Citation for published version:
Batenin, A 2006, Subjectivity and ownership: a perspective on software reuse. Computer Science Technical Reports, no. CSBU-2006-15, Department of Computer Science, University of Bath.

Publication date:
2006

Link to publication

Department of
Computer Science

UNIVERSITY OF
BATH

Technical Report

PhD. Dissertation: Subjectivity and Ownership: A Perspective on Software Reuse

Adam Batenin

Technical Report 2006-15 December 2006
ISSN 1740-9497

Contact Address:
Department of Computer Science
University of Bath
Bath, BA2 7AY
United Kingdom
URL: http://www.cs.bath.ac.uk

ISSN 1740-9497

Subjectivity and Ownership: A Perspective on Software Reuse

submitted by

Adam Batenin

for the degree of Doctor of Philosophy

of the

University of Bath

2005

Signature of Author ....

Adam Batenin

Summary

Construction of software from existing components is a long standing goal of software engineering. Cost is an important factor distinguishing a component created for reuse from a component built for a particular application. Construction of reusable components requires investment that the developer can recoup only by reuse or by marketing the component for reuse by others. Much of today's software construction is not aimed at markets but to fulfill specific objectives set out in requirements. This thesis proposes a means of constructing more reusable software, including software that is not destined for component markets, by combining subjectivity and ownership.

Subjectivity, in the form of Subject-Oriented Programming, is a software development technology in the area of Aspect-Oriented Software Development that enables software decomposition into partially overlapping modules known as subjects. Subjects enable the creation of modular implementations of use cases, features and systemic requirements, all within the familiar environment of object-oriented programming. Anomalous interactions during stateful inter-subject interactions are an acute problem in reuse and for modular subject development. In the worst cases, they require either patching or invasive modifications. To tackle this problem, we propose annotations in the form of ownership types. In object-oriented programming, Ownership Types have been proposed as a solution to the endemic problem of aliasing. Structured use of aliases facilitates the construction of robust software that ensures representation encapsulation and supports modular reasoning. The subject-oriented approach to problems previously modelled using object-oriented idioms requires a novel solution to the concept of ownership. Subjects do not have a representation; instead, ownership types annotate the ownership structure of object collaborations implemented by subjects.

In this thesis we propose the Subjective Alias Protection System or SAPS. It is a tool both for subject design and reuse. At a small syntactic overhead, SAPS supports the design of well structured subjects whose classes ensure representation containment. SAPS improves the reusability of subjects: Subjective Ownership Types are per-object annotations of the places an object may be referenced or modified. Our extensions to subject composition rules constrain subject and class reuse to meaningful cases and can prevent compositions leading to anomalous interactions. SAPS facilitates modular development of subjects because aspects of subject effect on state can be observed from the points of inter-subject interaction. Finally, Subject-Oriented Programming with SAPS can address more concerns than is possible without it.

Acknowledgements

This dissertation is dedicated to my mother, Larissa, who has supported me throughout and always believed in my eventual success. Her unwavering support helped me tremendously during the difficult periods. Thank you to my stepfather, David, for encouraging me to succeed.

I would like to thank Peter Wallis, my first supervisor, who opened for me the door to Computer Science research and stimulated my imagination. I would like to give a big thank you to Eamonn O'Neill, my second supervisor, who taught me to think more critically and helped to understand the scientific process.

My experience of Bath was enhanced by Michael and Heather; they are kind and generous people who made me part of their family. I will always have colourful memories of the Resident Tutor system at Bath. Anne W, Cat, Si, Tom, Eliana, Subbu, David G and Isobel are just some of the wonderful people that I had the pleasure of working and partying with.

A special thank you goes to Lucy, Sunniva and Edwige.

The PhD experience had its ups and downs but coming to the office was always an uplifting (and at times a distracting) experience. I have been lucky to share my work time with many brilliant colleagues including Dave P, Bill, Natee, Wafaa, Joy, Jerry, Tracy, Matt; and in the later years Adam D, Andy H, Manu, Sirapat, Peiyi, Ben, Emma, Owen and Vas. Last but not least I have benefited from the support and advice given by Bath Computer Science staff including Angela, James, Marina, Peter and Julian.

Contents

List of Figures

Chapter 1

Introduction

In this thesis we propose the Subjective Alias Protection System (SAPS) – a synthesis of Subject-Oriented Programming and an Alias Protection System. Subject-Oriented Programming (SOP) [49] is a programming paradigm that builds on the strengths of object-oriented programming by introducing *subject* as a new kind of module. Subjects abstractly modularise many concerns that are difficult to modularise using object-oriented programming technology. Each subject is an ordinary object-oriented program and subject interaction occurs at join points – key points in subject structure. These properties make subjects very good at cleanly separating many functional and implementation domain concerns.

These properties also make subjects very reusable, but we will show that subject reusability comes at a cost. The absence of an abstract functional interface is both a positive and a negative reuse factor. On the one hand, interfaces facilitate structured reuse that guarantees desirable correctness properties, and on the other, they make it difficult to extend or modify software in ways that were not intended by the original developer. Subject interaction can lead to undesirable interference that, at worst, requires invasive redefinitions. Reuse of stateful subjects is expensive because the reuser must understand the implementation in detail in order to reuse successfully.

To address these problems we specify Subjective Ownership Types (SOT). SOT are a new type system that supersedes the existing types in subjects. They are inspired by the Ownership Types for Flexible Alias Protection [23]. In object-oriented programming, Alias Protection Systems are an attempt to address the problems caused by proliferation of object aliases [57]. This continues to be the purpose of SOT when viewing programs one subject at a time. We propose SAPS as a combination of SOT and the necessary extensions for subject interaction. SAPS constrains subject composition (interaction) in order to ensure that only elements with mutually compatible types are joined. We will show that SAPS makes significant contributions in a number of areas: both modular development of subjects and subject reuse are more feasible than with SOP alone; some interaction problems are addressed directly while other anomalies are easier to detect because the extent of object aliasing is explicit in the types of elements at join points; and it is possible to use SOP to address new kinds of concerns.

In order to introduce our work, Section 1.1 establishes our position on software reuse. The position motivates us to understand reuse better and guides us towards proposing SAPS as a pragmatic reuse technology. The progression from the reuse position to SAPS is detailed in Section 1.2. Section 1.3 explains the objectives of this thesis and describes the way it is organised.

1.1 Position on Software Reuse

Our work is motivated by the technical challenges underlying software reuse. Software reuse has received a lot of research attention and there have been success stories [14, 113]. However, reuse remains a topic for research because software engineers are constantly facing new challenges as software pervades all areas of human activity and challenges grow in scale. To set out our reuse position we will play out a typical software development scenario that goes on in many software houses across the world.

Consider an application developer who has been tasked with creating a program to address the needs of some client. Our developer faces two constraints common to many of today's projects. The first is programming in a mainstream object-oriented programming language. The second is time. The time it takes the developer to create the product is a major contributor to cost. To reduce overall costs the programmer is prepared to purchase components, use application frameworks, scavenge code and apply the latest methods in software engineering. The developer takes pride in his work and wishes to create well structured software that will be easy to maintain, predictable during evolution and reusable in future projects. However, time is the overriding concern and corners can be cut to ensure that the product is delivered on schedule.

Now, an interesting question: what is the chance that the code he writes can be reused in future projects? We believe that the tug-of-war between the interests of the developer on a schedule and a reuser in a hurry are at the core of the software reuse problem.

The developer must complete the project on time which means that all good design ideas that aid future code reuse but cost time may not be adopted. Many design guidelines, although valuable in theory, are dropped by programmers in practice when they require effort. This effort is only rewarded in future maintenance, evolution and reuse tasks. For example, separating types from their implementations or using accessor methods for field look-up and update are known ways of improving the separation of concerns and, therefore, reusability. However, unless enforced technologically or institutionally, developers will often ignore good practices in order to save time.

The reuser would like to reduce costs by assembling code from pre-existing components rather than writing code from scratch. The problem is one of finding code to reuse, possibly extracting it from another application, and adapting it to meet the needs of the project. The reuser's job is made more difficult as the result of shortcuts taken by previous developers.

We believe that improving opportunities for reuse depends on ideas that are of value to the original developer *and* facilitate future reusability. Reuse ideas stand more chance of being accepted by practitioners when they are beneficial to the original developer. Our attempts to improve opportunities for reuse are influenced throughout by this position.

1.2 From Reuse Problems to SAPS

Ultimately motivated by issues in software reuse, this Section describes the progression towards the Subjective Alias Protection System. Software reuse is achieved through construction of reusable software [82]. In order to make software more reusable it is necessary to separate all pertinent concerns. Separation of concerns is of value to the software developer because tackling one subproblem at a time is easier than tackling the whole problem at once. We will show that using the current mainstream programming languages such as Java [45], the time-pressured developer cannot cleanly separate all pertinent concerns. Enter Multi-Dimensional Separation of Concerns (MDSOC) [122]. MDSOC proposes to organise software into multiple dimensions of concerns. By enabling the modularisation of all concerns along all dimensions that developers believe to be important we can reduce the cost of software development over the lifecycle *and* improve opportunities for reuse.

The pursuit of the MDSOC idyll is the domain of technology broadly referred to as Aspect-Oriented Software Development (AOSD) [90]. In AOSD, concerns are realised by modules called aspects. Instead of interacting by message passing, aspect interaction is based on so-called join points. Join points are defined in different ways [66], but usually they are either programming language constructs or arcs in the program's dynamic call graph. Join point interfaces enable separation of concerns for functional and non-functional requirements. Having identified the join point interfaces, aspects can be developed independently and integrated using aspect-oriented compilers. From a reuser's perspective, code associated with pertinent concerns from past projects is abstract and modular, making it more easily reusable than when programmed with conventional technology.

Subject-Oriented Programming is one strand of AOSD that adheres well to the MDSOC model. A subject is a module denoting an aspect. SOP concepts are realised in the programming language Hyper/J [121]. This language combines the modelling potential required for separating many concerns with the familiarity of mainstream object-oriented programming: subjects are written in pure Java. Each subject has a very large number of join points determined by the underlying language. Together these form its potential interface to other subjects. The actual interaction points, along which subjects communicate, only become apparent when subjects are composed, i.e. subjects do not explicitly publish a composition interface.

Our experience with programming Hyper/J has highlighted the strengths and weaknesses of evolution and reuse in the SOP paradigm. Each subject is relatively easy to understand as it addresses either a single concern or a well-defined concern set. However, relationships between concerns and subject interaction are often complicated. We will show that the difficulty of understanding all consequences of communication can lead to unwanted interactions or *interaction problems*. These affect subject reuse, potentially limiting the range of concerns to which Subject-Oriented Programming can be realistically applied.

Interaction problems are a topic of our investigation. Within the range of interaction problems, there are those that can be solved by re-specifying inter-subject interaction and those which require invasive modifications to subjects. Re-specification of interaction affects the 'cement' between 'building blocks', whereas changes to subjects affect the 'building blocks'. The latter is a lot more expensive to correct, making reuse uneconomical. Modular subject development is also affected by this problem; independent design can begin only when the effect of join point interaction on state is well understood.

One way to facilitate structured reuse is to introduce formal composition interfaces. That is, to allow join point interaction but only at predefined join points. However, subjects are meant to be reusable in ways not anticipated by their original developers and, for this reason, we must look for an alternative solution. Instead, we propose to help subject composers to understand the effect of composition on object state by making explicit the way subjects use objects. Our challenge is compounded by the reuse motivation problem stated in the reuse position: any solution must benefit the original developer as well as future reusers. We believe that Alias Protection Systems (APSs) satisfy our reuse position. APSs are a solution to problems caused by unstructured object aliasing in object-oriented programming. APSs constrain object aliasing to enable modular reasoning (on objects). But each subject is an object-oriented sub-program and design-in-the-small is a purely object-oriented activity. An APS is useful to the subject designer because it helps him to structure subjects better in order to avoid aliasing problems. An APS is also useful to the composer for it annotates the elements at composition join points, thereby helping to explain the effect that one subject has on another subject.

Inspired by Flexible Alias Protection, we propose the Subjective Alias Protection System. Subjects can be composed when Subjective Ownership Types at join points are mutually compatible. The new emphasis on aliasing issues helps to prevent some interaction problems. It also aids detection of other interaction problems by helping the composer to understand the effect of subject interaction on state.

1.3 Thesis Outline

1.3.1 Aims, Objectives and Limitations

The main aim of this thesis is to introduce SAPS as a reuse technology that has value to the original developer of software. The secondary aim is to tackle interaction problems experienced in subject-oriented programming. To motivate these aims and defend the thesis we propose a sequence of objectives:

1. Develop an understanding of the factors affecting software reuse and how to construct reusable software.

2. Review the state of the art in Aspect-Oriented Software Development with the goal of identifying the technological trends that best meet our reuse position.

3. Investigate the phenomenon of interaction problems in Subject-Oriented Programming and identify how they may be tackled.

4. Propose a set of requirements for an APS for Subject-Oriented Programming.

5. After presenting SAPS, show that SOT are a useful APS for subject design; explain how SAPS enhances subject-oriented software development; and demonstrate that SAPS addresses interaction problems.

The material presented in this thesis is of a conceptual nature, so our approach is predominantly informal. We emphasise the software engineering issues rather than a type system because we believe that an explanation of the relationships between reuse, interaction problems, subjectivity and ownership must come first. A rigorous formal model that follows on from the conceptual understanding is future work.

SAPS is not specific to any programming language, although it is expected that subjects will be developed in an object-oriented language that combines subclassing with subtyping and has single inheritance. Our subject composition semantics are based on the observed semantics of Hyper/J and our subject composition language is interoperable with the core of Hyper/J.

1.3.2 Conventions

Examples are presented in Java pseudocode. We use the following typeface conventions. When writing code fragments we use the `typewriter family` of fonts. Where programming languages use common English words for operators and keywords, we use the **boldface series** to distinguish from the words' general usage. Important words and phrases are emphasised with *italics*. The sans serif fonts are used in the presentation of SAPS concepts and to refer to ownership contexts. For displaying mathematical expressions the *slanted font* is used.

1.3.3 Chapter by Chapter

Chapter 2: Evolution and Reuse

Having already established our position, this Chapter describes the factors influencing reuse and the challenges in constructing reusable software. It reviews object-oriented programming as a reuse technology from the perspective of programmers of mainsteam programming languages.

Chapter 3: Advanced Separation of Concerns

In moving beyond OOP, this Chapter looks at research in the area of Aspect-Oriented Software Development. Multi-Dimensional Separation of Concerns is presented as a model for understanding many of the problems in software engineering. We evaluate AOSD technology based on the capability for separating two kinds of concerns: feature concerns from the problem domain, usually associated with object collaborations; and aspectual concerns from the solution domain that are difficult to modularise with conventional programming languages. Subject-Oriented Programming can modularise collaborations and many aspectual concerns. It also satisfies our position on reuse.

Chapter 4: Interaction Problems in Subject-Oriented Programming

In this Chapter we relate our own experience and that of other researchers with regard to interaction problems. The problems we identify are categorised based on the kind of solution they require. The first interaction problem can be addressed by reformulating the composition specification. The second by extending SOP with more powerful composition rules. The third requires invasive subject modifications and is caused by an unanticipated state change in an object. This anomaly is hard to detect because data concerns are scattered across subjects.

We propose to develop an Alias Protection System for SOP in order to encourage subject developers to create well structured subjects and to help subject composers to understand the effect of subject interaction on state.

Chapter 5: Alias Protection and Subjectivity

This Chapter presents the state of art in Alias Protection Systems in object-oriented programming and sets out the requirements for a system that is suitable for Subject-Oriented Programming. SOP decentralises class development, letting each subject define abstractions from its own viewpoint. The decentralised style of software development makes existing APSs unsuitable. The requirements for an APS in SOP lay the foundations for the Subjective Alias Protection System.

Chapter 6: SAPS – Subject Design

This Chapter presents the principles of Subjective Ownership Types. It describes the way SOT are used in subject design and describes the way types are checked.

The containment properties of SOT are similar to those of Ownership Types proposed by Clarke et al [23]. Ownership Types, like other APSs, are based on a centralised definition of classes. Subjects define new classes, most of which do not have a centralised view. We use the concept of centralisation to partition classes into two hierarchies called composable and uncomposable. Uncomposable classes can use an ownership type system very similar to that proposed by Clarke. Composable classes require a new type system. Instead of ownership parameters, we propose explicit and unknown context identifiers for labelling object owners.

Chapter 7: SAPS – Subject Composition

At the core of a subject-oriented language like Hyper/J is a subject composition model. We extend the subject composition model discussed by Ossher et al [95] with contexts and describe what it means to compose elements annotated with Subjective Ownership Types. The model is extensible allowing new kinds of composition rules to be defined. The challenges include the specification of unknown context resolution. This is a mechanism by which partial knowledge of ownership structure specified in one subject is filled in by composing with other subjects. The output of composition is a new subject containing the synthesis of concerns implemented by the input subjects.

Chapter 8: Evaluation

The evaluation presented in this Chapter demonstrates that SAPS can eliminate some interaction problems entirely and can help to detect other interaction problems by annotating the effect of subject interaction on state. The utility of SAPS to the subject-oriented developer is shown through a range of design problems. We show also how to use our system to express security concerns that cannot be represented in SOP without SAPS.

Chapter 9: Conclusions and Future Work

In the final Chapter, SAPS is reviewed in terms of its contributions to reuse, interaction problems, and as a tool for improving the design of subject-oriented programs. We conclude with a discussion of plans to extend SAPS with additional aliasing capabilities.

Chapter 2

Evolution and Reuse

The position on reuse, outlined in Section 1.1 on page 2, is that improving opportunities for reuse depends on technologies that also have value to the original developer. There are at least two ways of making the construction of reusable software beneficial to the developer. The first is building reusable artifacts with the aim of marketing to a wide audience. The second is better separation of concerns within software; that is, not to build software for reuse markets but to make software more reusable as a consequence of improved modularity.

