language, this is achieved with the addition of a decomposition capability supporting the structuring of object-oriented designs with requirements specifications. Within this scope, design models are easier to understand, extend and reuse. Understanding the design of a single requirement necessitates understanding the design of only one design model, without having to consider elements not relevant for that requirement. Alternatively, understanding a particular design model necessitates understanding only one requirement. Changing a design is simpler, as any change may be made separately, to be integrated later, as specified with a composition relationship. Re-use of design models is more achievable because of the lack of tangling of design elements supporting multiple requirements. With composition patterns, reuse of cross-cutting requirements is supported.

Though no evidence is presented to prove the same results are achievable for large-scale commercial projects or for all kinds of design models, the results illustrated are sufficiently encouraging to warrant further focus. As a priority, all the UML design models must be examined to assess the impact of composition on them. Another priority, without which the subject-oriented design model is arguably not usable, is the inclusion of support for the model in a CASE tool that is sufficient to handle large-scale projects. When these two areas have been handled, then the subject-oriented design model may be tested for its effectiveness in achieving the required benefits of software design. Results illustrated in this thesis lend encouragement and hope that the toolbox of the software engineer is considerably strengthened when the subject-oriented design model is included.

# Bibliography

| | |
|---|---|
| [Aksit et al. 1992] | Mehmet Aksit, Lodewijk Bergmans, Sinan Vural *“An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach”* In proceedings of European Conference on Object-Oriented Programming (ECOOP) 1992 |
| [Alencar et al. 1996] | Paolo Alencar, Donald Cowan, Torsten Nelson, Carlos Lucena. *“Towards a formal link between viewpoints in analysis and implementation”* In proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA) Workshop on Subjectivity, 1996 |
| [Andersen & Reenskaug 1992] | Egil Andersen, Trygve Reenskaug. *“System Design by Composing Structures of Interacting Objects”* In proceedings of European Conference on Object-Oriented Programming (ECOOP) 1992 |
| [Atkinson et al. 1990] | Malcolm Atkinson, François Bancilhon, David DeWitt, Klaus Dittrich, David Maier, Stanley Zdonik. *“The Object-Oriented Database System Manifesto”* Deductive and Obkect-Oriented Databases, Elsevier Science Publishers, 1990 |
| [Batini et al. 1986] | C. Batini, M. Lenzerini, S.B.Navathe. *“A Comparative Analysis of Methodologies for Database Schema Integration”* ACM Computing Surveys, Vol. 18, No. 4, 1986 |
| [Bell & Grimson 1992] | David Bell, Jane Grimson. *“Distributed Database Systems”* Addison-Wesley, 1992 |
| [Bertino & Illarramendi 1996] | Elisa Bertino, Arantza Illarramendi. *“The Integration of Heterogeneous Data Management Systems: Approaches Based on the Object-Oriented Paradigm”* In Object-Oriented Multidatabase Systems - A solution for Advanced Applications, Eds: Bukhres, Elmagarmid. Prentice-Hall 1996 |
| [Booch 1994] | Grady Booch. *“Object-Oriented Analysis and Design with Applications”* 2nd Edition, The Benjamin/Cummings Series in Object-Oriented Software Engineering, 1994 |
| [Booch et al. 1998] | Grady Booch, James Rumbaugh, Ivar Jacobson. *“The Unified Modeling Language”* The Object Technology Series, Addison-Wesley, 1998 |

| | |
|---|---|
| [Bright et al. 1992] | M.W.Bright, A.R.Hurson, Simin H. Pakzad. *“A Taxomony and Current Issues in Multidatabase Systems”* IEEE Computer, March 1992 |
| [Bright et al. 1994] | M.W.Bright, A.R.Hurson, Simin H. Pakzad. *“Automated Resolution of Semantic Heterogeneity in Multidatabases”* ACM Transactions on Database Systems, Vol. 19, No. 2, June 1994 |
| [Carmichael 1994] | Andy Carmichael (Editor) *“Object Development Methods”* SIGS Books, 1994 |
| [Chambers 1993] | Craig Chambers. *“Predicate Classes”* In proceedings of European Conference on Object-Oriented Programming (ECOOP) 1993 |
| [Chiba & Masuda 1993] | Shigeru Chiba, Takashi Masuda. *“Designing an Extensible Distributed Language with a Meta-Level Architecture”* In proceedings of European Conference on Object-Oriented Programming (ECOOP) 1993 |
| [Clarke 2000a] | Siobhán Clarke. *“Extending UML Metamodel for Design Composition”* In proceedings of the International Conference on Software Engineering (ICSE) Workshop on Multi-Dimensional Separation of Concerns in Object-Oriented Systems, 2000 |
| [Clarke 2000b] | Siobhán Clarke. *“Composing Design Models: An extension to the UML”* In proceedings of the 3rd Unified Modeling Language (UML) conference, 2000 |
| [Clarke 2000c] | Siobhán Clarke. *“Designing Reusable Patterns of Cross-Cutting Behaviour with Composition Patterns”* In proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA) Workshop on Advanced Separation of Concerns, 2000 |
| [Clarke et al. 1999a] | Siobhán Clarke, William Harrison, Harold Ossher, Peri Tarr. *“Subject-Oriented Design: Towards Improved Alignment of Requirements, Design and Code”* In proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA) 1999 |
| [Clarke et al. 1999b] | Siobhán Clarke, William Harrison, Harold Ossher, Peri Tarr. *“The Dimension of Separating Requirements Concerns for the Duration of the Development Lifecycle”* In proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA) Workshop on Multi-Dimensional Separation of Concerns in Object-Oriented Systems, 1999 |
| [Clarke et al. 1999c] | Siobhán Clarke, William Harrison, Harold Ossher, Peri Tarr. *“Subject-Oriented Design: Support for Evolution from the Design Stage”* In proceedings of the Workshop on Software and Organisation Co-Evolution (SOCE) 1999 |

| | |
|---|---|
| [Clarke et al. 1999d] | Siobhán Clarke, William Harrison, Harold Ossher, Peri Tarr. *"Separating Concerns Throughout the Development Lifecycle"* In proceedings of European Conference on Object-Oriented Programming (ECOOP) Workshop on Aspect-Oriented Programming, 1999 |
| [Clarke et al. 1999e] | Siobhán Clarke, William Harrison, Harold Ossher, Peri Tarr. *"Designing for Evolution with Subjects"* In proceedings of the International Conference on Software Engineering (ICSE) Workshop on Software Change and Evolution, 1999 |
| [Clarke & Murphy 1998a] | Siobhán Clarke, John Murphy. *"Composition of UML Design Models: A tool to support the resolution of conflicts"* In proceedings of Object-Oriented Information Systems (OOIS) 1998 |
| [Clarke & Murphy 1998b] | Siobhán Clarke, John Murphy. *"Developing a Tool to Support Aspect-Oriented Programming principles at the Design Phase"* In proceedings of the International Conference on Software Engineering (ICSE) Workshop on Aspect-Oriented Programming, 1998 |
| [Clarke & Murphy 1998c] | Siobhán Clarke, John Murphy. *"Verifying Components under development at the design stage: A tool to support the composition of component design models"* In proceedings of the International Conference on Software Engineering (ICSE) Workshop on Component-Based Software Engineering, 1998 |
| [Clarke & Murphy 1997] | Siobhán Clarke, John Murphy. *"Developing a Tool to support Composition of the Components in a Large-Scale Development"* In proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA) Workshop on Object-Oriented Behavioural Semantics, 1997 |
| [Coleman et al. 1994] | Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes, Paul Jeremes. *"Object-Oriented Development. The Fusion Method"* Prentice Hall 1994 |
| [Collet et al. 1991] | Christine Collet, Michael Huhns, Wei-Min Shen. *"Resource Integration Using a Large Knowledge Base in Carnot"* IEEE Computer, December 1991 |
| [Cook & Daniels 1994] | Steve Cook, John Daniels. *"Designing Object Systems. Object-Oriented Modelling with Syntropy"* Prentice-Hall, 1994 |
| [deChampeaux & Faure 1992] | Dennis de Champeaux, Penelope Faure. *"A Comparative Study of Object-Oriented Analysis Methods"* Journal of Object-Oriented Programming, March/April 1992 |
| [D'Souza & Wills 1998] | Desmond D'Souza, Alan Cameron Wills. *"Objects, Components and Frameworks with UML. The Catalysis Approach"* Addison-Wesley, 1998 |