The position on reuse also describes the pressures on developers which make code less reusable, highlighting the significance of making all software more reusable and not just software which was intended for reuse. This dissertation takes the second approach above: to seek improvements in modularity as the way of improving reusability. To defend the approach, the present Chapter describes software evolution and reuse. Section 2.1 defines reuse and reusable software. Section 2.2 explores the modularity issues in reuse. Section 2.3 presents the challenges of constructing software for reuse. Mainstream software development is presently dominated by object-oriented programming (OOP). Object-oriented programming was touted as a means of improving opportunities for reuse. In Section 2.4, we discuss its successes and failures in that respect.

2.1 What Is Reuse?

Software reuse has been proposed as a solution to the software crisis – the problem of building large, reliable software systems in a controlled and cost effective way [82]. The benefits of reuse are improved quality of the finished product from reuse of pre-tested artifacts and reduced development costs due to economies of scale – the development cost of a single reusable artifact is amortised by all who integrate it in their products. Software reuse is difficult because useful reuse abstractions are typically complex. The programmer must either be familiar with the artifacts or take time to study and understand them. Either way, it must be cheaper to reuse the software artifact than to develop software from scratch [70].

It is important to distinguish software reuse from reusable software. Software reuse is the activity that takes place afterward, when software was initially created in the past. To best support this, reusable software must be created beforehand in such a way that it is easy to reuse later.

In its most general sense, reuse is the act of taking existing artifacts related to the creation of software and incorporating them in a new project or extending software with new functionality.

The types of artifacts that can be reused are not limited to pieces of code. It is possible to reuse requirement specifications, design patterns, test cases, and anything else related to the construction of software. However, when talking about reuse we usually think of code, and that is what we will mean when discussing reuse.

There exists a difference between the act of reuse and usage of software by a client. For instance, according to Poulin [103], the use of high-level languages, software development tools, applications and application generators is not reuse because an applications developer is generally not expected to write this software. Categories which represent reuse are:

- The first use of a component but not the subsequent uses.
- Code from utility, domain-specific and corporate libraries.

We broadly agree with Poulin's categorisation but would like to include software evolution with the aim of incorporating new requirements. We define software evolution as the process by which systems are extended with new code due to changing requirements. The new code is called the extension. Poulin [103] does not count evolution as reuse because strictly it does not involve using code in an unrelated project. But in the 'real world', evolution and reuse often look like two sides of the same coin. Consider the way Meyer defines reusable components [87]:

> "a software element that must be usable by developers who are not personally known to the component's author to build a project that was not foreseen by the component's author."

This definition readily applies to software evolution. For the following reasons, we are inclined to include evolution in forthcoming discussions on reuse:

- The person creating the extension is not necessarily the original author and therefore may be unfamiliar with the application.
- The adaptation of the extended artifact to accommodate the extension is often unanticipated with respect to the original requirements, requiring the original program to be adapted to accommodate the extension.
- The extension writer invests time in creating the extension instead of reconstructing the application from the ground up. Just as when one reuses code in an unrelated application, the extension writer must believe that understanding the original program takes less effort than rewriting it.
- As often happens, documentation may be absent or hopelessly out of date.
- In object-oriented programming, inheritance is associated both with seamless evolution for creating families of types and with code reuse [101].

Impediments to software reuse are technical and non-technical. Although the emphasis of this dissertation is on technical factors, the non-technical factors are also reviewed.

Organisations are generally all too happy to cut costs. In the experience of Schmidt [110] organisations would like to reward internal reuse efforts but a number of non-technical factors conspire to make reuse hard:

Organisational. Development, deployment and support of reusable artifacts requires a deep understanding of the application developer's needs and business requirements. In a large organisation with many projects the number of reusable artifacts increases, making it harder to structure an organisation to provide interaction between teams.

Economic. Creation of reusable assets requires investment which needs to be charged for each project. Organisations find it difficult to institute appropriate taxation on reused artifacts when reuse departments are responsible for balancing their books.

Administrative. It is common for developers to scavenge classes or functions from existing programs developed within their immediate workgroup. However, it is harder to catalogue, archive and retrieve reusable assets across multiple business units in a large organisation.

Political. Rivalry between business units may stifle reuse of artifacts developed by other units when it is perceived as a threat to job security or influences the balance of power.

Psychological. The 'not invented here' syndrome is ubiquitous in many organisations. Enforcement of reuse practices is seen as management lacking confidence in the developers' technical ability.

Reuse as a multi-organisational problem requires a community of developers who are prepared to share ideas, tools, methods and code. However, sharing is not traditionally an ethic of commercial companies. On the contrary, companies prefer to keep their products proprietory in order to maintain competitiveness [80].

2.2 Modularity and Reuse

In order to understand reuse problems it is important to understand what the developers want to reuse. Reuse is usually discussed in terms of particular modular artifacts such as functions and classes, whereas programmers generally wish to reuse code associated with concerns. Modularity is at the core of reusability; getting it right will have great impact on reusability. Maintenance significantly impacts reusability. We argue that improving the traceability of requirements in designs can reduce the negative impact of evolution and facilitate the reuse of code associated with those requirements.

2.2.1 Modularity

Decomposing artifacts into smaller parts is at the core of software development. We decompose systems into modules because tackling problems one module at a time is easier than tackling the whole problem at once. In terms of software, a system is modular when each activity of the system is performed by exactly one module, and when the inputs and outputs of each module are well defined [99]. An activity can be understood as code which executes in response either to a client or system requirement.

Parnas questioned the criteria we use for decomposing systems into modules [97]. He stated that modules should hide difficult design decisions or design decisions which are likely to change. Applied to data representation, this principle is the foundation for abstract data types (ADTs) and is at the core of object-oriented programming. The module user or client is interested in what the module does and not how it does it. The functionality of the module is accessed through an interface which does not reveal the way the module is implemented. The process of modular decomposition continues until each module in the system has a clear purpose.

The complete set of modules exhibits a hierarchical structure. At the root is the whole system and each module is composed of modules below it in the hierarchy. Parnas observed that better reuse can be achieved if modules higher up in the hierarchy use modules lower down but not the other way around. This way the modules closer to the root can be removed and a new tree grown using the low level modules at the leaves.

Modularity affects evolution and reusability directly [17]. The concerns which the developer chooses to modularise will be easy to maintain, evolve and reuse. Other concerns, which were not deemed important or which were not made modular for one reason or another will be harder to reuse. A number of factors conspire to make the initial choice of modules less than straightforward:

- The technology must enable separation of concerns identified as important. It is well known that it is possible to write a program in any language that is general enough, but some languages are better suited to separating certain kinds of concern. For example, in object-oriented programming, inheritance can be used seamlessly to introduce a new variant of a type. Dynamic dispatch – the technology that makes this possible – can be simulated in a procedural programming language. However, the procedural programmer will not build dynamic dispatch into the program before it is needed.

- There are many concerns, which makes it hard to determine which ones to modularise. It is hard to identify those concerns which are important. For example, at an early stage in the specification of a matrix manipulation system, the developer is concerned with matrix operations available to a client. The set of available operations may change. If this happens, it would be nice to introduce new operations without invasive changes to existing code. For this reason, the developer considers treating each matrix operation as a module. For reasons of efficiency, the implementation of each operation is tied closely to the matrix implementation. There may be one algorithm for sparse matrices and one for full matrices, or a single algorithm that treats both kinds of matrix the same. Now the developer believes that each kind of matrix should be a module.

- Having separated concerns identified as important it should be possible to compose modules cheaply and predictably. When reuse of modules becomes common, relatively little time will be spent writing new modules, and most of the programming effort will lie with combining modules [59]. Composition is part of the cost of reuse. As the time spent adapting and debugging the interaction increases, so reuse of those modules becomes less appealing.

Modularity clearly affects reuse, but what affects modularity? The choice of modules is guided by the programming technology. Reusable artifacts are associated with what the underlying programming language determines as modules.

2.2.2 Reuse Artifacts

Reuse fundamentally depends on the reuser's ability to extract code. If the concern is realised as a modular artifact from a library then no work is necessary. Otherwise, the reuser must disentangle the code in order to reuse it. In order to be cost-effective, reuse should not involve major modifications.

In mainstream programming technology, functions, procedures or components are modular artifacts. Programming technology dictates those concerns that become modular and those that do not have a modular form, instead becoming implicit or tightly integrated in components and architectures.

Functions are the most fine-grained modular artifacts of reuse that we distinguish. For example, many cryptographic libraries are full of functions that compute factors of large numbers. Pure functions are some of the easiest things to reuse because they have single entry and exit points, do not modify global data and offer referential transparency. Referential transparency allows a pure function to be replaced by its value which means that a function can be referenced anywhere without adverse consequences [37].

Procedures embody elements of functionality that may depend on or modify some global state. By comparison, pure functions are easier to reuse because in order to reuse procedures one also has to understand the procedure's effect on shared state. In object-oriented programming, non-trivial procedures are analogous to object collaborations. The state is not global but instead is attributed to objects involved in the collaboration. Collaborations are hard to reuse because objects are generally associated with multiple collaborations [86]. Reusing collaborations involves factoring out all other concerns attached to the objects. For example, graph traversal algorithms are useful in many applications. Among the algorithms which can be applied to an arbitrary graph is the computation of the number of unconnected subgraphs. This algorithm should be readily reusable in many applications but the objects playing the roles of vertices or edges also contain other behaviour that is hard to separate. Programming languages and other technology for improving collaboration modularity are reviewed as part of Chapter 3 (page 22) on Advanced Separation of Concerns.

Components are aggregations of functions and procedures. They present interfaces that let clients access their functionality and customise the components to address application requirements. The thing that characterises components and makes them different from sets of related functions and procedures is the sharing of a representation – a common implementation that remains hidden beneath the facade. Components can be large or small. At one extreme are common abstract data types; at the other there are components which can function as stand-alone applications. For example, a spreadsheet tool is a component that has interfaces for adding and removing data from cells, for changing the number of rows and columns, for creating relations between cells and for changing data views. It is normal for clients to request extra functionality from successful components which may require the interfaces to be extended or modified.

Architectures are assemblies of components; they are subsystems that provide services. Reuse of architectures permits substantial savings over stand-alone components. Developers are keen to reuse architectures to leverage application development. Architectures can support concerns such as distribution, letting the application developers concentrate on the business end of their system. For example, an agent framework is an architecture. An agent framework allows for the creation of autonomous, heterogeneous objects that have the ability to 'reason' for themselves, negotiate with other agents or refuse to accept messages [130]. Conformance with existing technology may guide the developer to selecting one agent platform over another. Reusability is often in the requirements for architectures.

The reused concern or the unit of change during evolution is not determined by the technology used to implement it but by what the reuser or maintainer considers important. Reuse is simpler when the concern happens to coincide with a modular artifact, be it a function, a procedure, a component or an architecture. Programmers may wish to reuse all sorts of concerns including code associated with implementation abstractions and feature concerns, i.e. code addressing an aspect of end-user functionality. To facilitate reusability, all pertinent concerns require a modular representation.

Seamless evolution and reusability are achievable through investment in architecture but external pressures negatively impact architectures, slowly eroding any inherent flexibility.

2.2.3 The Effect of External Pressures on Reusability

It has been observed that reusability is affected by the *deadline effect.* Projects often have tight schedules that force the programmers to come up with quick solutions to problems. Any reusability which was inherent in programs initially is eroded little by little by a sequence of unanticipated extensions [104]. Many projects start out with a well-defined architecture which gradually gets eroded until the program becomes a big ball of mud. According to Foote and Yoder [41], the big ball of mud architecture predominates in practice. Programs that have such architecture are "haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle". The problem with big balls of mud is that they are of no use to anybody except the current users who become dependent on the services these systems offer. Big balls of mud are an oil slick in the ocean of reusable software – as systems grow larger they become more and more difficult to understand, raising the cost of reuse [17].

A project which starts out with a discernible architecture becomes a big ball of mud largely because of cost. The customer usually needs something by tomorrow. Often, the people who manage the development process simply do not regard architecture as a pressing concern. If programmers know that workmanship is invisible, and managers do not want to pay for it, a vicious circle is born.

In muddy code the important data is global, the rest is passed surreptitiously through various channels. Variable and function names are uninformative or even misleading. Control flow and programmer intent is hard to understand. The code is patched numerous times by different maintainers and there is no up to date documentation.

The prevalence of the big ball of mud approach to software development has lead Foote and Yoder to conclude that it is a method which works. It is a path of least resistance when one is confronted with the forces described above. In order to restore structure Foote and Yoder suggest that systems should be *refactored.* Refactoring improves the program's structure, improving its understandability and facilitating evolution without changing the program's functionality [93]. Refactoring enables *consolidation* – a process by which experience accrued as the system evolves gets absorbed into the system's structure. When code has declined beyond comprehension and repair, *reconstruction* or restarting from the ground up remains the sole viable approach.

We believe that realistic technology-based solutions to stopping programs turning into big balls of mud must take into account the time pressures placed on developers. For example, in a pressured environment, changes in requirements are hacked directly into code. Instead of applying changes to design artifacts and then applying the changes to code, code is changed and the designs slip into obsolescence. Design artifacts are discarded because of the effort associated with maintaining them when all that really counts is whether the code works or not. Making designs more useful to maintainers is the way to raise interest in design artifacts. We believe that code is less likely to turn into big balls of mud when up-to-date design documents are available for assessing the impact of modifications and extensions during maintenance.

For object-orientation, Clarke *et al* [25] believe that there are three reasons why developers do not use designs throughout the software lifecycle:

- Designs are often large and monolithic. Classes and interfaces are centralised notions and only one designer at a time can work on a given design unit. Centralisation causes early commitment to structure which may overconstrain the set of possible designs too early, consequently increasing the impact of change.
- Designs are too difficult to reuse because they bundle too many pieces together. Classes designed for a particular system are too specialised for general use. Potentially reusable classes include a lot more functionality than a reuser requires, decreasing their comprehensibility and reusability.
- Most importantly, there is structural misalignment between requirements and code with the design caught in the middle. In requirements, the units of abstraction and decomposition relate to capabilities, features and other concepts in the problem domain. Object-oriented code focuses on classes, interfaces and methods. This causes the problems of traceability of requirements in code. Design languages such as UML [18] produce designs that align well with code. Consequently, designs also align poorly with requirements. When the requirements change, the developer does not wish to incur the cost of making changes twice – once for design and once for code. So he changes only code.

The reuse lesson is that designs and code should modularise what is in the requirements as well as what is necessary to modularise in the implementation.

Decentralisation can drive down costs for the original developer: given the right technology one development team can be assigned to the implementation of each requirement despite the structural overlap in implementations. Modular development of each requirement may speed up project delivery making the technology that enables decentralisation attractive to the original developers and the reusers.

2.3 Challenges in Constructing Reusable Software

Component and framework development are associated with the construction of reusable software. In order to successfully market software as a reuse artifact, it is necessary to make software adaptable to a range of applications. This requires developers to anticipate changes and then provide flexibility through design. Blackbox and whitebox are two strategies for making software adaptable to evolution.

2.3.1 Frameworks and Component Based Development

Composition and generation technologies are two accepted ways of constructing reusable software [16]. Frameworks – a generation technology – are semicomplete applications that can be used to generate custom applications. They are specialised to a range of applications and designed to solve a narrow set of problems [36]. Component-based development (CBD) – a composition technology – involves building systems using prepackaged components. Standard component architectures such as CORBA [13], JavaBeans [48] and Microsoft COM [108] enable developers to market components to wide audiences.

Components, such as those designed for the JavaBeans model, can be customised but only in the way intended by the developer. The adaptation interface is limited to introspection – the ability to observe and modify a predefined range of properties. There is no conventional access to the internal design which makes it difficult to modify beans if the modification was originally not anticipated. Also, there exist concerns that we would like to reuse but which are not easily modularised by a component. For instance, a Tracer component that gathers statistics about data flow between other components is difficult to define. All components that may be traced are required to implement a certain interface and to support the notification of data flow within their implementation. Manual selection of data flow points is prone to error. Even more importantly, when components are developed independently by third parties, it is not reasonable to expect component developers to know about all other components with which they may be connected.

Frameworks can significantly increase software quality and reduce development effort [36]. One problem is finding the right framework to reuse. Companies attempting to use large-scale frameworks often fail to recognise and resolve challenges such as [35]:

- the learning curve the programmers must go through before they become proficient at using a particular framework,
- integration between frameworks that address parallel concerns,
- framework maintenance will require application code to be updated,
- reliance on framework developers to remove defects, and
- efficiency penalties over custom applications.

Many impediments to framework reuse are non-technical and are, in general, connected to problems a customer can experience when relying on external services. Overall, the benefits of frameworks significantly exceed the drawbacks but reuse remains a problem in new application areas where frameworks are unavailable.

2.3.2 Problems with Planning for Reuse

When constructing reusable software, the developers aim to to make their component as generally useful as possible in order to open it up to a wider market. The process requires the developer to anticipate variation and create hooks for future evolution. It also helps to avoid invasive code modifications with respect to planned extensions. For example, in framework construction, certain extensions are part of the requirements and, therefore, should be built-in. In order to create a successful framework one must foresee the uses to which the framework will be put.

The programming language used in software development determines the cost of providing extension points. For instance, abstract data types such as lists are conceptually generic with respect to the kinds of components that can be stored within. When building component libraries, ADTs can be made generic in any sufficiently general programming language. However, in order to easily construct generic ADTs the language must support either generalisation or inheritance. For example, C++ supports genericity with template classes [115]. Template classes allow families of related classes to be specified without a significant syntactic overhead. When the cost of providing a particular kind of adaptation is not significant within some language, programmers will take advantage of the available language features in order to make components more adaptable as part of good design. Programming language features such as genericity can lead to more reusable software without requiring investment.

The problem with planning for reuse beyond the original requirements is elegantly summed up by Fowler [42]:

> "One way to deal with changing requirements is to build flexibility into the design so that you can easily change it as the requirements change. However, this requires insight into what kind of changes you expect. A design can be planned to deal with areas of volatility, but while that will help for foreseen requirements changes, it won't help (and can hurt) for unforeseen changes. So you have to understand the requirements well enough to separate the volatile areas, and my observation is that this is very hard."