| | |
|---|---|
| [Easterbrook 1991] | Steve Easterbrook. *"Elicitation of Requirements from Multiple Perspectives"* Ph.D. thesis, Department of Computing, Imperial College, London. 1991 |
| [Engels & Groenewegen 2000] | Gregor Engels, Luuk Groenewegen. *"Object-Oriented Modeling: A Roadmap"* In proceedings of "The Future of Software Engineering 2000", Editor: Anthony Finkelstein, International Conference on Software Engineering. |
| [Gamma et al. 1994] | Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *"Design Patterns. Elements of Object-Oriented Software"* Addison-Wesley 1994 |
| [García-Solaco et al. 1996] | Manual García-Solaco, Fèlix Saltor, Malú Castellanos. *"Semantic Heterogeneity in Multidatabase Systems"* In Object-Oriented Multidatabase Systems - A solution for Advanced Applications, Eds: Bukhres, Elmagarmid. Prentice-Hall 1996 |
| [Gosling et al. 1996] | James Gosling, Bill Joy, Guy Steele *"The Java™ Specification Language"* Addison-Wesley 1996 |
| [Gotthard et al. 1992] | Willi Gotthard, Peter C. Lockemann, Andrea Neufeld. *"System-Guided View Integration for Object-Oriented Databases"* IEEE Transactions on Knowledge and Data Engineering, Vol.4, No. 1, February 1992 |
| [Gowing & Cahill 1996] | Brendan Gowing, Vinny Cahill. *"Meta-Object Protocols for C++: The Iguana Approach"* In proceedings of Reflection'96, San Francisco, USA, 1996 |
| [Graham 1993] | Ian Graham *"Object Oriented Methods"* Addison-Wesley, 1993 |
| [Griss et al. 1998] | Martin Griss, John Favaro, Massimo d'Alessandro *"Integrating Feature Modeling with the RESB"* In proceedings of International Conference on Software Reuse (ICSR) 1998 |
| [Hailpern & Ossher 1990] | Brent Hailpern, Harold Ossher *"Extending Objects to Support Multiple Interfaces and Access Control"* IEEE Transactions on Software Engineering 16(11), pp 1247-1257, 1990 |
| [Härder et al. 1999] | Theo Härder, Günter Sauter, Joachim Thomas. *"The intrinsic problems of structural hetergeneity and an approach to their solution"* The VLDB Journal, 8: 25-43, Springer-Verlag, 1999 |
| [Harrison & Ossher 1993] | William Harrison, Harold Ossher. *"Subject-Oriented Programming (a critique of pure objects)"* In proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA) 1993 |
| [Harrison et al. 1996] | William Harrison, Haim Kilov, Harold Ossher, Ian Simmonds. *"From dynamic supertypes to subtypes: A natural way to specify and develop systems"* IBM Systems Journal, 35, 244-256, 1996 |

[Helm et al. 1990] Richard Helm, Ian Holland, Dipayan Gangopadhyay *"Contracts: Specifying Behavioral Compositions in Object-Oriented Systems"* In proceedings of Object-Oriented Programming Systems, Languages and Applications (OOP-SLA) 1990

[Holland 1992] Ian Holland. *"Specifying Reusable Components Using Contracts"* In proceedings of European Conference on Object-Oriented Programming (ECOOP) 1992

[Hutt 1994] Andrew Hutt (Editor) *"Object Analysis and Design. Comparison of Methods"* Object Management Group, John Wiley & Sons, 1994

[IBMa 2000] *"IBM Business Management Workbench"* IBM Corporation, 2000

[IBMb 2000] *"Worldwide Project Management Method"* IBM Corporation, 2000

[Jackson & Zave 1998] Michael Jackson, Pamela Zave. *"Distributed Feature Composition: A Virtual Architecture for Telecommunications Services"* IEEE TSE Special Issue on Feature Interaction, 1998

[Jacobson 1994] Ivar Jacobson. *"Time for a Cease-Fire in the Methods War"* Panel on "Methodology Standards: Help or Hindrance?" In proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA) 1994

[Jacobson et al. 1999] Ivar Jacobson, Grady Booch, James Rumbaugh. *"The Unified Software Development Process"* Addison-Wesley, 1999

[Jacobson et al. 1992] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, Gunnar Övergaard. *"Object-Oriented Software Engineering. A Use-Case Driven Approach"* Addison-Wesley, 1992

[Kiczales et al. 1991] Gregor Kiczales, Jim des Rivières, Daniel G. Bobrow. *"The Art of the Metaobject Protocol"* Massachusetts Institute of Technology 1991

[Kiczales et al. 1997] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, John Irwin. *"Aspect-Oriented Programming"* In proceedings of European Conference on Object-Oriented Programming (ECOOP) 1997

[Kiczales & Lopes 1999] Gregor Kiczales, Cristina Lopes. *"Aspect-Oriented Programming with AspectJ*™" http://www.aspectj.org

[Kilov & Ross 1994] Haim Kilov, James Ross. *"Information Modeling. An object-oriented approach"* Prentice-Hall 1994

[Kim & Seo 1991] Won Kim, Jungyun Seo. *"Classifying Schematic and Data Heterogeneity in Multidatabase Systems"* IEEE Computer, December 1991

[Klas et al. 1996] Wolfgang Klas, Peter Fankhauser, Peter Muth, Thomas Rakow, Erich Neuhold. *“Database Integration Using the Open Object-Oriented Database System VODAK”* In Object-Oriented Multidatabase Systems - A solution for Advanced Applications, Eds: Bukhres, Elmagarmid. Prentice-Hall 1996

[Kristensen & Østerbye 1996] Bent Bruun Kristensen, Kasper Østerbye. *“Roles: Conceptual Abstraction Theory and Practical Language Issues”* Theory and Practice of Object Systems, Volume 2(3), 143-160, 1996

[Kristensen 1997] Bent Bruun Kristensen. *“Subject Composition by Roles”* In proceedings of Object-Oriented Information Systems (OOIS) 1997

[Kuno & Rundensteiner 1996] Harumi Kuno, Elke Rundensteiner *“The MultiView OODB View System: Design and Implementation”* Theory and Practice of Object Systems, Volume 2(3), 203-225, 1996

[Lieberherr 1995] Karl Lieberherr. *“Adaptive Object-Oriented Software. The Demeter Method with Propagation Patterns”* PWS Publishing Company, 1995

[Lopes & Kiczales 1997] Cristina Lopes, Gregor Kiczales. *“D: A Language Framework for Distributed Programming”* Technical Report Number SPL97-010 P9710047, Xerox Parc, 1997

[Lunau 1997] Charlotte Pii Lunau. *“A Reflective Architecture for Process Control Applications”* In proceedings of European Conference on Object-Oriented Programming (ECOOP) 1997

[Maughan & Durnota 1994] G. Maughan, B. Durnota. *“MON: An object relationship model incorporating roles, classification, publicity and assertions”* In proceedings of Object-Oriented Information Systems (OOIS) 1994

[Minsky & Rozenshtein 1987] Naftaly Minsky, David Rozenshtein. *“A Law-Based Approach to Object-Oriented Programming”* In proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA) 1987

[Mowbray & Zahavi 1995] Thomas Mowbray, Ron Zahavi. *“The Essential CORBA: Systems Integration Using Distributed Objects”* Object Management Group, John Wiley & Sons, 1995

[Mulet et al. 1995] Philippe Mulet, Jacques Malenfant, Pierre Cointe. *“Towards a Methodology for Explicit Composition of MetaObjects”* In proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA) 1995

[Navathe & Savasere 1996] Shamkant Navathe, Ashoka Savasere. *“A Schema Integration Facility Using Object-Oriented Data Model”* In Object-Oriented Multidatabase Systems - A solution for Advanced Applications, Eds: Bukhres, Elmagarmid. Prentice-Hall 1996

[Nierstrasz & Tsichritzis 1995] Oscar Nierstrasz, Dennis Tsichritzis. *“Object-Oriented Software Composition”* Prentice-Hall 1995

[Nuseibeh 1994] Bashar Nuseibeh. *“A Multi-Perpsective Framework for Method Integration”* Ph.D. thesis, Department of Computing, Imperial College, London. 1994

[Nuseibeh et al. 1994] Bashar Nuseibeh, Jeff Kramer, Anthony Finkelstein. *“A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification”* IEEE Transactions on Software Engineering, 20(10):760-773, October 1994

[OED 1989] The Concise Oxford Dictionary

[Okamura & Ishikawa 1994] Hideaki Okamura, Yutaka Ishikawa. *“Object Location Control Using Meta-Level Programming”* In proceedings of European Conference on Object-Oriented Programming (ECOOP) 1994

[Ossher et al. 1996] Harold Ossher, Matthew Kaplan, Alexander Katz, William Harrison, Vincent Kruskal. *“Specifying Subject-Oriented Composition”* Theory and Practice of Object Systems, Volume 2(3), 179-202, 1996

[Parnas 1974] D.L. Parnas *“On the criteria to be used in decomposing systems into modules”* Communications of the ACM, 15(12):1053-1058, December 1972