The advice of Extreme Programming [11] is that you do not build flexible components on purpose. Let the structures grow as they are needed. The reasons are economic – if work is done on features that may be needed tomorrow, time will be lost for features that need to be done for this iteration. Also, working on things for the future is outside the contract the programmer has with a customer. It should be up to the customer to decide what extra work should be done.

2.3.3 Black-box and White-box Reuse Strategies

The terms white-box reuse and black-box reuse are defined in relation to what the reuser believes to be the interface for adapting the artifact. With white-box reuse, programmers are free to modify code beyond the adaptation interface to suit their needs. In CBD, the adaptation interfaces of components are the points of interaction between the components and with the component model. In frameworks, the adaptation interface consists of the preplanned extension points. This approach gives a lot of freedom for component adaptation but is also fraught with difficulty because consistent modification requires complete familiarity with code. The other extreme is black-box reuse which disallows unanticipated modification of the retrieved component. The black-box strategy can make it more difficult to find suitable artifacts. Black-box components in CBD technologies allow a limited degree of adaptation which may be insufficient to customise the component to the needs of another project.

Common abstract data types are black-boxes because the reuser is interested in their functionality but not implementation. Larger-grained components are often black-boxes to reuser-clients but white-boxes to reusers who require access to parts of the internal design. For example, consider the development of user-interface (UI) software for mobile phones. There are two kinds of reusers. The UI company producing the software and the telecom company configuring the software. Changes to the underlying model are made by the UI company who see the software in white-box form. The telecom company may configure the software for phones with a different number of keys, displays in monochrome or colour, and introduce different menu options. The telecom company is a black-box reuser.

Confusingly, frameworks are classed as white-box reuse [35] despite the implementation of framework classes being hidden from the reuser. The reuser needs to access documentation explaining how to extend the framework to create an application but does not need to know the details of implementation of framework classes.

The information hiding aspect of black-boxes is appealing because it allows more complex systems to be built by using black-boxes as building blocks. The open-ended adaptability of white-boxes is appealing too because it gives considerable freedom to adapt components. The situation is analogous to the drivers of racing cars. Although it is possible to drive a car without understanding anything about the details of engine or gearbox design, or the principles of power and torque, an expert driver will use his knowledge of the way the machine works to harness its potential for winning. Furthermore, to drive a racing car well it is not necessary to know anything about, for example, the way the gearbox shifts cogs. Hence, at every stage, there is extra information which separates an expert user from a novice and additional details which are not relevant to perfoming the task well.

Returning to computing, Kiczales believes that Open Implementations can address the problem by providing multiple interfaces [64]. Open Implementations is a proposal for writing substrate systems – programs used by developers for creating and supporting the execution of client applications. Programming technology and operating systems are examples of substrate systems. Such programs have two kinds of interfaces: the meta-level interface is a side door into substrate systems that is used to tailor the base-level interface to meet the special needs of clients. The meta-level interface uses meta-object protocols [65] to provide three kinds of openings:

Introspection. Access to implementation state.

Invocation. Access to internal fuctionality.

Intercession. Changes to behaviour or implementation strategy to improve performance.

A power user can exploit the meta-level interface to modify the system in powerful yet structured ways. Writing substrate systems as Open Implementations is more expensive initially. The cost is recouped through reduced extension costs. When the substrate lacks functionality needed by its user, the user can use the meta-level interface to extend the substrate.

Clearly, only a small proportion of programming activity is concerned with writing substrates. Nevertheless, meta-object protocols demonstrate their flexibility when coping with changing requirements and the resulant changes to systems. In order to be more adaptable to unanticipated changes a component needs to provide facilities for changing from the inside.

2.3.4 Setting The Research Direction on Reuse

Construction of reusable software plays an important role in the software reuse spectrum but it is not the whole of the spectrum. A lot of software is not built for reuse but to address the functional requirements.

There are at least two ways that construction of reusable software can be motivated. The first is to develop marketable components. The incentive comes in the form of component trade. Building reusable components for marketing is an established practice in the software industry. One of the challenges concerns opening up component markets to improve availability and drive down prices [105]. The second is to seek improvements in software modularity. Due to the unanticipated nature of evolution and reuse, it is often not possible to predict what concerns the current project will share with other projects. However, the units of software decomposition will be more reusable if each module addresses one well-defined concern. As a starting point, the feature concerns identified in the requirements specification should be considered for modularisation.

This dissertation focuses on modularity for reuse. Object-orientation is today's dominant programming paradigm. In the rest of this Chapter, the way object-oriented programming enables software reuse is examined.

2.4 Reuse in Object-Oriented Programming

When object-oriented programming (OOP) was introduced, it was marketed as a programming paradigm that facilitates reusability (e.g. [40]). But as we have shown, reuse has many facets and two people who have an intuitive understanding of reuse may each have a different intuitive understanding. Presently, OOP is discussed in the way it is commonly perceived by many object-oriented programmers [83]: through the languages C++ [115] and Java [45]. For reasons of compatibility and for non-technical reasons programmers are often required to use these languages in projects.

2.4.1 Key Aspects of Object-Oriented Programming Languages

A number of factors combine to make object-oriented programming amenable to code reuse:

Abstraction. Data abstraction encourages the creation of modules which hide their implementation behind an abstract interface.

Inheritance. A new class can be derived by reusing code from an existing class.

Polymorphism. With polymorphism, an object of a derived class can be used in place of an object of the expected class.

Abstraction

Objects can be used to represent abstractions in the problem domain and in the solution domain, i.e. the domain of implementation. To understand what an object does it is not necessary to look inside the object; the behaviour is characterised by what is observed at the interface. The data abstraction properties of object-oriented languages are well suited for modelling abstract data types.

The class is the modular design unit in OOP. Its designer decides what is internal and external. Visibility modifiers `private` and `public` determine the services that are available to clients. The abstraction properties support the construction of black-boxes where the interface makes available those services that are required by clients in the object's sphere of application. It is not in the interests of the original designer to anticipate any additional services to which a reuser will require access.

The conceptual separation of type specification from implementations of the type is opaque in C++ and Java. A class defines both the type of objects and their implementation; although, Java does allow programmers to separate the type from the implementations with interface constructs. Multiple classes can implement an interface, defining variants, and a single class can implement multiple interfaces, in effect permitting an object to have many types or views. In order to be useful in reuse the developer must be consistent in separating all classes from interfaces. Good object-oriented practice suggests also that one should hide all field variables. Accessor methods should provide controlled access. These practices improve the separation of concerns by abstracting the client away from the implementation, allowing one to change the implementation without affecting the clients. This good advice is not always followed by programmers probably because it requires extra keystrokes or due to misplaced concerns for execution speed.

Solutions need not come in the form of a different programming language. Extensive labour-saving tool support exists for Java and C++ to discourage bad practice.

Inheritance

Inheritance is a relationship defined between classes. In object-oriented terminology, a subclass extends a superclass inheriting its non-private members, possibly overriding inherited methods and defining new members. Class members are field variables and methods. Reuse with inheritance comes in two forms: superclass reuse and client reuse.

Superclass reuse is the most common form associated with inheritance. When two classes have similar parts, these can be extracted and placed into a common superclass which both classes inherit [61]. Alternatively, one can start from an existing class or classes (in C++), inherit and make the required modifications. The first form of superclass reuse is more likely to occur during initial development as code gets factored into a common superclass. The second form is common during *a posteriori* reuse.

Client reuse is the reuse of code associated with clients. When inheritance is used to create a subtype, all the code that uses the supertype instances can seamlessly migrate to using subtype instances. The amount of code reused this way is often more significant than with superclass reuse [15].

Multiple inheritance allows a subclass directly to inherit members from multiple superclasses. Languages that support multiple inheritance of implementation, such as C++, also have a conflict resolution mechanism for disambiguating the order of inherited members when similarly named members are inherited from different superclasses. Multiple inheritance simplifies reuse because it allows one to create classes that contain combinations of properties inherited from distinct superclasses. On the other hand, it can make the behaviour of objects more difficult to understand because it requires comprehension of more branches in the hierarchy.

Polymorphism

Subtype polymorphism, genericity and multiple dispatch are different kinds of polymorphism. C++ and Java have subtype polymorphism in common. Objects interact by sending each other messages. The object whose code is executed in response to a message is known as the receiver. In these languages, subtype polymorphism requires a dynamic check. When a method is called, the code which gets executed is determined at run-time based on the type of the receiver.

This polymorphic behaviour is useful when a group of objects have the same general form but differ in specific details. The client can treat related objects in the same way but the behaviour that is invoked depends on the actual type of the receiver. Polymorphism reduces initial development costs by treating a set of objects of different types in a generic way; clients can refer to the subset of the interface all objects share. To achieve the same effect in procedural languages one can define maintenance intensive `if-then-else` structures.

Genericity is also known as parametric polymorphism. C++ supports genericity through template classes. Template classes are instantiated by giving concrete classes as special type parameters. Genericity is extremely useful where an upfront requirement for genericity is identified, e.g. for creating reusable containers such as `List[X]` which can be parameterised by different types `X` of list elements.

Multiple dispatch appears in CLOS [63] – the object-oriented extension to the Lisp language. It allows code selection based on the dynamic type of the receiver and parameters. Multiple dispatch can reduce the cost of class reuse in some designs. Consider the example of two or more modems.

```
Modem m1 = new XModem(); // notice that static type is different
Modem m2 = new XModem(); // to dynamic type.
Modem m3 = new YModem();
m1.connect(m2); // invokes proprietory
m1.connect(m3); // invokes standard

abstract class Modem {
  void connect(Modem m) { .. } // the standard protocol
}
class XModem extends Modem {
  void connect(XModem m) { .. } // proprietory protocol
}
class YModem extends Modem {
  void connect(YModem m) { .. } // proprietory protocol
}
```

Figure 2-1: Example demonstrating multiple dispatch.

Suppose that to enable faster data transfer, a modem connects to another modem of the same make using a proprietory protocol. Two modems of different makes communicate using the standard protocol. Multiple dispatch allows us to define a simple interface that is common to all modems. When a new modem model is produced, we create a new subclass of the abstract modem class which defines the proprietory protocol. The client code can treat all modems the same but connection invokes the propretory protocol when the dynamic type of the receiver and parameter are the same. Figure 2-1 shows the Modem example using Java-like pseudocode. Unlike Java, the parameter type is dynamic. In the absence of multiple dispatch, the effect of `connect(..)` will be to call the standard protocol each time.

2.4.2 The Role of Inheritance in Reuse

Object-oriented programming associates reuse with classes. A class is a versatile reuse unit because it spans all levels of abstraction from basic abstract data types to large and complex components. Inheritance is the main reuse operator introduced by object-oriented programming. Reusers employ inheritance to derive a new class from existing classes. Maintainers also use inheritance when requirements change. Reluctance to modify existing classes for fear of breaking them leads maintainers to using inheritance instead. Inheritance acts as version control: the subclass is a newer version of the superclass. However, inheritance is also used for conceptual modelling, to express supertype-subtype relationships, and to introduce variant implementations of types. Using inheritance for reuse without establishing a clear conceptual relationship between the superclass and the subclass leads to the the ball of mud architecture described earlier. The elimination of multiple inheritance of implementation from the Java language can be seen as a way of trying to combat bad practice.

LaLonde and Pugh [72] discuss three distinct interpretations of inheritance. Subclassing refers to inheritance of implementation. Subtyping permits an instance of a subclass to be used in the place of the superclass. Inheritance between classes is modelled using the ‘is-a’ test. The problem is that mainstream programming languages have few ways of expressing the different relationships. Inheritance problems are a consequence of misunderstanding the precise nature of the relationship.

Porter [101] proposed separating the subtype hierarchy from inheritance of implementation as a way of improving the understandability of object-oriented programs. In the subtype hierarchy, method signatures defined in the supertype can be redefined in the subtype, but no method implementation overriding takes place. The subtype hierarchy achieves full substitutability. In the implementation hierarchy code sharing occurs. An implementation class implements zero or more types and can inherit implementations from multiple classes.

All reuse examples so far have concentrated on reuse of a single class in creating a new class. However, much can be gained by subclassing multiple abstractions. The absence of multiple inheritance is a hurdle to class reuse. The problem is one of using multiple inheritance in a structured way in order to keep programs easy to understand and facilitate reusability in the future.

Gardner [44] has distinguished between different fundamental forms of inheritance. She proposes five structured inheritance relationship (SIRs) for object-oriented programming. The relationships are conceptually orthogonal, all SIRs are necessary to model the conceptual relationships that occur in software systems, and SIRs are sufficient for modelling uses of inheritance described in object-oriented literature:

Variant. Describes a relationship where the subclass satisfies the type specification in the superclass. For example, a linked list or array implementation of a stack type.

View. Describes a use of multiple inheritance by which an instance of the superclass can be viewed as a number of different types. In this way it is possible to develop different interfaces to the same object which are appropriate to different kinds of client. For example, view of a person as a student, parent, employee, patient, etc.

Evolution. Allows the implementation of abstraction to be built up over time due to changing requirements. The evolved abstraction is not expected to work in the original system, although the evolved abstraction may be conformant with the old system. For example, in moving from monochrome to colour displays we may inherit class `Point` to create `ColourPoint`.

Construction. A form of inheritance for reuse which uses an existing class in building another class. For example, a number of graphical application windows may inherit the same menu abstraction.

Specialisation. Creates a hierarchy of types where a subtype is substitutable wherever the supertype is expected. For example, `Child` and `Adult` subtypes of a `Customer`.

Of the five relationships, **specialisation** is associated with the behavioural notion of subtyping [74] and **variant** with type conformant implementations. All but **specialisation** can be used to establish some kind of code reuse relationship. Multiple inheritance can be used with most SIRs to create new abstractions. The atomic natures of each SIR ensures that the relationship with the inherited abstractions is conceptually sound and explicit in the design. For example, **variant** and **view** SIRs can be used together, e.g. an object of the `2DPoint` class can be viewed as a `IPair`. Implementation class `IPair` is a variant of type `TPair`. In another example, **construction** can be used multiple times, e.g. to add the behaviour of `Scrollbar` and `TitleBar` abstractions to a `Window` abstraction. Gardner demonstrates that multiple inheritance is conceptually valid and that it has a role in modelling and reuse.

In object-oriented programming languages, visibility modifiers `public` and `private` delineate the interface from the implementation and protect secure data from direct access. Class members marked `protected` are accessible within subclassess but not to external clients. In Java, the modifier final signifies that the subclasses should not redefine that member. final also facilitates compiler optimisation. These modifiers fundamentally affect the reusability of a class by specifying valid extension points. Reuse is a problem where the required extension point is not visible to the subclass due to the presence of certain modifiers.

Object-oriented programming permits black-box reuse of classes using delegation and inheritance. With delegation, instances of existing classes are used to build new abstractions. Programmers may prefer to use inheritance over delegation due to the advantages associated with client reuse. When the subclass is not a behavioural subtype of the reused abstraction the reuser must ensure that the new abstraction will be conformant with all existing clients. With delegation, the clients must always co-evolve whether the new abstraction is conformant or not. Inheritance is often preferred to delegation when the derived abstraction has a similar interface and shares aspects of implementation with the reused class. Using inheritance to do programming-by-difference the reuser only specifies the way in which the new class differs from its superclass(es).

2.4.3 Reuse Artifacts Not Associated with a Class

Reuse problems also occur when the concern is not captured by a single class. Either a single class addresses multiple concerns or there are many classes that jointly contribute to a concern. Respectively, these are problems of tangling and scattering. Discussion of such reuse artifacts forms part of Chapter 3 (page 22) on Advanced Separation of Concerns.

In summary, reuse in object-oriented programming languages requires a degree of preplanning. Concerns that were not modularised by a class originally are difficult to reuse. Non-invasive evolution of classes is not always possible because the variance points are hidden within and are not part of the object's externally specified behaviour.

2.5 Conclusion

This Chapter discussed the challenges associated with software evolution and reuse. Better separation of concerns is a way to improve the reusability of software in projects where reusability is not a primary concern. Modularisation in design of concerns derived from the problem domain has the potential to benefit both the original developer and the reuser. The original developer benefits from parallel modular development of concerns. Reusers who share goals with those addressed by existing projects can reuse artifacts from those systems. Due to traceability of requirements in design, the reuser is better able to identify and extract the code associated with the reuse artifact.

Object-oriented programming emphasises the reuse of classes. Preplanned reuse is supported through a combination of inheritance and delegation, but reuse of software where reusability was not a concern *a priori* often requires access to extension points that are hidden inside the class. In moving beyond mainstream object-oriented languages, the next Chapter looks at extensions to mainstream languages and other programming technologies which

- modularise concerns that are not easily represented by classes, and
- facilitate reuse of software in ways not anticipated by its original developers.

Chapter 3

Advanced Separation of Concerns

Modularity is key to making reuse possible. The previous Chapter reviewed object-oriented programming as perceived by developers of mainstream languages. Today, thanks to object-oriented programming, the reuse of abstractions represented by classes is a reality. However, problems remain when a class cannot be extended due to the absence of suitable extension points. Reuse of concerns that are not modularised by a single class requires a degree of advance planning. For instance, many concerns in the problem domain and certain concerns in the solution domains, such as persistence, synchronisation and distribution are not modular in object-oriented programs. Modularity of these concerns is achieved through advanced separation of concerns.

The aim of this Chapter is to review the state of art and understand the challenges involved in advanced separation of concerns. Our view on modularity coincides with that taken by Tarr et al [122]. These researchers propose Multi-Dimensional Separation of Concerns (MDSOC) as a new programming paradigm for improving the modularity of concerns that developers identify as important. Section 3.1 presents MDSOC and describes the motivational factors for changing the way software is developed.

Many concerns in the problem domain are not captured by a single class but associated with collaborating suites of classes. These collaborations are modular in design languages such as UML in the form of sequence and collaboration diagrams. MDSOC technologies for making collaboration modular in code are reviewed in Section 3.2.