[Pedersen 1989] Claus Pedersen *“Extending Ordinary Inheritance Schemes to Include Generalization”* In proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA) 1989

[Pólya 1957] George Pólya, *“How to Solve It: A New Aspect of Mathematical Method”* 2nd edition, NY, USA, Doubleday, 1957

[Reenskaug et al. 1995] Trygve Reenskaug, Per Wold, Odd Arild Lehne. *“Working with Objects. The OOram Software Engineering Method”* Manning Publications Co. 1995

[Rumbaugh et al. 1991] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen. *“Object-Oriented Modeling and Design”* Prentice-Hall, 1991

[Shaw & Garlan 1996] Mary Shaw, David Garlan. *“Software Architecture. Perspectives on an Emerging Discipline”* Prentice Hall, 1996

[Sheth & Larson 1990] Amit Sheth, James Larson. *“Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases”* ACM Computing Surveys, Vol. 22, No. 3 1990

[Sheth et al. 1993] Amit Sheth, Sunit Gala, Shamkant Navathe. *“On Automatic Reasoning for Schema Integration”* International Journal of Intelligent and Cooperative Information Systems, Vol. 2 No. 1 1993

| | |
|---|---|
| [Shilling & Sweeney 1989] | John J. Shilling, Peter F. Sweeney. *“Three steps to views: Extending the object-oriented paradigm”* In proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA) 1989 |
| [Shlaer & Mellor 1988] | Sally Shlaer, Stephen Mellor. *“Object-Oriented Systems Analysis. Modeling the World in Data”* Yourdon Press Computing Series, 1988 |
| [Siegel 1996] | Jon Siegel. *“CORBA Fundamentals and Programming”* Object Management Group, John Wiley & Sons, 1996 |
| [Smith & Ungar 1996] | Randall Smith, David Ungar. *“A Simple and Unifying Approach to Subjective Objects”* Theory and Practice of Object Systems, Volume 2(3), 161-178, 1996 |
| [Spaccapietra & Parent 1994] | Stefano Spaccapietra, Christine Parent. *“View Integration: A Step Forward in Solving Structural Conflicts”* IEEE Transactions on Knowledge and Data Engineering, Vol. 6, No. 2, April 1994 |
| [Spaccapietra et al. 1992] | Stefano Spaccapietra, Christine Parent, Yann Dupont. *“Model Independent Assertions for Integration of Heterogeneous Schemas”* VLDB Journal, 1, 81-126 1992 |
| [Stonebraker et al. 1991] | Michael Stonebraker, Lawrence Rowe, Bruce Lindsay, James Gray, Michael Carey, Michael Brodie, Philip Bernstein, David Beech. *“Third-Generation Database System Manifesto”* Object-Oriented Databases: Analysis, Design & Construction, Elsevier Science Publishers, 1991 |
| [Stroustrup 1991] | Bjarne Stroustrup *“The C++ Programming Language Second Edition”* Addison-Wesley 1991 |
| [Szyperski 1998] | Clemens Szyperski. “*Component Software. Beyond Object-Oriented Programming”* Addison-Wesley, 1998 |
| [Tarr et al. 1999] | Peri Tarr, Harold Ossher, William Harrison, Stanley Sutton. *“N Degrees of Separation: Multi-Dimensional Separation of Concerns”* In proceedings of the International Conference on Software Engineering (ICSE) 1999 |
| [Tarr & Ossher 2000] | Peri Tarr, Harold Ossher. *“Hyper/J™ User and Installation Manual”* http://www.research.ibm.com/hyperspace |
| [Turner 1999] | C. Reid Turner *“Feature Engineering of Software Systems”* PhD Thesis, Department of Computer Science, University of Colorado, 1999 |
| [Turner et al. 1999] | C. Reid Turner, Alfonso Fuggetta, Luigi Lavazza, Alexander L. Wolf *“A Conceptual Basis for Feature Engineering”* In the Journal of Systems and Software, December 1999. |
| [Ungar & Smith 1987] | David Ungar, Randall Smith. *“Self: The Power of Simplicity”* In proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA) 1987 |

|  |  |
|---|---|
| [UML 1999] | *"OMG Unified Modeling Language Specification (draft)"* Version 1.3. beta R7, June 1999 |
| [Vlissides 1998] | John Vlissides. *"Pattern Hatching. Design Patterns Applied"* Software Patterns Series, Addison-Wesley 1998 |
| [Walker 2000] | Robert J. Walker. *"Eliminating Cycles in Composed Class Hierarchies"* Technical Report TR-00-07, Department of Computer Science, University of British Columbia, 2000 |
| [Warmer & Kleppe 1999] | Jos Warmer, Anneke Kleppe. *"The Object Constraint Language. Precise Modeling with the UML"*. Addison-Wesley, 1999 |
| [Wieringa et al. 1996] | Roel Wieringa, Wiebren de Jonge, Paul Spruit. *"Using Dynamic Classes and Role Classes to Model Object Migration"* Theory and Practice of Object Systems, Volume 1(1), 61-83, 1995 |
| [Wirfs-Brock et al. 1990] | Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener. *"Designing Object-Oriented Software"* Prentice-Hall 1990 |
| [Zave 1997] | Pamela Zave *"Classification of Research Efforts in Requirements Engineering"* ACM Computing Surveys XXIX(4):315-321, 1997 |
| [Zave 1999] | Pamela Zave *"FAQ Sheet on Feature Interaction"* available from htttp://www.research.att.com/~pamela/faq.html |

# Appendix A: Partial Illustrations of UML Metamodel

This appendix presents a reproduction of the class models that represent the UML metamodel from a different perspective to how they are described in [UML 1999]. Here, each construct that is interesting for composition (generally, all composable elements) is presented from its own perspective.

## Package

Figure 142 illustrates the part of the UML metamodel that refers to Packages. The definition of Subject for the purposes of this thesis is as specified in "Scope of Work" on page 72, and is that:

> "a subject is a stereotyped Package, stereotyped for the purposes of restricting its contents to subjects, classifiers, associations, generalizations, dependencies, constraints and collaborations".

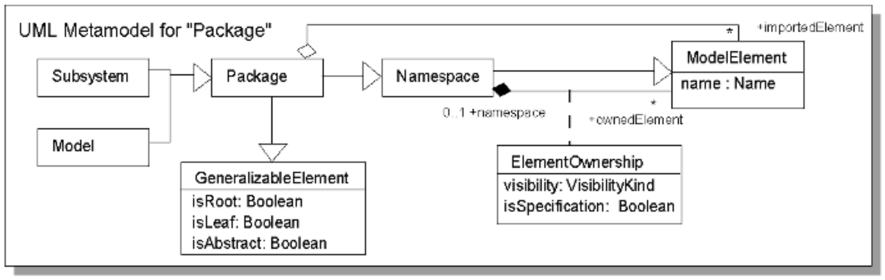


**Figure 142: Partial UML Metamodel for Package**

This stereotype definition restricts the kinds of model elements that may be "owned elements" (see Figure 142). Further scoping restrictions for these elements are discussed with their detail.

## Classifier

Figure 143 illustrates the part of the UML metamodel that refers to Classifiers. For the purposes of scoping this work, the only classifiers considered in this thesis are Class, Interface and Datatype.

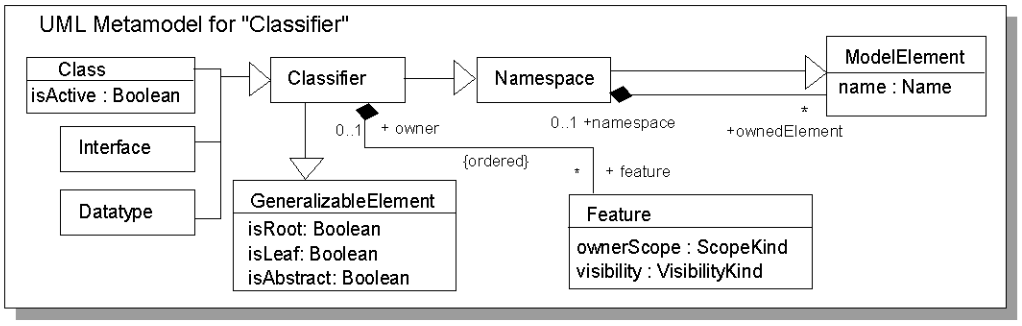
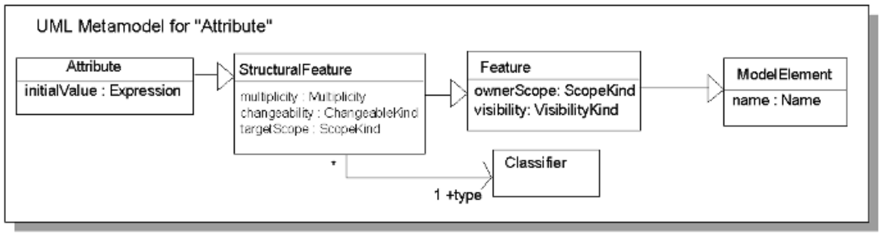