Aspect-Oriented Programming (AOP) addresses certain goals of MDSOC by modularising persistence, synchronisation and distribution concerns, as well as many other concerns that cut across application functionality. Mechanisms for modularisation of solution domain concerns that have proven difficult to modularise in object-oriented programs are reviewed in Section 3.3.

Subject-Oriented Programming (SOP) is an instance of MDSOC that can modularise collaborations and many solution domain concerns. Section 3.4 justifies the selection of SOP as the vehicle for supporting reuse.

3.1 Multi-Dimensional Separation of Concerns

MDSOC is a new paradigm for modelling and implementing software artifacts [122]. It proposes the separation of overlapping concerns along multiple dimensions of composition and decomposition. A concern is any matter of interest in a software system. Dimensions group concerns; they are a

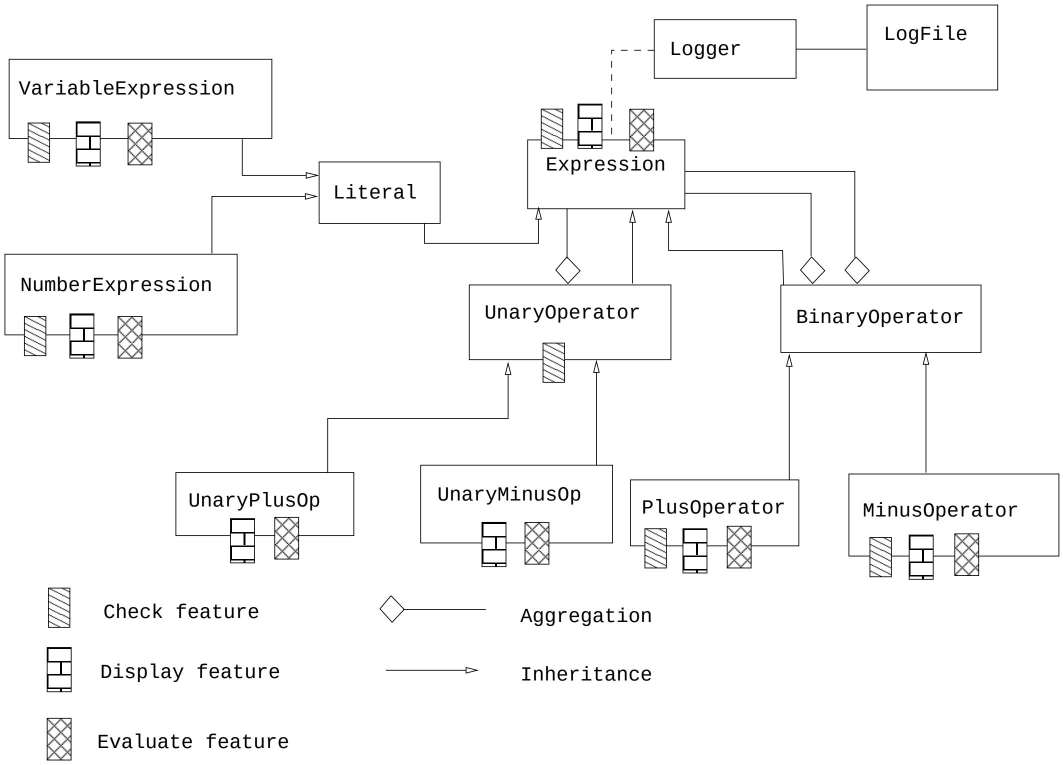

Figure 3-1: Scattering and tangling in SEE

perspective on to the system. MDSOC improves reusability by modularising all pertinent concerns at the same time – concerns from the problem domain, those emerging as part of the design, during initial software development, and in maintenance.

The problems of degrading software comprehensibility, common maintenance tasks leading to high-impact invasive changes and limited reusability are caused, in large part, by our inability to keep separate all concerns of importance in software systems. All formalisms support the decomposition of problems into subproblems to some extent, but provide a restricted set of decomposition and composition mechanisms. These mechanisms support a single dominant dimension of separation, ignoring all other possible dimensions. Tarr et al have termed this phenomenon the 'tyranny of the dominant decomposition'. In order to break the tyranny, the MDSOC proposal requires technology to support simultaneous separation of multiple concerns in multiple dimensions.

What follows is a summary of the problems identified by Tarr et al that motivate the introduction of the MDSOC model; a review of the main concepts of the MDSOC model that help to explain the way systems should be modelled; and a a critique MDSOC based on our reuse position.

3.1.1 Motivation for MDSOC

To illustrate the problems in software development Tarr et al [122] develop the Software Engineering Environment application. The application supports the specification of algebraic expressions with a collection of tools that manipulate the shared abstract syntax tree representation. The initial tool set includes an evaluation capability to determine the result of evaluating an expression, a display capability, and a check capability which determines both semantic and syntactic correctness of expressions.

The UML design for the application (shown in Figure 3-1) contains a class for each kind of `Expression` in the abstract syntax tree. Each class defines the `eval()`, `display()` and `check()` operations which realise the tools in the standard object-oriented fashion. The code has a similar structure to the design. This example illustrates an important issue in software development: the system is decomposed differently when viewed from the perspectives of requirements and design/code. Requirements are decomposed by tool or feature, and design/code is decomposed by class. This phenomenon leads to the problems of scattering and tangling:

Scattering. A single requirement affects multiple design and code units.

Tangling. Multiple requirements are implemented within a single module.

Each of `eval()`, `display()` and `check()` is scattered across the class hierarchy with many classes contributing to the realisation of each concern. The implementation of each class tangles the feature concerns.

Having used the application, clients request functionality for optionally making expressions persistent. The clients also require different kinds of style checking functionality, and it should be possible to mix and match syntactic, semantic and different kinds of style checker.

These seemingly simple additions (from the perspective of the client) significantly impact the design and code; scattering and tangling pose a problem to the evolution of the Software Engineering Environment. Persistence requires modifications to the accessor methods of each class to retrieve persistent objects and save modified state to the database. It is possible to use inheritance to add persistence functionality but all clients must be evolved to create instances of the new abstractions. Moreover, the persistence requirement is affected by the selection of checkers; the style checkers must include their persistent state together with expressions. Mix-and-match of checkers requires infrastructure support that was not necessary originally. The Visitor design pattern introduces the flexibility at the cost of higher coupling between the AST classes and the visitor classes [43]. The Visitor pattern is useful when many distinct operations need to be performed on objects in an object structure. The introduction of the Visitor pattern may impact future extensions if new kinds of `Expression` need to be defined. The changes are invasive because units of change do not match the units of abstraction within the design/code. Subclassing and design patterns require particular changes to be anticipated, but anticipating future change is not in the requirements of many projects.

Different artifacts associated with software creation have varying levels of abstraction. They are decomposed and structured differently because of emphasis on different kinds of concerns. Scattering and tangling of requirements occur because the concerns of importance in the requirements do not map cleanly to design and code units. Hence, when changes to requirements happen, propagation takes a great deal of effort.

The cause of the problem is the 'tyranny of the dominant decomposition'. Today's formalisms support a small set of decompositions and usually have a single 'dominant' one at a time. The dominant decomposition satisfies some important needs but usually at the expense of others. For example, in the original object-oriented solution to the Software Engineering Environment, decomposition based on data encapsulation concerns reduces the traceability of feature concerns which are equally important. Solving the problem involves breaking the tyranny by modularising simultaneously all concerns identified as important. The dimensions of concerns identified as important to the Software Engineering Environment include:

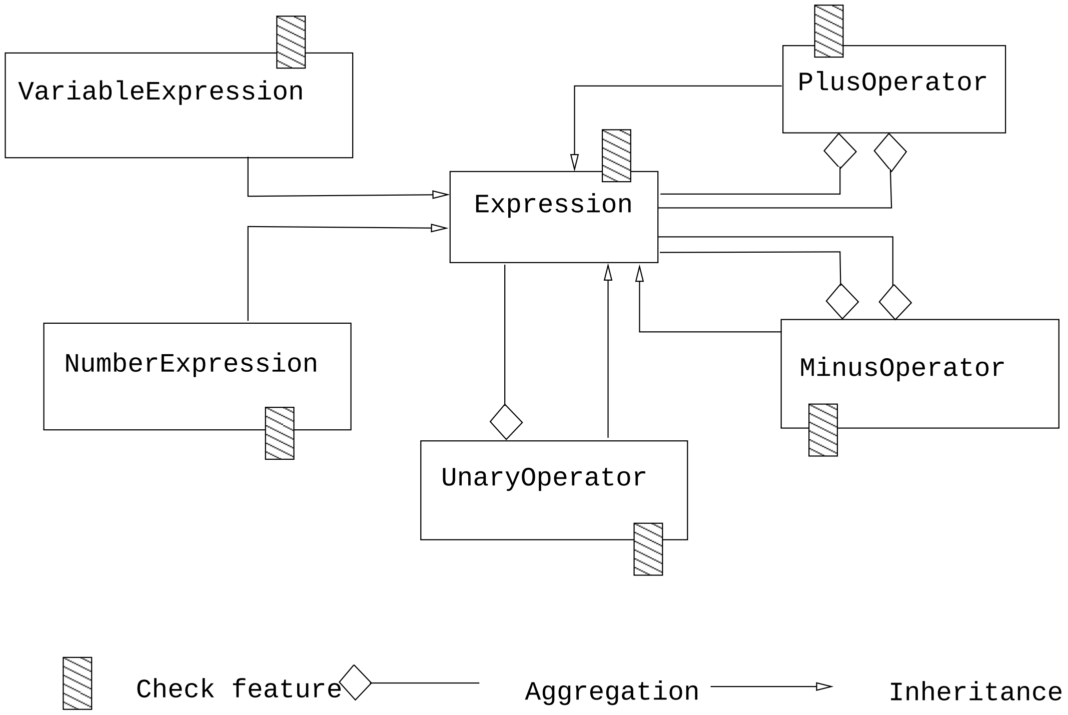

Figure 3-2: The Check Feature as a hyperslice

Feature. These include display, evaluate, persistence, syntactic check, style check, semantic check.

Unit of change. Additions made due to user requests.

Customisation. Additions or changes needed to specialise the component to a particular purpose.

Object. The classes involved in the system.

There are many other dimensions of concerns which may be applicable, such as to separate 'optional' from 'required' pieces, or to customise the application to different kinds of user, etc. The dimensions are rarely orthogonal, they overlap and can affect one another. A flexible solution to modularisation must allow the pertinent dimensions to apply at the same time and handle overlap and interaction between them.

3.1.2 The MDSOC Model

The MDSOC model is intended to capture all concerns and all dimensions of concerns in a software intensive project. It introduces *hyperslices* as an additional flexible means of software decomposition. Hyperslices are intended to modularise concerns in dimensions other than the dominant one. Hyperslices are implemented using a set of convensional *modules* and *units*, written in any formalism. For instance, Figure 3-2 shows hyperslices applied to UML class diagrams. The modules are classes and a hyperslice is a collection of classes. Attributes and operations are the units in the hyperslice. A collection of units corresponds to a module, e.g. a class or an interface. To understand the hyperslice it should not be necessary to look inside its units. The hyperslice contains exactly those modules and units that are required to address the concern.

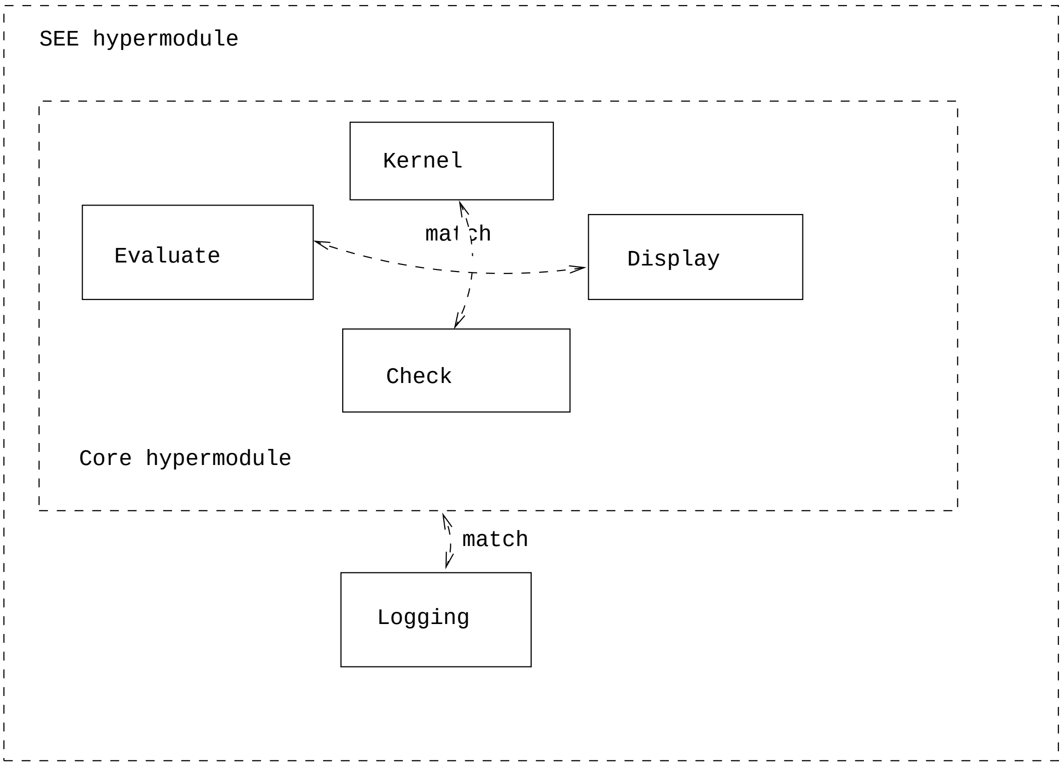

Figure 3-3: The hypermodules created by composition

Hyperslices are expected to overlap such that the modules and units in one occur, possibly in a different form, in other hyperslices. A system is written as a collection of hyperslices, reflecting the concerns in the system that have been identified as important along as many dimensions as necessary. Systems are created by composing hyperslices on the basis of *composition rules.* A set of input hyperslices and a composition rule together are called a *hypermodule.* A new hyperslice is created by applying a composition rule to input hyperslices, so a hypermodule can be used whenever a hyperslice is expected. The complete runnable system artifact, e.g. a component or a subsystem represented as a class diagram, can be modelled as a hypermodule.

Composition is established by a process known as *matching* – identifying elements which describe the same concept in different hyperslices. The differences between corresponding elements are resolved before *integration* of elements produces a unified whole. The specification of composition is part of the design process and cannot be automated. Figure 3-3 shows the matching process and generation of a hypermodule in SEE.

Application of the MDSOC model to the Software Engineering Environment example leads to the separation of major concerns of importance identified during requirements analysis. Hyperslices can modularise the 'kernel' functionality which contains the state and accessor methods, and each of the display, evaluation and checking features. Checking is itself a hypermodule made up of hyperslices specifying the different kinds of checks present in the system. With MDSOC instantiated on UML class diagrams, each hyperslice contains the design for one concern. Hyperslices modularise the features, and within each hyperslice the object concerns are separated in the class diagram. If these modules can be kept separate in code then separation of feature concerns can persist over the lifecycle.

Modularisation of features also solves the problem of scattering and tangling: all elements pertaining to a concern are specified within its hyperslice. The AST concept – the most general concern in the application – is modularised by the `kernel` hyperslice. It has been separated from its context and can be reused in other applications. Other concerns, such as `checking`, although more specific, can still be reused in other contexts if the situation arises.

The final application is created on the basis of the composition rule. By including or excluding hyperslices it is possible to mix and match features. When requirements change, such as the addition of persistence described earlier, a new concern is introduced along two dimensions. Persistence is a feature and also represents a unit of change. It can be modelled as a separate hyperslice and composed with other hyperslices.

The MDSOC model is not a panacea for bad design. It is possible to over or underseparate concerns. Overseparation leads to a large number of hypersices with complex inter-slice relationships that may actually reduce comprehension and increase complexity.

3.1.3 MDSOC and The Position on Reuse

Technology implementing the MDSOC model can facilitate reuse by improving the modularity of concerns that are presently tangled and scattered in designs in code. Therefore, this technology is going to be attractive to practitioners who wish to create well structured systems. However, at least two aspects of MDSOC may detract practitioners:

- The MDSOC model introduces a lot of duplication. If each hyperslice defines the functionality it requires, the same behaviours may be defined multiple times. By contrast, modern programming trends have tended to minimise duplication. Pragmatic MDSOC technology must endeavour to avoid duplication during hyperslice design and coding stages; although, as in multiple inheritance, duplication during reuse should be expected.

- The MDSOC model improves modularity but does not improve encapsulation. A class is both a modular artifact and a unit of encapsulation; however, a hypermodule does not provide any additional encapsulation. Composing hyperslices to form a hypermodule reduces duplication by integrating matching modules and units, but a hypermodule does not hide any more implementation details than the hyperslices from which it is created. Recall that a hyperslice consists of modules and units. Only the units of a hyperslice are modular artifacts in the traditional sense as defined in Section 2.2.1 on page 9.

The absence of additional means of information hiding can impact hypermodule reusability. Understanding the impact of adding further hyperslices is no easier with hypermodules than with the hyperslices from which a hypermodule is composed.

MDSOC permits decomposition of systems along many dimensions. For example, requirements, design and code hyperslices pertaining to the development of the ‘expression’ concept in the ‘kernel’ concern of the Software Engineering Environment can be a hypermodule. This decomposition helps to trace the development of the ‘kernel’ through the lifecycle. This and other decompositions may be useful during reuse or maintenance but what decompositions have value to the original developer? In order to change the way practitioners develop software, it is important to motivate new kinds of decompositions. We believe that there are two dimensions of concerns that have value to the original developer:

- Concerns in the feature dimension. Modularisation of a system along feature concerns may enable concurrent development of features by teams. Feature modularisation facilitates requirement traceability and may reduce the duration of development cycles as teams work in parallel on realising each feature.