Figure 143: Partial UML Metamodel for Classifiers

## Attribute

Figure 144 illustrates the part of the UML Metamodel that refers to Attributes



UML Metamodel for "Attribute"
Attribute
initialValue : Expression
StructuralFeature
multiplicity : Multiplicity
changeability : ChangeableKind
targetScope : ScopeKind
Feature
ownerScope: ScopeKind
visibility: VisibilityKind
ModelElement
name : Name
*
Classifier
1 +type

Figure 144: Partial UML Metamodel for Attributes

## Operation

Figure 145 illustrates the part of the UML Metamodel that refers to Operations.

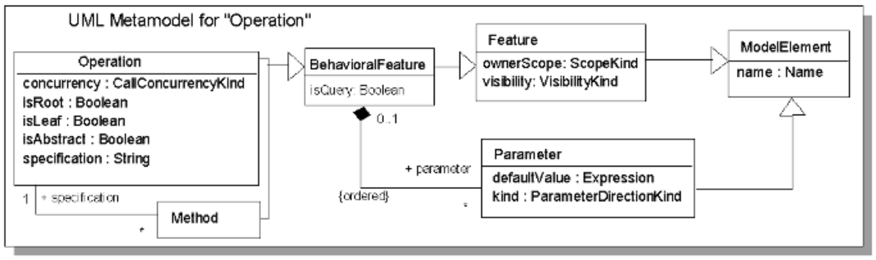


Figure 145: Partial UML Metamodel for Operations

## Relationship

Figure 146 illustrates the part of the UML Metamodel that refers to Relationships.

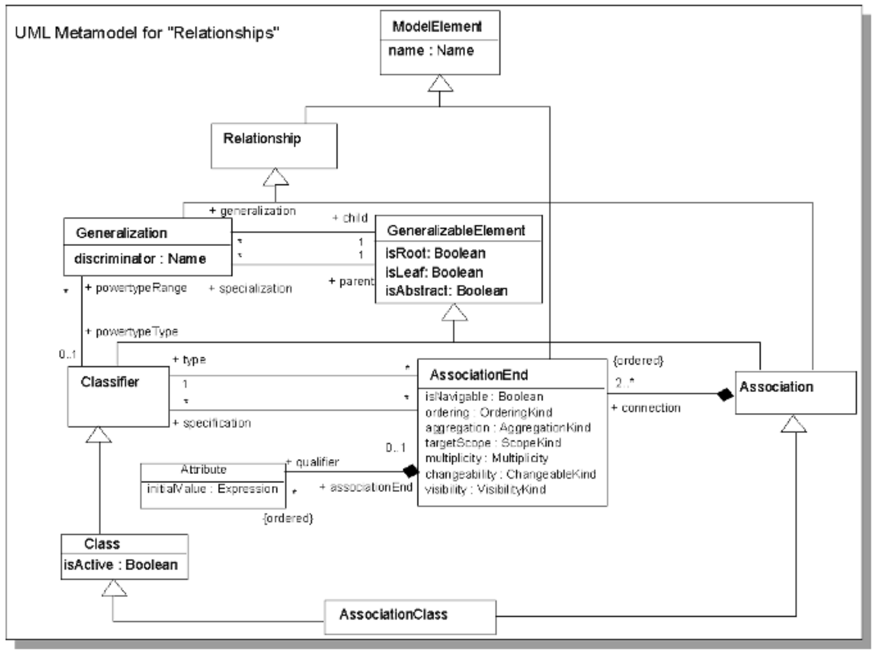


**Figure 146: Partial UMl Metamodel for Relationship**

## Dependency

Figure 147 illustrates the part of the UML Metamodel that refers to Dependency

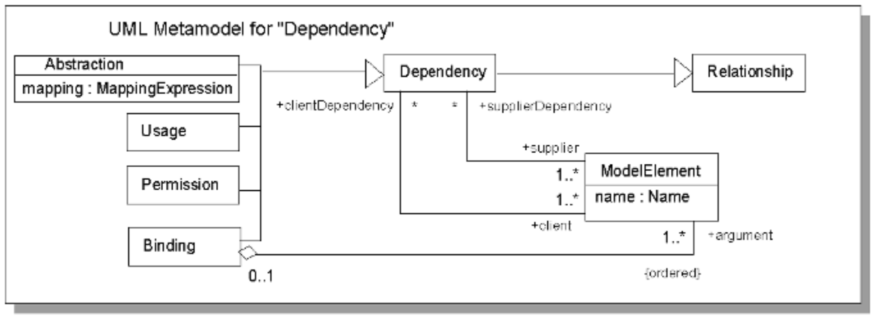


**Figure 147: Partial UML Metamodel for Dependency**

## Constraint

Figure 148 illustrates the part of the UML Metamodel for Constraint.

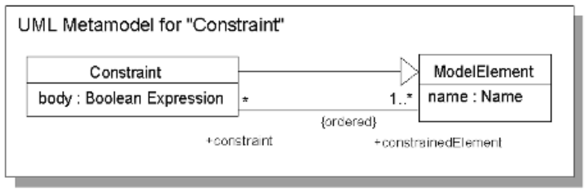


**Figure 148: Partial UML Metamodel for Constraint**

## Collaboration

Figure 149 illustrates a partial specification of Collaboration as defined by the UML.

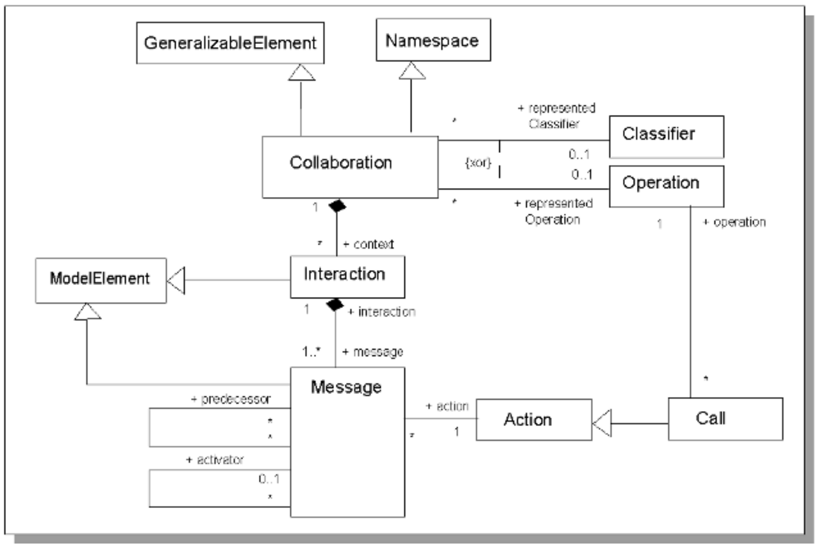


**Figure 149: Partial UML Metamodel for Collaborations**

Collaborations also provide a context for participants playing different *roles* within the collaborations. Figure 150 illustrates a partial meta-model for collaborations that shows the metaclasses that represent roles for associations and classifiers. These roles are in the context of sending and receiving messages.

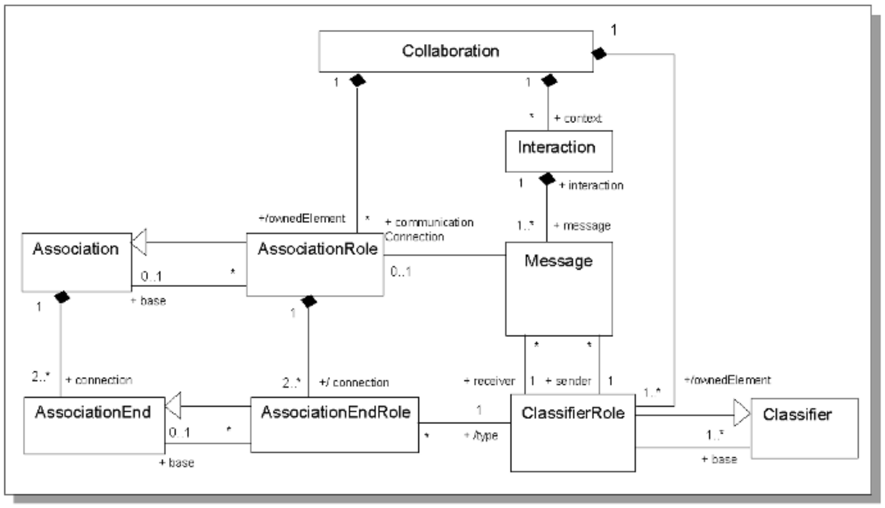


**Figure 150: Partial UML Metamodel for Collaboration Roles**