- Cross-cutting concerns in the solution domain. Cross-cutting concerns that are scattered and tangled in the dominant dimension have become popularly known as *aspects*. In addition to the usual benefits associated with improved modularity, modularisation of aspects often reduces the amount of code overall [66].

In any large project there are concerns which arise in the solution domain. These concerns are defined by the solution techniques. Aksit et al [4] emphasise the importance of solution domain over problem domain concerns. They argue that the problem domain concerns do not include the necessary concerns for implementing the software system because many important concerns are transparent to the user. To illustrate the point an example of sharing components in a network is presented. The subconcerns of consistency management and performance optimisation emerge as part of the solution domain. These are not explicitly included in the requirements specification but should be treated as aspects of the system.

An alternative view is presented by Jacobson [60] who argues that systems should be sliced use case by use case. Use cases have been widely adopted for requirements specification. They are a core part of the Unified Process – a way of constructing software with UML. In order for a feature-based decomposition to be attractive in projects where feature modularity is not an *a priori* requirement, the total cost of development, including hyperslice composition, should be comparable to the cost of conventional software development. In other words, the benefits to the original developer should help outweigh the initial investment.

In order to make it easier to concentrate on the problem domain concerns, it has been proposed to make feature concerns oblivious to aspects, i.e. system level concerns [38]. For instance, it should be possible to specify the features of the application without making provisions for security. The security policy is applied separately. The features neither need to declare the secure artifacts nor make special provision for security to be applied. Obliviousness makes it possible to make functional changes to the application without concern for particular aspects; the aspects can easily adapt to changes in feature code. Obliviousness is not an intrinsic property of MDSOC but programs created using MDSOC technology can be designed to support obliviousness with respect to certain concerns. Most Java programmers are oblivious to details of memory deallocation thanks to garbage collection. Also, in component development with JavaBeans aspects of bean deployment are separated from bean functionality. The design of beans is oblivious to the deployment strategy. Essential to obliviousness in MDSOC are powerful means of connecting hyperslices.

In the following two Sections we review the technology for modularisation of collaborations and aspects. There is a degree of overlap: a number of proposals are well suited to modularisation of both collaborations and aspects. Subject-Oriented Programming [49] and Object Teams [54] are two such models. The presentation of these proposals is split over the two Sections.

3.2 Technology for Modularisation of Collaborations

Many functional concerns are associated with collections of classes rather than with a single class. At runtime, objects of the types derived from these classes collaborate on a task defined by the concern. A way to achieve the reusability of object collaborations is with a straightforward mapping from design level concepts into distinct implementation elements [111].

In the present Section, the recent work related to the modularisation of collaborations is presented. The presentation starts with attempts to extend object-oriented programming with support for collaborations, and leads to alternative software development models that also enable related classes to be modularised.

3.2.1 Collaborations in Object-Oriented Programming

Contracts is the name given to a technique for formally specifying behavioural compositions [53, 58]. A contract defines a set of communicating *participants* and their *contractual obligations.* Participants are mutually recursive: they refer to each other and send each other messages. Contractual obligations consist of:

Type obligations. The participant must support certain variables and an external interface.

Causal obligations. The participant must perform an ordered sequence of actions and make certain conditions true in response to messages sent to the external interface. Causal obligations capture the behavioural dependencies between objects.

Each contract also defines invariants that participants cooperate to maintain and actions which should be taken to resatisfy the invariant. In order to initiate a contract, the state of all participants must be set up in line with the invariants.

This formalism has constructs for the *refinement* and *inclusion* of behaviour defined in existing contracts. Refinement allows for the specialisation of contractual obligations and invariants. The obligations of multiple participants are specialised in concert. Inclusion allows contracts to be composed from simpler contracts. A subcontract relates a subset of the participants of the contracts which include it.

Contracts are specified in a high-level language that allows abstract description of behaviour and realised using *conformance declarations.* In a conformance declaration, classes map to participant specifications, i.e. the program must be shown to satisfy the specification. A class conforms when its methods and instance variables satisfy the typing and causal obligations required by the participant definition. The implementation of a participant can be distributed among a number of classes related by inheritance, and a class can implement the contractual obligations of a number of participants. For example, code common to a number of contracts may be extracted into an abstract superclass.

Contracts make explicit those interactions which in object-oriented programming are hidden inside constructors or implicit in sequences of method calls. They are intended to be implemented directly in an object-oriented language. Although modularity of collaborations is achieved at the design level, the separation of concerns is not propagated into code, thereby losing the traceability between design and code.

Template-Based Implementations of Collaborations

An implementation of collaborations based on template classes in C++ is proposed by VanHilst and Notkin [125, 126]. Each object in a collaboration is said to play *roles* in collaborations with other objects [12]. Template classes can be used to implement roles. Role participants are passed as parameters to the templates. A template parameterises all participants to which it must refer, including self. For example, the father's role in a two parent household might be defined in part as:

```
template <class ChildType, class MotherType, class SuperType>
class FatherRole : public SuperType {
  ChildType *child;
  MotherType *mother;
  ...
};
```

In this collaboration, the `ChildType` and `MotherType` parameters are the collaborators with this `FatherRole`. Template parameters indicate that, as yet, they are of unknown type. The `SuperType` role is used in every definition, since every role is part of some unknown class. Templates are instantiated by specifying classes for each template parameter. For example, suppose that `ChildClass` and `MotherClass` play the child and mother roles in the above collaboration, `HusbandClass` plays the self role, Then an instantiation of the `FatherRole` appears as:

```
class FatherClass : public FatherRole<ChildClass, MotherClass, HusbandClass> {};
```

Roles from different collaborations can be composed in this model. It is possible to compose roles from different but related collaborations and from repeated uses of the same collaboration. New roles are created by passing template classes as parameters to other templates. Smaragdakis and Batory later propose an improvement to template-based implementations of collaborations entitled Mixin Layers [112]. Mixin Layers address certain scalability issues by defining roles as nested or inner classes of an outer class that denotes the entire collaboration. A C++ implementation of a two parent household is given below:

```
template <class CollabSuper>
class TwoParentFamily : public CollabSuper {
public:
  class MotherRole : public CollabSuper::MotherRole { ... };
  class FatherRole : public CollabSuper::FatherRole { ... };
  class ChildRole  : public CollabSuper::ChildRole  { ... };
};
```

The template-based approach makes it possible to implement many collaborations modularly in code, addressing the traceability problem associated with contracts.

The roles of a collaboration can be reused in the creation of new roles using inheritance, but in many collaboration specialisation scenarios a set of role classes must evolve at the same time. The set of evolved classes participating in the collaboration must be used together. Although code is shared with the super-roles, the roles are not type substitutable for their superclasses. Neither subtype polymorphism nor multiple dispatch can provide the static safety guarantees which ensure that only objects of the same collaboration participate. To address this problem, Ernst has proposed family polymorphism [34].

Family Polymorphism

The classes of objects participating in a collaboration, i.e. roles, form a family. Family polymorphism allows to statically declare and manage the relations between several classes polymorphically, in such a way that a given set of classes is known to constitute a family but it is not known statically exactly what classes they are.

In a system containing more than one variant of a class family, in order to avoid mixing families inappropriately it is necessary to maintain consistency in the usage of family members. Family polymorphism is a mechanism that helps to resolve this problem, statically ensuring that the roles of any set of families are never mixed. Besides supporting collaboration refinement, family polymorphism also confines role objects to their family, thereby encapsulating role objects in collaborations. The Object Teams approach to making collaborations modular uses the encapsulation properties of family polymorphism to create reusable collaborations.

3.2.2 Object Teams

The Object Teams [54] proposal introduces a new kind of module, a *team*, for modularising object collaborations. A team is an instantiable aggregation of confined objects called *roles*. It contains a collection of classes that define the roles and a set of operations and variables defined at team-level. Teams support three kinds of inheritance-style relationship:

- *Explicit inheritance* between teams is used to create specialised teams as well as to reuse code specified in the superteam. For example, imagine an application that allows a passenger to collect bonuses with every flight. A `Bonus` team, as an abstract collaboration between the scheme `Subscriber` and a `BonusItem`, may be extended to create a `FlightBonus` team. The `FlightBonus` refines `Bonus` by redefining the function `calculateCredit()` to return a rounded value. Explicit inheritance links `FlightBonus` to `Bonus`.

- Inheritance between roles is called *implicit inheritance*. By redefining a role class in a subteam we implicitly gain access to the members of that role in the superteam. The `calculateCredit()` operation is overridden due to implicit inheritance between `BonusItem` in team `FlightBonus` and the same role class in team `Bonus`.

- Team composition is achieved with *object-based inheritance* (which is in fact delegation). This establishes a relationship between a role in a team and some base class, i.e. the class begins to play a role. For example, suppose we want to apply the `FlightBonus` team to a particular application involving air miles. Class `Segment` defines the attribute which specifies the segment length. In order to `calculateCredit()`, we require access to the air miles travelled. This is formalised in code by declaring:

```
class BonusItem playedBy Segment
```

Object Teams supports the encapsulation of team representation. Usually, a role instance is confined to its enclosing team, however, a role can be exposed but only if the team reference is declared `final`, i.e. immutable. The modifier ensures that no other team is assigned to this variable while the role is exposed. The exposed roles cannot be passed to a different team from the one in which they originate.

3.2.3 GenVoca

GenVoca is a component model for constructing hierarchical software systems [10]. It provides direct language support for a design model that supports component composition. The familiar notions of abstraction, encapsulation and parameterisation are extended to include the new kind of GenVoca component which has the following properties:

Abstraction. Support for standardised interfaces in the form of the *realm* construct. Standardisation leads to functionally similar, interchangeable and interoperable components.

Encapsulation. Large-scale construction is supported through the *component* construct. Components can encapsulate collaborations of multiple classes.

Parameterisation. Customisation and composition of components is supported with parameterisation. In particular, realm parameters can be passed to components to create layered or hierarchical compositions of components.

A set of function and class declarations defines the realm. To implement a realm it is necessary to specify implementations for all classes and functions defined in the realm. It is also possible to introduce totally new classes and members in the implementation.

Instead of using inheritance, GenVoca employs parameterisation to create connections between components. Any component which is instantiated with a realm parameter implements a new *layer*. A layer is the term used to describe a component built by reusing another component. The new layer can extend the realm parameter's interface to create new classes. The result is a new component which implements new functionality on top of the component it extends.

GenVoca is implemented in the P++ language which is an extension to C++. The language hides a template-based implementation, similar to the one described in the previous Section. Similar to Mixin Layers and Object Teams, GenVoca adds a concept of a higher-level module.

GenVoca emphasises construction of reusable components with a strong emphasis on valid combinations of features implemented by components. More recently Batory et al propose to scale step-wise refinement to hyperslice level [8, 9]. Step-wise refinement asserts that complex programs can be derived by progressively adding features. By considering the combinations of orthogonal, i.e. non-overlapping features, Batory et al show that valid combinations can be specified relatively concisely. In a model that can be decomposed along n dimensions of concerns with k features along each dimension, there could be as many as $O(k^n)$ feature combinations to consider. However, based on the results achieved by Batory and his colleagues, specifications of length $O(kn)$ can be produced. The shorter specifications enable the programmer faster to convince himself of the correctness of his program for all combinations of its features.

3.2.4 Subject-Oriented Programming

Subject-Oriented Programming (SOP) [49] is a programming paradigm that can modularise feature concerns. In SOP, the *subject* is the artifact playing the role of the hyperslice. A subject models its domain from its own particular perspective and is implemented using classes, instance variables and operations in a standard, object-oriented way. It is a subprogram that addresses a concern from the problem domain or the solution domain. Subjects facilitate a clean separation of concerns by defining only those elements which contribute to addressing the concern.

Harrison and Ossher [49] observed that OOP is well suited for building independent applications but less well suited for building integrated suites or families of applications. The traditional view of OOP is of a model for representing abstractions in the real world. The complexities of object implementations are hidden behind a compact, abstract interface. However, real world abstractions have a multitude of requirements and constraints. For instance, a car abstraction can be viewed from the perspectives of the driver, salesman or mechanic. Each domain has its own vital properties of the car and has particular demands on behaviour which can affect those properties. The driver may classify cars based on size, economy, reliability and is concerned primarily with the behaviour of driving. The salesman may classify cars based on model designation; the choice of cars to buy and sell depends on the demand for a particular model, the wholesale and retail prices. The mechanic classifies cars based on parts and tool availability. European models have attachments measured in metric units while American cars use imperial measures. In the traditional object-oriented view of the world, when requirements arise, all these views must be accommodated by the interface. As software evolves, more requirements get introduced, bloating the interface further and leading to the scattering and tangling problems discussed in the motivation for MDSOC in Section 3.1 on page 22.

The SOP solution facilitates independent development of cooperating applications as subjects. Cooperation is achieved by sharing objects and jointly contributing to the execution of operations. In the above example, the domains of driving, car sales and mechanic responsibilities can be treated as separate subjects which can be implemented independently and subsequently composed to satisfy application goals. The aims of SOP are:

- Treating each subject as an application: there should be no explicit dependence in code on other applications.
- The composed applications may cooperate loosely or closely.
- It should be possible to add new applications that serve to extend existing applications in unanticipated ways.
- Each application should maintain the advantages of inheritance, polymorphism and encapsulation.

SOP conforms well to the MDSOC model described earlier. Subjects can implement features and concerns emerging in the solution domain. Composition of subjects takes place after all points of interaction between subjects have been agreed. Subjects can be implemented independently or reused if a suitable subject exists already.

The similarity between SOP and MDSOC is not surprising given that MDSOC generalises many of the ideas first presented as part of SOP. The programming language Hyper/J [121] implements all SOP concepts. Our description of SOP semantics is based on the specification of Hyper/J.

Subject Design

Within a single subject, design is a purely object-oriented activity. In the Hyper/J language, each subject is programmed as a Java package. Composition is performed on compiled subjects, i.e. on the Java Virtual Machine bytecodes. Therefore, each subject must compile correctly before it can be composed and the classes of each subject must be valid Java classes. For example, it is common for a problem decomposed by feature to share a 'kernel' concern which defines operations used in the implementation of other features. The other subjects can either define a method with an empty implementation or declare the shared method `abstract`. Neither solution is ideal. In the first, a non-`void` method must return a value. As no implementation is defined, the subject author must specify an arbitrary value to return. In the second, if a method is `abstract` then the Java class which declares it must also be abstract. In Java, an abstract class has no direct instances, making this solution unsuitable in those cases where the subject needs to instantiate that class.

As an object-oriented artifact, a subject has a functional interface defined by one or more of its classes. As a subject-oriented artifact, it also has a compositional interface. The behaviour of a subject can be invoked using either or both interfaces. When the subject implements a feature it often has a functional interface that is invoked by external clients through method dispatch. Subjects may also implement concerns which are not invoked as a result of a method call but in conjunction with control flow related events in other subjects, e.g. when an operation is called within another subject. For instance, consider a `Caching` subject that can be applied to a subject implementing an arbitrary data structure. Saved values are stored in the cache before being stored in the data structure, and retrieved values are first looked up in the cache. The caching behaviour is invoked when values are *stored* and *retrieved* from a data structure. The `Caching` subject is activated via the compositional interface. The subjects to which caching applies are affected at their compositional interface.

The compositional interface is wider than the traditional functional interface. The points in code where elements are stored and retrieved from a data structure need not be part of the functional interface. The compositional interface is essential to cleanly separate concerns, and it helps to add new concerns without modifying existing code.

Subject Composition

Composition forms a single program which is a synthesis of the input subjects. It takes place statically, before the composed program is run. Subject composition is defined in terms of two concepts: *correspondence* and *integration*. Correspondence identifies the places of interaction between subjects and integration determines the action taken on corresponding elements.

The primary point of interaction is the class or the interface. Classes can correspond only to classes and interfaces only to interfaces. The views of the same kind of object as expressed by the class or the interface in corresponding subjects must be composed. Subjects can agree that a set of classes or interfaces represent the same type of object from different perspectives without having anything else in common. Class or interface composition makes it possible to view an object via different types in each subject.

At runtime, most subject interactions need to share more than just object identity; behaviour and state may also need to be shared. Subjects are static entities and do not have state as such. State is associated with objects of executing subject-oriented programs. Statically, that is in terms of program text, state is captured by instance variables. Behaviours affecting the state take the form of operations which are associated with classes directly or inherited. Subject interactions that involve state or behaviour are specified by defining correspondences between instance variables and operations of corresponding classes.

State can be shared between subjects when corresponding classes define the same instance variable. For example, both the car driver and salesman share the notion of car key. By establishing correspondence between the instance variables representing the car key, subjects can share key objects at runtime. Behaviour is shared when subjects define the same activity in response to an action. For instance, starting the car is a behaviour which is the same for both its driver and the salesman. The realisation of shared behaviour can be delegated to one subject and activated by all who need it. We have also observed other interactions:

- Request by one subject to invoke behaviour in another. For example, a driver who has lost his key may request the mechanic to start the car using other means. Here the behaviour is not the same for both, rather, just one subject has the behaviour which the other may require.
- Performance of an activity in which another subject participates, e.g. a prospective purchaser may ask a mechanic to help him evaluate the car's condition. The purchaser's decision is based both on his own assessment and that of the mechanic.
- An event which may be of interest to another subject, e.g. if the car is stolen, its driver will want to notify the police.

Integration is the process of establishing an interaction between corresponding elements. In order to synthesise a single program from the inputs, SOP unifies the corresponding elements based on integration rules. During execution, methods are invoked from multiple input subjects. Which methods are invoked depends on the subject where the call originates, the correspondences, and the integration strategy. Many kinds of integration rules can be defined but there are two general-purpose rules which are used in many compositions. The **merge** rule describes a union of corresponding elements and the **override** rule describes the selection, at composition time, of one of the corresponding elements.

The definitions of both these integration rules are overloaded to specify the unification of multiple kinds of corresponding elements. The **merge** rule is defined as follows:

- For instance variables, its effect is to create a single variable in the output.
- For operations, the method bodies are set to execute in arbitrary order (but not in parallel). When the operation is called in any input subject, all method bodies are executed. If methods return values, all the return values are packaged into an array and a composer-specified summariser method is used to determine the return value for the **merge**d operations.
- For classes, this integration rule creates a single class in the output. All member integrations can only be performed in the context of a corresponding class. For example, if two subjects both declare `anEngine` to be an instance variable of class `Car`, then in order to merge the views of engines, it is necessary to specify the correspondence between `Car` classes.

The **override** rule also applies to classes and their members. The compositional effect of **override** on instance variables is the same as **merge**. For operations, the overriding method replaces all overridden methods such that when any one of the corresponding operations is called, only the overriding method executes. The overridden methods do not contribute to the behaviour of the output subject and cannot be invoked. On classes, the **override** rule has a quantifying effect: each element of the overriding class replaces the corresponding elements of the overridden classes. The members of the overridden classes without corresponding overriding elements are unchanged.

The integration rules presented above and other, custom integration rules (that can be defined by a power user who is familiar with the SOP rule framework) are the operators in the SOP *composition* *language.* The language gives the composer fine-grained control over the interaction, making it possible to express many compositions. However, when subjects implement feature concerns that have been designed in concert, the compositions should be concise. More verbose specifications may be required to compose subjects which have been developed separately.

Composition is specified in terms of a top-level rule that applies to all elements, followed by a sequence of lower-level rules describing exceptions and additional directives. The top-level rules are[1]:

compose. Specifies a sequence of subjects to compose and the name of the output subject, e.g. `compose S1, S2 into S;`.

mergeByName. Establishes correspondences between all identically named elements and applies the **merge** integration rule to each correspondence.

overrideByName. Establishes correspondence between identically named elements and uses the **override** integration rule. The first subject in the **compose** clause is the overriding subject (the source of the overriding elements).

The simplest composition specification contains one **compose** directive and one of the **ByName** rules. At the lower level, correspondence between elements that have different names can be established using the **equate** directive. This takes n elements of the same kind from n different subjects and specifies that these elements correspond. Exceptional integrations are specified in terms of **merge**, **override** and other basic integration rules.

When composing feature concerns, it is common to use the **mergeByName** strategy at the top-level. The features represent corresponding views which must be integrated to form the complete program. At the lower level, when two or more subjects share a method implementation, only one subject needs to implement it. The **override** rule can be used to select the implemented method. This integration rule can also be used to select just one implementation from a set of equivalent implementations of a method, e.g. one setter method implementation from the set of equivalent setters.

Additional directives which can be specified after the top-level rule include **bracket** relationships. Brackets are useful for specifying aspectual interactions: when one subject augments or modifies the behaviour of another at key points in its control flow. These relationships are discussed in the next Section together with technology for supporting aspect-oriented programming.

On Composition Validity

So far we have described key principles of composition but not the way an SOP language checks composition correctness.

In the Hyper/J language, subjects are pieces of ordinary Java code. When instance variables and operations are composed, the return types and parameters in the same positions must have the same types. Type correspondence is required during **merge** integration because any input subject may attempt to access or modify the shared element. Type equivalence is also conceptually meaningful during merging as it suggests that the views of the object interface are mutually compatible. Hyper/J requires type equivalence for **override** also. The need for type equivalence here is less clear as

[1] Note that we diverge from Hyper/J syntax in order to simplify the presentation, but composition semantics are unaltered.

override makes it possible to apply changes one subject at a time, rather than one class at a time as is the case with inheritance in a language like Java. It is possible to use **override** to evolve an application to using a new family of types. Returning to the earlier example, in some modern vehicles the engine is started not with a key but by entering a code on a keypad. A new subject can be developed which replaces the `Key` class and changes all existing clients of the `Key` class. Nevertheless, in Hyper/J type equivalence in corresponding elements is required even for **override**.

Type equivalence is not as restrictive as it may sound; corresponding classes cause elements of these types to be composition compatible. Hence, different subjects can have different names for the same concept, e.g. `Warranty` and `Guarantee` describe the same kind of artifact. As another example, consider the merging of `Sink` and `Source` classes (as variants of some kind of buffer). One subject creates a `Sink` object; the reference becomes visible in another subject as a `Source` object through a shared instance variable or a merged operation. The **merge** of these classes is meaningful only if all objects of these types can be viewed from both perspectives.

The choice of Java constrains what can be done to check composition correctness. It is well known that types are only a small part of what makes a program correct. The emphasis on formal specification of collaborations in contracts and on valid combinations of GenVoca realms is aimed at ensuring that the interactions are not only conceptually relevant but functionally correct. Family polymorphism [34] is concerned with consistent evolution of a family of classes in a way that preserves type subtitutability and maximises class reuse. Comparatively, in SOP little attention is given to the important topic of interaction validity. In the following Chapter on interaction problems in SOP (starting on page 44), we analyse the interactions that are difficult to detect during reuse or before independent subject development commences.

3.2.5 Conclusion

The technologies presented in this Section enable the modularisation of collaborations. In object-oriented programming, contracts enable the high-level specification of mutually recursive objects, collaboration refinement and composition. Programming techniques based on templates and inner classes can be used to compose collaborations. Family polymorphism supports the evolution of sets of classes while maximising class and client reuse.

Direct programming language support for collaborations can be found in Object Teams, GenVoca and Subject-Oriented Programming. For programmers of a mainstream object-oriented language transition to SOP is probably the easiest. SOP lets the developer implement subjects using the familiar object-oriented techniques. Only the subject composer needs to know about the composition language. However, the SOP composition rules presented to now have affected classes and some decompositions require object-level granularity. GenVoca also operates on classes rather than objects but allows new components combining a number of features to be synthesised dynamically. By contrast, subject-oriented composition is specified during a separate phase of software development. Object Teams has object-level granularity. Teams can be activated and deactivated dynamically, and multiple instances of a team can be present at the same time. Consequently, with Object Teams it is possible to separate more concerns more cleanly.

The previous Chapter discussed the often unanticipated nature of reuse. To facilitate reuse it should be possible to reuse a component in a way not anticipated by its original developers. The compositional interfaces in SOP enable subjects to be connected to other subjects in unanticipated ways. The layer reusers in GenVoca create new layers by extending existing layers. The set of valid layer extension points is restricted to the predefined interface. The team modules of Object Teams are reused at their functional interface but Object Teams also support additional forms of interaction detailed in the following Section on Aspect-Oriented Programming.

A priori creation of reusable software in SOP is supported by inheritance and delegation in the language used to implement the subjects. GenVoca supports the creation of component families. The client can create new components by mixing and matching features created earlier as part of a family. Object Teams leverages family polymorphism for reuse of collaborating suites of classes. GenVoca supports polymorphism using the realm construct but a realm is the interface of a single component rather than a component family.

GenVoca and Object Teams provide stronger support for creation of reusable components for use by third parties. GenVoca focuses on valid permutations of modules and Object Teams focuses on type substitutable component families. By contrast, Subject-Oriented Programming has an extensible set of composition rules that make subjects better suited for reuse in ways which were not initially anticipated. SOP has comparatively poor support for checking interaction correctness. Unlike teams, subjects can modularise collaborations but do not encapsulate collaboration state. There is no way of determining interaction correctness until all subjects in the composition are known. In conclusion, we believe that in order to improve the reusability of collaborations without *a priori* investment in reusability, Subject-Oriented Programming is the better candidate technology.

3.3 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) is an approach to modularising cross-cutting concerns – concerns that are hard to modularise using today's mainstream programming technology. Initially, the term 'AOP' was associated with a particular approach to addressing cross-cutting concerns by Kiczales et al [66], but due to the popularity of the programming language AspectJ, AOP has become synonymous with other programming technology including Composition Filters [3], Adaptive Programming [73] and Subject-Oriented Programming. See [90] for an overview of approaches.

AOP (as described in [66]) is a generalisation of the ideas behind domain-specific languages RIDL and COOL for distribution and concurrency [78, 76]. AspectJ introduces language extensions to Java that allow distribution, concurrency and other concerns to be modularised. A domain-specific language allows only a few concerns to be separated and a set of domain-specific languages may not be mutually compatible. AOP addresses some of the goals of MDSOC: it facilitates the modularisation of many solution domain concerns that are difficult to modularise within a mainsteam programming language.

The concepts of *join point* and *aspect* unite all AOP approaches. Join points are places in the program where modules interact. Join points can be determined in different ways. In SOP, language constructs like instance variables, operations and classes are join points, so the places of subject correspondence and join points are synonymous. In AspectJ, join points are certain places in the program's control flow. Thus, join points are dynamic concepts. Aspects are modules containing the code which addresses some concern.

Join point interaction can be symmetric or asymmetric [50]. In an asymmetric model, the aspects are 'woven' into the 'base' modules at the join points. The selection of join points comes from the base program and aspects are written with respect to some base. AspectJ is an example of an asymmetric model. In a symmetric model, every module is treated as an aspect. A base is not distinguished linguistically although it may be distinguished logically within the domain of application. The join points come from each aspect and the interaction is usually specified separately. The SOP model is symmetric.

In order to separate some concerns it is necessary to have fine control over the join points. Many concerns affect only a subset of instances of a class. For example, AOP technology can be used to modularise the reusable parts of design patterns [47]. Consider the Observer pattern [43]: subscribers register with publishers[2] to receive notifications about state changes. The reusable parts of the aspect include the `Subscriber` and `Publisher` interfaces, and the protocol for enabling `Subscriber` registration and event notification.

Suppose that in an application, the `Rectangle` dimensions are observed by `Shape`s whose proportions are linked to that of the `Rectangle`, and the `Rectangle`'s `colour` property is observed by a different but possibly overlapping collection of `Shape`s which set their colour in relation to the colours of adjacent `Shape`s. Each `Shape` is possibly both a `Publisher` and an `Subscriber` and a `Shape` needs to be a `Publisher` to two sets of `Subscriber`s. Composition rules such as **merge** in SOP affect classes but cannot distinguish between instances and it is not possible to reuse the Observer pattern in the required setting. In fact all subject-oriented composition rules presented to now relate classes rather than objects.

This Section reviews the technology for modularisation of concerns that cross-cut other functionality. The review includes today's most popular AOP technology, AspectJ; extensions to SOP which enable more concerns to be separated; and the two most recent proposals Caesar [88] and Object Teams [54].

3.3.1 AspectJ

AspectJ is a forward-compatible extension to the Java language: valid Java programs are also valid AspectJ programs. AspectJ introduces a new kind of module known as an *aspect*. Like an ordinary Java class, an aspect contains members that define its state and behaviour. Instead of the usual functional interface, an aspect has a compositional interface that is based on join points. To select the join points of interest the aspect body defines *pointcuts* – specifications of join points of interest to an aspect. The behaviour associated with an aspect is set to execute *before*, *after* or *around* the join points. The last of these executes the aspect code instead of the code at the join point, possibly calling the code at the join point using the `proceed(..)` statement.

Pointcuts are specified in terms of *designators*. These describe events in the control flow such as when some instance variable is read or some event is thrown. Although AspectJ has a huge selection of designators, the most commonly used ones are concerned with method invocation and execution. In the 'Hello World!' of AspectJ programs, an aspect is used to modularise the `Tracing` concern. The `Tracing` concern, shown in Figure 3-4, requires a message to be printed immediately before and immediately after any method executes.

Note that asterisks are used as wildcards to match the execution of any operation on any class. In a traditional object-oriented program, tracing requires either that every method body is modified to include a call to the tracer module or for the program to be run within a debugging suite. The first solution does not scale while the second one is fine by itself but causes complications when collaboration with another application is required, e.g. a review of the recorded trace to see that

[2] The usual terms 'subject' and 'observer' have been replaced by 'publisher' and 'subscriber' to avoid confusion with SOP terminology.

```
aspect TraceAllClasses {
    pointcut myMethod(): execution(* *(..));

    before (): myMethod() {
        Trace.traceEntry("" + thisJoinPointStaticPart.getSignature());
    }
    after(): myMethod() {
        Trace.traceExit("" + thisJoinPointStaticPart.getSignature());
    }
}

public class Trace {
    public static void traceEntry(String str) {...}
    public static void traceExit(String str) {...}
}
```

Figure 3-4: AspectJ program implementing the `Tracing` concern.

execution has passed certain key points.

AspectJ has a very powerful join point language that allows programmers to modularise code pertaining to many non-functional concerns, separating the non-functional concerns from the program's main functionality. Concerns that should be separated [75] include synchronisation, location control (organisation issues), real-time constraints, failure recovery, debugging, persistence and transaction management. The aspect is linked to the base program through pointcuts, but the base program creates no explicit links to the aspect. Furthermore, good aspect-oriented design suggests using inter-aspect inheritance to separate the aspect functionality from the pointcuts that declare the way the aspect interfaces the base.

AspectJ is appealing to the original developer because modularisation of scattered code reduces the size of the whole program and improves its understandability. The declarative style for describing join points makes AspectJ more attractive to programmers than meta-object protocols from which it evolved.

3.3.2 Bracket Relationships in SOP

Bracket relationships (brackets for short) are an advanced compositional mechanism in the subject-oriented programming language Hyper/J. We describe them here because their introduction was inspired by the dynamic join points of AspectJ. Bracket relationships allow the methods of one subject to wrap the method call and execute sites in other subjects. The methods which are set to execute before or after another method are called *wrappers* and every bracketed method or method call point is known as the *wrappee*. Similar to **call** and **execution** designators of AspectJ, the bracket relationships of Hyper/J can use pattern matching to specify the wrappee points.

Figure 3-5 shows subject `Tracing` which contains the code for this concern and a fragment of the composition specification which enables the tracing of all method calls in the program. The meta-parameter `$signature` references the `String` represention of the operation signature at the join point. The syntax of the example is simplified from true Hyper/J syntax but consistent with SOP examples that follow.

Bracket relationships are implemented with the correspondence and integration rules of SOP. Brackets actually set up correspondences between classes containing the wrappers and the classes

```
subject Tracing {
  public class Trace {
    public static void traceEntry(String str) {...}
    public static void traceExit(String str) {...}
  }
}

bracket ‘‘*.*’’ with before Tracing.Trace.traceEntry($signature)
                     after  Tracing.Trace.traceExit($signature);
```

Figure 3-5: Hyper/J-style program implementing the `Tracing` concern.

containing the wrappees. **merge** semantics are used to integrate the corresponding elements; the instance variables and operations are integrated, then calls to wrappers are inserted at the relevant points.

Bracket relationships are applied after the top-level composition strategy, e.g. **mergeByName**. The composition specification may contain **equate** directives which create correspondences between differently named elements which should be composed, and the wrappers execute around all equated wrappees whenever at least one of them matches the pattern. Because brackets also setup correspondences between the classes of the wrappers and the wrappees, it is an error for the wrappers to have correspondences derived from the top-level composition rule. However, a wrapper class does have a functional interface of its own and it is possible to create instances of wrapper classes directly.

Hyper/J does not have the **around** construct of AspectJ which helps to separate concerns in some cases. The **around** construct permits the wrappers dynamically to determine if the wrappee should be executed. Although, the SOP framework provides an opportunity to introduce this and other rules.

3.3.3 Caesar

The Caesar language [88] builds on the strengths of the AspectJ approach to modularisation of cross-cutting concerns by improving the separation between the definition of the aspectual modules and the definition of the deployment of those modules with the base and other aspects.

The Observer pattern, seen earlier, benefits from instance level application. But even then, its definition in AspectJ is not as reusable as may be required in practice. The AspectJ definition does separate the pattern protocol from its applications by using inter-aspect inheritance but the solution is not sufficiently general. Specifically:

1. As discussed earlier, a component may play the role of a publisher more than once, i.e. it may be observed independently for changes of two different properties. For this reason, the AspectJ implementation uses `PublisherSubscriberProtocol` aspect singleton to combine both the publisher and subscriber roles, using a hashtable to map between instances of publishers and their subscribers. The tangling of roles in a single aspect makes the relationship between the core pattern functionality and different specialisations of it unclear. For instance, if later `SpecialPublisher` specialises the publisher role conceptually, we have to subclass the whole protocol to create an extension of just one role. To avoid such problems it is necessary to keep each aspect role separate.

2. Applying the aspect to a problem requires mapping each role directly on to a class, i.e. af-

fected classes have to implement the Publisher or Subscriber interfaces. However, for some applications of the pattern, the role may be implicit in a collaboration. For example, suppose that Screen needs to observe changes in points and lines. Points are represented by Node objects and lines implicitly by pairs of adjacent Node instances. Thus, for lines, there is no single abstraction that can inherit the Publisher interface.

3. In certain cases the aspect bindings should be reusable. Such is the case with compositions between a class-based representation of an abstract syntax tree and a general tree representation. Abstract syntax trees are used in many applications. Many different tree implementations may be used with such a binding, e.g. ones to display trees or to perform algorithms. These kinds of bindings require a special kind of polymorphism to make them truly reusable.

4. In AspectJ, the aspect is woven statically at compile time. It is not possible dynamically to select a different aspect implementation at run-time, to activate or deactivate the aspect. For example, with respect to the Observer pattern, we must select in advance whether to have synchronous or asynchronous notification of changes, i.e. there is no aspectual polymorphism.

Caesar addresses these shortcomings through *aspect collaboration interfaces* (ACIs). These are bidirectional interfaces between two sets of modules, called *aspect implementations* and *aspect bindings*. The first part is called the *provided* interface and the second, the *expected* interface. The aspect implementations realise the part of the ACI's interface that is concerned with the aspectual functionality, e.g. the Observer pattern protocol. The provided part of the ACI binds the aspect roles to the target application abstractions.

Addressing the deficiencies in the AspectJ solution, we see that for point 1 the aspects can be represented in terms of their own class structure. There is no need for global aspect state because each aspect deployment hides the state within instances of aspect roles. For point 2, Caesar uses aspect bindings to map aspects to domain abstractions. Bindings are fully-fledged Java classes with additional features that the programmer can use to specify complex aspect interactions with base code. For point 3, aspect binding reusability is achieved in three directions. One can define functionality that is polymorphic with respect to (a) aspect implementations by being written to a certain aspect binding type, (b) aspect bindings by being written to a certain aspect implementation type, or (c) both of them, by being written to an ACI. Finally, for point 4, inheritance between aspects is combined with polymorphism to allow specialised aspects to be used in the future.

Caesar is an instantiation of MDSOC as it divides problems into multiple dimensions of concerns, allowing one to view and use the system from different perspectives. Unlike Caesar, in SOP there is no explicit notion of composition interface and reuse is associated with subject code but not usually the composition specifications. Reuse of composition specifications is not considered because it is expected that for most compositions, the relationships can be concisely defined using a top-level composition strategy. This is certainly the case for subjects designed in concert where most correspondences are inferred by name equivalence. Where subjects are built for reuse, the composition specification is also a potential reuse artifact. To reuse subjects successfully, one requires documentation which includes a description of the way the subject may interact. The examples can be used as informal templates for creating a composition specification. Finally, Caesar is an object based approach whereas SOP is class based.

3.3.4 Object Teams

Object Teams also supports advanced interaction between teams in the style of AspectJ. Recall from Section 3.2.2 on page 31 that a team class encapsulates a number of role classes. Using object-based inheritance each role is bound to a class that begins to play that role. Object Teams requires all roles to be bound.

Team interaction is handled by so-called *callin*s and *callout*s. Method delegation uses callouts. These allow a role instance to delegate the call to an instance of a base class. A callin mimicks AspectJ's **before**, **after** and **around** advice. At certain points specified in the team, the base object calls into the the role, passing state and meta information to the role.

3.4 A Case for Subject-Oriented Programming

This Chapter has discussed technology for advanced separation of concerns. The goals of reuse can be better addressed by modularising concerns in the feature dimension and by separating more of the cross-cutting concerns that emerge as part of the solution from the rest of the functionality.

We propose SOP as the reuse vehicle because it best fits our reuse position. Subject-Oriented Programming keeps the initial development costs low for programmers familiar with today's mainstream programming languages. It has a powerful join point language that enables the separation of many kinds of concerns. SOP supports reuse by allowing extensions and compositions at points that may not have been anticipated by the original developer.

SOP also has disadvantages compared to other approaches:

- A subject *is* a modular artifact. However, when non-public join points are used for creating extensions and specifying subject interactions, the subject no longer encapsulates the state of collaborations it implements.

- The benefits of inheritance and polymorphism are restricted to subject implementations. Consequently, construction of software for reuse and expression of conceptual relationships where inheritance between subjects is appropriate must rely more on composition rules. For instance, present composition rules cannot express the relationship between an 'abstract' subject and its 'concrete' variants.

We believe that the second problem may be tackled with new composition rules but the first point is more challenging. Subjects break the encapsulation of objects yet fail to encapsulate state common to the collaboration as happens in Object Teams. Broken object encapsulation is an inhibitor to modular development and reuse of subjects. In the next Chapter we detail our experiences of programming with Hyper/J and discuss interaction problems which we believe result from the invasive nature of subject interaction.

Chapter 4

Interaction Problems in Subject-Oriented Programming

Subject-Oriented Programming is a technology for separation of concerns. It enables subject reuse using powerful composition rules. Subject reuse and evolution is not defined on pre-declared composition interfaces in the same way as traditional components. Instead, interaction between subjects takes place at the points defined by subject structure. The absence of a more abstract interface makes it difficult to predict all consequences of interaction in advance. We call the unwanted interactions between subjects *interaction problems*. The present Chapter explores interaction problems in SOP and suggests possible solution spaces.

Section 4.1 defines interaction problems. We compare interaction problems in subject-oriented programming to feature interaction problems in telecom applications. Examples of interaction problems are drawn from existing work and through our personal experience of programming with Hyper/J. Sections 4.2, 4.3 and 4.4 present examples of interaction problems. In each case, we compare the subject-oriented program to a functionally-equivalent OO program, and present existing work intended to address the problem or a description of a possible solution. Section 4.5 concludes by proposing to develop a subject-oriented Alias Protection System.

4.1 Introduction to Interaction Problems

Decomposition of programs by feature rarely leads to orthogonality. Features often read and write a common set of properties which in an object-oriented design would be encapsulated in an object. SOP supports decomposition by feature by defining corresponding classes and class members, and by allowing one subject to interface another at method call sites and potentially other internal join points. Broken object encapsulation leads to a wide interface, making modular development of subjects more challenging compared to conventional programming where encapsulation is preserved.

For example, in order to reuse a subject in a composition, it is not enough to know *what* the component does. One must look beneath the interface at *how* the component is implemented. So, if two subjects both have views of the `Employee` class which share the `salary` field, both must agree on the type of this field, its valid range and usage policy. This kind of interaction requires one to have knowledge of the way other subjects implement and use data.

We believe that broken encapsulation combined with a high number of interacting subjects leads to interaction problems. In order to be able to tackle interaction problem, it is first necessary to find a suitable definition for interaction problems. A definition should help us to understand the problem space and also to evaluate the solution space. There has been little published on interaction problems in SOP specifically, so the net is cast further afield to other domains where interaction occurs between modules that describe overlapping perspectives.

Viewpoint-Oriented Systems Engineering (VOSE) [39] is a framework for supporting the design of heterogeneous systems. A viewpoint is a locally managed entity which encapsulates partial knowledge about the system or domain, specified in a particular, suitable representation scheme; it carries partial knowledge of the process of design. VOSE uses *inter-viewpoint checks* to verify the consistency of a specification with those maintained by other viewpoints. Conflict resolution is part of the specification process. The aim of inter-viewpoint checks is to eliminate inconsistencies and produce a conflict-free design. Inconsistencies are equally undesirable in subject-oriented composition because they make subjects uncomposable. However, SOP does permit variation to a degree. Each subject can define classes from its own perspective and each subject can define its own class hierarchy. Other inconsistencies are undesirable, requiring invasive modifications to subjects.

Feature interaction problem is a term coined in the telecommunication domain to describe interference between services. A bad feature interaction is one that causes the overall system behaviour to be undesirable. Interference occurs when the behaviour of one feature is affected by the behaviour of another feature or another instance of the same feature [68]. It has been recognised that research into methods for detection and resolution of feature interactions in telecom systems is also of signficance outside the telecom domain. According to Plath [100], feature interaction problems can often be traced back to the fact that two or more features manipulate the same entities in the system, and in doing so, violate some underlying assumptions about these entities that the other features rely on. Our intuitive understanding of interaction problems is comparable to feature interaction problems as defined by Plath.

In her work on subject-oriented design, Clarke has investigated the composition of object-oriented design models [24]. In her experience there exist

> "...additional properties arising for the output of composition. These are not defined in any input subject but arise as a result of composition itself."

Clarke's definition implies that the composition is meaningful overall but particular interactions were either not foreseen or unexpected. Assuming that the additional properties are unwanted, some action must be taken to correct the interaction. Changes can be made to the composition specification, the input subjects or by using a patch subject. The last two approaches are least desirable because they raise the cost of subject reuse. The following definition reflects our view that whether the interactions are unforeseen, unspecified or unexpected, the overall effect is undesirable and requires some corrective action on behalf of either the composer or a subject designer.

Definition: (Interaction Problem) A subject interaction occurs when the behaviour of one subject influences the behaviour of another. Interaction problems are unwanted subject interactions.

Interaction problems are important to us because they affect modular subject development and reuse of subjects. Having established a definition, we now look at the current understanding of interaction problems in related areas and evaluate the approaches to tackling them.

4.1.1 Feature Interaction Problems in Telecom Applications

The best understanding of interaction problems that we have comes from the telecom domain. Telecom applications are built around features which surround the basic service. The aim is to develop features modularly and resolve any interaction issues between features in a reasonable amount of time without modifying the feature specifications. Features subtly interfere because they manipulate the same common service variables. In the telecom domain it is important to be able to rapidly develop and deploy new features without disrupting the functionality of existing features.

Kimbler points out [67] that the trick is not in finding resolutions but in developing mechanisms to detect and resolve interactions that are efficient and that apply in the majority of cases. The approaches to tackling feature interactions can be broadly grouped into design methods, architectural approaches and runtime techniques [52]:

- In the design approach, specifications of separately developed features are composed. The composition is searched for interactions. Undesirable interactions are resolved by modifying feature specifications. However, a feature cannot be redesigned independently to eliminate the interaction and features may not be open for re-design. So instead, most design-time resolutions specify how groups of features behave together, using various techniques to define valid permutations of features. The main problem with the design approach is scalability. Most telecom systems have hundreds of features with the number of interactions growing exponentially as yet more features are added.

- The architectural solutions involve co-ordinating the features' access to shared resources. For instance, a pipe-and-filter architecture serialises features' reactions to each event. The problem with this approach is that it tends to over-constrain access. The analysis of feature interaction is still required to ensure that features do not miss key events occurring further down the pipeline. Architectural solutions are too general to prevent interactions that violate all feature constraints.

- Resolutions that are not resolved statically must be resolved dynamically. Resolution can be deferred until the unwanted interaction occurs and some action needs to be taken.

The design approaches are also applicable to subject-oriented interaction problems. For instance, Van Der Straeten and Brichau [114] propose to use declarative metalevel representation of the feature's implementation to detect interaction and interference (i.e. interaction problems). Feature interaction is detected using logic rules, e.g. a logic rule can detect access to the same instance variable by two different features. Rules for detecting interference are expressed as constraints or invariants on the implementation.

Analysis of subject-oriented composition is also constrained by scalability. The complexity of interaction analysis grows significantly as the number of interacting subjects increases. The design approach is a useful way of understanding interactions and resolving problems but unless the model is imbedded in code, the reuser must construct a new model for each new feature every time its subject is introduced to an existing set. A design approach consistent with our reuse position must have value not just for the composer but also for the subject developer.

An architectural solution is useful in the telecom and other domains where there exists some well defined set of common resources. But Subject-Oriented Programming can be used in areas where there is either no common set of resources or the architecture itself is subject to evolution.

SOP only supports design-time composition. Runtime resolution of conflicts introduces overheads that may be acceptable during testing but not in the deployed system. We believe that resolvable conflicts are addressed best using composition rules.

4.1.2 Composition Anomalies

Tekinerdogan et al [123] have conducted an evaluation of the different kinds of composition schemes (e.g. inheritance, delegation and join point based composition) in order better to understand anomalies occuring during concern composition. They distinguish three categories of problems:

- Composition is not possible for logical reasons. One tries to compose concerns which are inherently uncomposable.
- Composition cannot be realised because the adopted composition scheme does not support it, although composition is possible from the logical perspective.
- Composition is realisable with the adopted composition scheme, but requires additional workarounds or glue code that reduces the maintainability of the resulting design.

Composition anomalies are examples of the last two categories only. In order to be composable concerns have to be both *functionally* and *procedurally* composable [123]. Functional composability depends on composition being conceptually sound. For instance, it makes sense to compose a buffer concern with locking facilities but not with a random number generator. The composition must have useful and correct semantics, which means that the integration of subconcerns must provide the intended functionality. For example, although it is meaningful to compose a graph representation with an algorithm for counting subgraphs, the two are semantically uncomposable if they are represented as sets of vertices and adjacency lists. Procedural composability refers to interoperability or the dependencies and interactions between components. There are three kinds of procedural composability:

- Signature level composability refers to the signatures of various components.
- Protocol level composability refers to the ordering of operations.
- Semantic level composability refers to the semantics of the composed operations.

For a given composition scheme, a composition anomaly occurs when concerns which are functionally and procedurally composable either cannot be composed using the composition scheme or deviate from the expected interaction. The composer may adopt a different composition scheme, modify one or more of the input concerns, or create glue code to achieve the desired behaviour. According to Tekinerdogan et al,

> “Although there is no fundamental problem with the need for additional code, it turns out that this reduces quality properties such as adaptability, reusability and maintainability, in virtually all cases.”

With respect to SOP interaction problems, choosing a different composition scheme is synonymous with changing the composition specification or defining new composition rules; modification of input concerns is analogous to subject modification; and glue code is the creation of a new subject to patch up an interaction. A different composition rule may not be available or it may not be possible to create a practical implementation within the SOP composition framework.

Composition Rules versus Refactoring

One problem with reusing subjects is the absence of adequate join points [7]. In the terminology of Tekinerdogan et al [123] from the previous Section, the subjects are functionally composable but the set of available composition rules cannot establish protocol level composability.

Lopes et al believe [77] that more powerful means of referencing are required, capable of exposing all kinds of join points. A powerful composition language enables the separation of more concerns and makes it easier to change the specification of interaction without changing the structure of the module. An alternative to new composition rules is *refactoring*. Refactoring is a semantics preserving program transformation [124]. Where possible, the subject composer may refactor a subject to expose join points for the purpose of composition. Refactoring should be the preferred approach when it leads to lower coupling between subjects. Refactoring can also make composition specifications shorter and easier to understand.

In the following discussion of interaction problems, it will be assumed that join points necessary to express the composition exist already.

4.1.3 Interaction Analysis in AOP

Interaction problems have been observed in the domain of Aspect-Oriented Programming. This is not surprising given that separation of feature concerns and cross-cutting concerns from the solution domain both use join point interception.

Douence et al [33] propose to formally analyse stateful aspects. Analysis takes place on an asymmetric AOP framework which supports the concepts of join point, aspect and aspect composition. Aspect composition determines when aspects match; aspects are said to interact when they match the same join point. Stateful aspects are defined in terms of sequences of join points; they take account of the history of computation instead of a single point. The framework permits static analysis of interactions between aspects. Specifically, it is possible to detect when aspects do not interact. The framework supports composition rules which specify the correct order for execution of aspects when multiple aspects affect a join point. The composition rules are the main means of resolving conflicts. Aspect reuse is addressed by the use of explicit requirements on the base program. These requirements specify join point history conditions that ensure the correct application of the aspect.

The reuse requirement part of Douence's AOP framework is of use to the aspect developer also. It makes explicit the otherwise implicit requirement for the correct application of an aspect, making it useful for supporting the modular development of the aspect.

Katz [62] proposes to diagnose harmful aspects using regression verification. He defines harmful aspects as those that make the desirable properties of the base object-oriented system untrue in the combination of the base with the aspect. Regression testing is a process by which the system is tested with every new aspect that is added to it, to ensure that the test suite which previously was passed is still passed. Regression testing is not well suited to aspect-oriented systems because aspects affect the original control flow making original tests irrelevant. As aspects inherently affect many parts of the program, it is difficult to determine what part of the test suite might still be relevant. Instead, Katz proposes regression verification as a combination of static type analysis, deductive verification and model checking.

Static analysis can prove that an aspect does not invasively affect the base system. Deductive verification over aspect code can establish a lack of harm with respect to specific properties, e.g. an invariant of an existing system can be shown to be an invariant of the system containing the aspect. Model checking techniques can help to detect harmful aspects by showing that each interaction specified in the aspect is acceptable. Each interaction triggers a set of verification tasks and all required test conditions are automatically checked. In order to enable regression verification, the systems to which the aspects are added need to be augmented with specifications describing the desirable properties of the system. Regression verification may also be applicable to subject-oriented development. However, it requires better discipline on behalf of the subject developer to specify the desirable properties of subjects.

4.1.4 Towards Understanding Interaction Problems

The following three Sections present three interaction problems. These cases have been chosen because they demonstrate the kinds of actions that need to be taken in order to correct an anomaly. In the worst cases correction entails invasive modifications to subjects or patching. In order to understand the impact of subject-oriented decomposition on interaction problems, every subject-oriented solution is compared with a functionally equivalent object-oriented solution.

4.2 Persistence and Association

The first example of an interaction problem is concerned with the ordering of subjects in their interaction with each other[1]. There are three concerns, implemented as three subjects.

- The Persistence concern stores objects' fields in a file system or a database. An object is made persistent automatically when another persistent object holds a reference to it. Once an object becomes persistent it stays persistent until it is destroyed.
- The Association concern updates binary associations between objects. Suppose x and y are related by association, if some object x is set to reference y then y is set to reference x. Each object can be involved in at most one association relationship.
- The Transaction concern implements a business case relating a `Customer` and an `Order`. A customer references an order and an order is associated with a customer.

The code for these subjects is given in Figure 4-1. By bringing these three subjects together we create a new kind of `TransactionPA` subject that supports persistence and association functionality. Combinations of `Transaction` with `Association` or of `Transaction` with `Persistence` work very well. But when the three are brought together, their order of interaction becomes significant.

Let us consider the first of two composition specifications.

```
compose Persistence, Association, Transaction into TransactionPA;
mergeByName;
equate class AssocX, Customer into Customer;
equate class AssocY, Order into Order;
equate field AssocX.y, Customer.order into order;
equate field AssocY.x, Order.cust into cust;
equate operation AssocX.setY, Customer.setOrder into setOrder;
order operation AssocX.setY after Customer.setOrder;
equate operation AssocY.setX, Order.setCust into setCust;
order operation AssocY.setX after Order.setCust;
bracket ‘‘*.set’’ with before Persistence.PersistentClass.setValue;
```

[1] Adapted from an example in Renaud Pawlak's PhD thesis [98].

```
subject Persistence {
  class PersistentClass {
    static Store s;
    void setValue(Object value) {
      if(s.isPersistent(this)) {
        s.makePersistent(value);
    }
  }
}

subject Association {
  class AssocX {
    AssocY y;
    void setY(AssocY y) {
      if(y.getX() != this) y.setX(this);
    }
    AssocY getY() { return y; }
  }
  class AssocY {
    AssocX x;
    void setX(AssocX x) {
      if(x.getY() != this) x.setY(this);
    }
    AssocX getX() { return x; }
  }
}

subject Transaction {
  class Customer {
    Order order;
    void setOrder(Order order) { this.order = order; }
  }
  class Order {
    Customer cust;
    void setCust(Customer cust) { this.cust = cust; }
  }
}
```

Figure 4-1: The subjects implementing the `Persistence`, `Association` and `Transaction` concerns

Consider a program that has a customer object `aCust` and an order object `anOrder`. To start with, `aCust` is transient and `anOrder` is persistent, and there is no link between them. We execute `aCust.setOrder(anOrder)`. In addition to the behaviour specified in Transaction, the effect of this interaction should be to make `aCust` persistent and to set `anOrder` to reference its customer. The following sequence of method bodies is run:

1. `Persistence.PersistentClass.setValue`: receiver object `aCust` is not persistent, do nothing.

2. `Transaction.Customer.setOrder`: set the customer to reference the order.

3. `Association.AssocX.setY`: `anOrder` does not have a reference to `aCust` so update the association.

4. `Persistence.PersistentClass.setValue`: receiver object `anOrder` *is* persistent, so save the state of `aCust`, i.e. save `anOrder.cust = null`.

5. `Transaction.Order.setCust`: set the order to reference the customer. That is, `anOrder.cust = aCust`.

6. `Association.AssocY.setX`: `aCust` does have a reference to `anOrder`, so do nothing.

After this interaction, the value in storage is different to the value in memory. In store we have `anOrder.cust = null` and in memory `anOrder.cust = aCust`. This is an anomalous interaction.

Suppose we change the composition specification such that the Persistence concern is applied *after* the **merge** of Association and Transaction, i.e. replace line 11 in the above composition specification with:

```
bracket ‘‘*.set’’ with after Persistence.PersistentClass.setValue;
```

Now we get the following sequence of calls. The values in storage and memory are the same, and the interaction is as intended:

1. `Transaction.Customer.setOrder`: set the customer to reference the order.

2. `Association.AssocX.setY`: `anOrder` does not have a reference to `aCust` so update the association.

3. `Transaction.Order.setCust`: set the order to point to the customer.

4. `Association.AssocY.setX`: do nothing because the customer already references the order.

5. `Persistence.PersistentClass.setValue`: control is still with the order object. `anOrder` is persistent, so save the state of `aCust`, i.e. save `anOrder.cust = aCust`.

6. `Persistence.PersistentClass.setValue`: control has now returned to the customer object. `aCust` is now persistent. Save `aCust.order = anOrder`.

4.2.1 Interaction Problem Analysis

Would this problem occur in an object-oriented program? A functionally similar solution is the one where the `Persistence` and `Association` concerns are tangled and scattered in a single program. In an object-oriented program, the interaction is a sequence of statements, probably appearing within a single method body. This problem is much less likely in object-oriented programming because the statements are localised. It is also possible that the subject composer as a reuser has failed to understand the artifacts well enough in order to reuse them successfully.

The understandability of reuse artifacts may be improved by defining composition interfaces. However, we are reluctant to define composition interfaces because they may restrict composition in ways not anticipated by the original developer.

The interaction problem may be avoided through better concern modelling. Sutton and Rouvellou [120] propose Cosmos for modelling concerns through the lifecycle. A concern in Cosmos is 'any matter of interest in a software system'. A Cosmos model provides a form of documentation for basic information about concerns and their relationships. The detailed information about concerns is found inside subjects (or other design and implementation artifacts) but a schema affords a global perspective. Relationships between concerns can be modelled within the schema. *Physical relationships* describe the composition dependencies between concerns. A specialisation of the Cosmos schema to SOP may be used to describe the precedence of concerns in the above example, e.g. `Persistence activatesAfter Association` where `activatesAfter` is a physical relationship.

After the relationship has been identified, assertions can be used to validate the composition [69]. Assertions describing composition relationships between concerns are an extension to design-by-contract rules as used in the Eiffel language [85]. Assertions can validate that the application contains a suitable set of subjects applied in the right order but it cannot help to identify in which order the subjects should be composed.

4.3 Water Beans

Our second example concerns the development of a series of JavaBeans components [119]. JavaBeans are reusable software components that can be manipulated visually in a builder tool and composed to create end-user applications. The requirements for an object to be a bean are as follows:

- Objects must have a zero-argument constructor and must be either Serializable or Externalizable.

- Any properties of the object that are to be treated as bean properties, changeable by the user, should be indicated by the presence of appropriate get and set methods whose names are `getP` and `setP` where `P` is property name.

- Some bean properties, known as bound properties, fire events whenever their values change so that any registered listeners (i.e. other beans) will be informed of those changes. Making a bound property involves keeping a list of registered listeners, and creating and dispatching event objects in methods that change the property values, such as `setP` methods.

The application consists of three beans: `WaterSource`, `Valve` and `Pipe`. These can be connected arbitrarily while observing the condition that no component can be connected up-stream of a

`WaterSource` component. Water can flow *out* from any `WaterSource`, `Valve` and `Pipe` to any number of `Valve`s and `Pipe`s. Water can flow *in* to a `Valve` or `Pipe` from any number of `WaterSource`s, `Valve`s and `Pipe`s. Water comes from `WaterSource`s at some user specified volume and all components have a graphical representation that shows when water is flowing. From the topology of the example one should be able to determine the water pressure in any given pipe segment.

The client would like to build networks of Water Beans by using the Bean Box environment for JavaBean experimentation [31]. The Bean Box environment allows beans to be connected in order to enable data flow between them. The Water Beans have two responsibilities. First, they should enable water pressure to be determined based on the supply volume and the network topology. Secondly, they should have a graphical representation which conveys a sensation of water flowing through the network.

SOP may be used to extract bean behaviour [51]. The benefit of separating bean properties from an abstract data type, e.g. a `CartesianPoint` class, are clear – it untangles the ADT implementation from code for listeners and automatic firing of events for changed property values. The benefit of separating bean properties from the Water Beans is less clear – the event system and property value changes are part of the model. Water Beans properties are conceptually tangled with the JavaBeans implementation logic and separation in code would be overkill. Therefore, the system is decomposed into the following two subjects based on the two main concerns identified from the requirements.

- The `WaterPressure` subject abstracts the algorithms for calculating the water pressure for each pipe segment in the network based on some user specified supply volume and the properties associated with the valve or pipe. Supply volume downstream from a valve drops to zero when a valve is closed. Supply volume is restored when the valve is open. The topology and the state of the pipe network determines the pressure in any given pipe.
- The `Graphics` subject contains Water Beans drawing algorithms. The design of the subject follows the Model-View-Controller paradigm [20] whenever possible, separating the drawing algorithms from the model that controls their activation. The *controller* interprets user input during network design in the Bean Box. The Bean Box creates connections between the beans. In the *model*, instances of the `WaterEventObject` class 'carry' the water supply. The `WaterSource` 'drips' one `WaterEventObject` per second to its list of listeners. An open `Valve` passes on the received `WaterEventObject`s to its `WaterListeners`. A closed `Valve` does not pass on any `WaterEventObjects`. A `Pipe` works in the same way as an open `Valve`.

This decomposition modularises the definition of graphics for the Water Beans, localising the changes that affect the aspect of Water Beans visual representation. Also, it becomes possible to create families of Water Beans, i.e. with or without the `WaterPressure` concern.

Both the `WaterPressure` and the `Graphics` subjects use the event system to enable communication between the Water Beans. The client requires that connections between the Water Beans should be established by drawing a single link from the upstream component to the downstream component. This integration requirement indicates that the event systems of each subject should be combined.

Before a detailed design can be developed for each subject we have to work out the subjects' composition interfaces. In order for the event models to be composable a degree of coordination is required in the area of event model design. One area of interaction identified between concerns is water supply volume dropping to zero. This event is similar to a valve being shut; the water stops

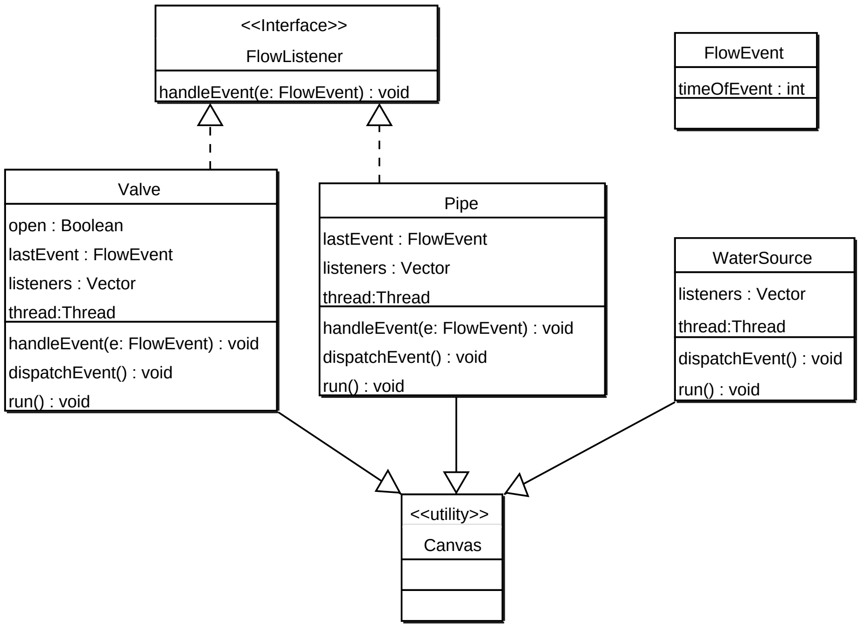

Figure 4-2: Water Beans class diagram for the `Graphics` subject

flowing. Once the composition interface has been defined, work can commence independently on the detailed design for these subjects.

4.3.1 Detailed Design Considerations

The `Graphics` subject uses the JavaBeans event model to simulate water delivery. Every event carries a timestamp. In the model underlying the `Graphics` concern, a Water Bean is considered to be carrying water if the timestamp of the last event is less than 2 seconds before the present time. If the last event came more than 2 seconds ago then the bean changes its representation to indicate that water is no longer flowing. The effect of the model is to produce the sensation of water emptying over time rather than instantaneously. Figure 4-2 contains the class diagram of the main

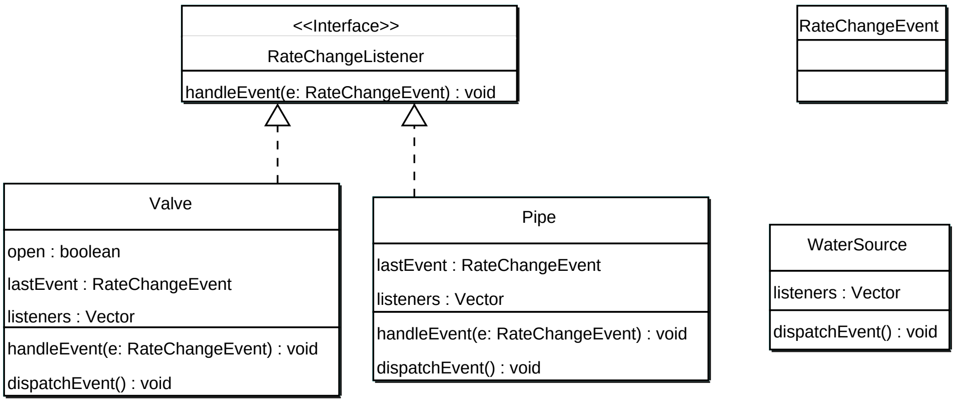

Figure 4-3: Water Beans class diagram for the `WaterPressure` subject

features of the Graphics subject. Each bean uses a thread to check for elapsed time. Method `run()` executes a loop that periodically compares the timestamp of the last event against the current time.

The `WaterPressure` subject uses the JavaBeans event model to propagate changes in supply volume. `Pipe`s have an observable pressure property that is determined by the rate at which water enters the pipe (rate equals volume over time) and the diameter of the pipe. `Valve`s are either fully open or fully closed; either they let the full volume of water pass through or none at all. When the water supply volume changes the change in water pressure in each pipe is virtually instantaneous. Figure 4-3 contains the class diagram of the main features in the `WaterPressure` subject. `handleEvent(..)` methods are called by the bean framework when a new event is received. The `dispatchEvent()` methods are called when the rate change is propagated to listeners.

To compose, **merge** integrates the listener interfaces, the event classes, and the Water Bean classes that correspond by name.

4.3.2 An Interaction Problem

Composition of these subjects produces an anomaly. When a closed valve `v` opens, the volume of supplied water toggles from zero to the volume upstream from the valve. The upstream rate is taken from the last event that `v` receives. The change in volume causes `v` to release events to all listeners. When a bean downsteam of `v` receives the event, it coincidentally changes its visual representation because the event also indicates that water is flowing. In the meantime, `v` is still showing that water is not flowing because it is waiting for an event from an upstream object. Finally, when `v` receives an event from an upstream bean, its representation changes to indicate that water is passing through.

Figure 4-4 shows a time trace for this interaction problem. `WaterSource` (visually represented by a tap) supplies water at 10 litres per minute. The tap is connected in sequence to a pipe, followed by a valve, followed by two more pipe segments. At time t the valve is closed, there is water in the pipe preceeding the valve but no water beyond it. The water pressure before the valve is 5 bar and 0 bar afterwards. At time $t + 1$ the valve opens. In the `WaterPressure` subject, the valve creates and dispatches an event to all listeners to indicate that a change in supply volume has occurred and, consequently, a change in pressure. Coincidentally, the pipes downstream from the valve interpret the incoming event as water flow, rendering the pipes a grey colour. The valve is still clear because its representation will change only in response to an incoming event. At time $t + 2$ the event from the pipe connected immediately before the valve arrives at the valve, and the program returns to the correct state.

There is no simple way to correct this interaction problem. On the one hand, keeping the event models separate leads to no undesirable interactions. On the other, there is a requirement to integrate event models which leads to this anomaly. With the SOP integration rules we have described up to now, to rectify the interaction we must modify the input subjects or create a new subject to patch things up.

The problem is that the `WaterPressure` and `Graphics` subjects use the event model in different ways. The integration requirement demands a single event type to describe both variants. In effect, a single event type quantifies over or generalises the two uses of the event system. A distinction between different events is required in order to specialise the treatment for each event type. SOP does not define integration rules to express this relationship at present.

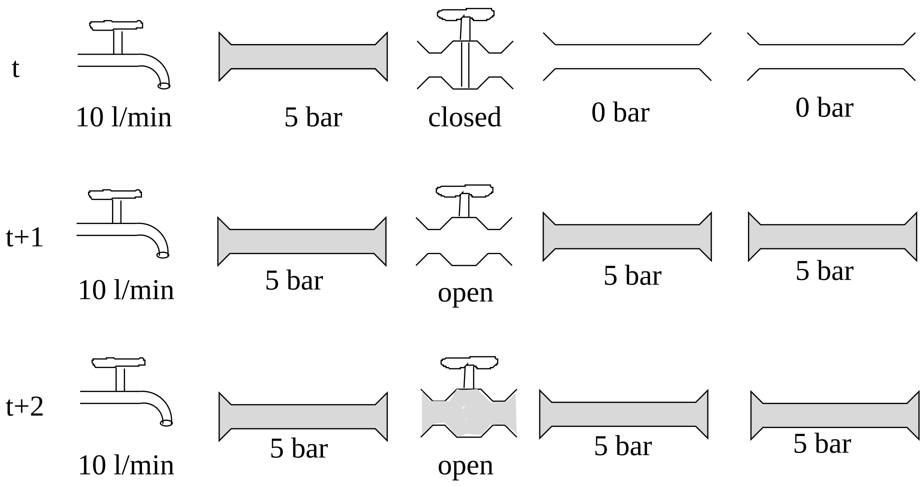

Figure 4-4: The Water Beans interaction problem.

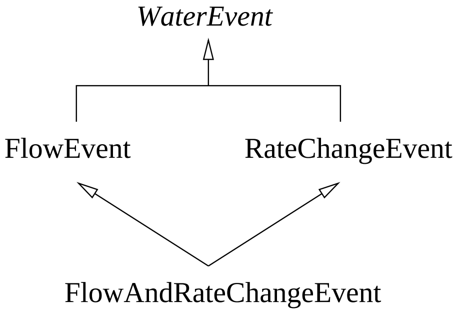

Figure 4-5: Water Beans conceptual event model

4.3.3 An Object-Oriented Solution

Lets look at the way the Water Beans could be designed as an object-oriented program. It is possible to model this interaction using inheritance. We need to define a common event type such that a single connection between Water Beans is sufficient to establish their interaction. Also, it should still be possible to distinguish between three kinds of event: water flow, water pressure change and the union the two. Figure 4-5 shows an inheritance hierarchy describing this relationship. At the root of the hierarchy is an abstract `WaterEvent` class which quantifies over all other kinds of events. The `WaterPressure` concern creates `RateChangeEvent`s, and the `Graphics` concern creates `FlowEvent`s. When a new Water Bean is connected into the network, the upstream bean should dispatch a `FlowAndRateChangeEvent` object. Conceptually, this is an action that is performed jointly by the Graphics and WaterPressure concerns.

There are problems with implementing this design elegantly in JavaBeans.

- Java does not support multiple inheritance of implementation, so the `FlowAndRateChangeEvent` has to be declared as a direct subclass of `WaterEvent` instead. This increases code duplication.
- Java does not have multiple dispatch – a variant of the inheritance mechanism that uses the dynamic type of the parameters as well as well as the dynamic type of the receiver to choose the method to dispatch. Multiple dispatch enables the dynamic selection of the right

```
void handleWaterEvent(WaterEvent e) {
  if(e instanceof FlowEvent) { ... }
  else if(e instanceof RateChangeEvent e) { .. }
  else if(e instanceof FlowAndRateChangeEvent e) { ... }
  else { ... }
  ..
}
```

Figure 4-6: Simulating multiple dispatch in Java

method to execute depending on whether one gets a `FlowEvent`, a `RateChangeEvent` or a `FlowAndRateChangeEvent` object from an upstream bean. Instead, the receiver must manually switch based on the type of the incoming event as shown in Figure 4-6.

- There are conceptual problems with using Java's inheritance mechanism to model this relationship. The `Graphics` concern is interested only in the timestamp of the event, whereas the `WaterPressure` concern is interested only in the volume of water. Inheritance in Java is monotonic – classes cannot disinherit the methods they inherit. One can call the `getVolume` method on a `FlowEvent` object in spite of its not being well defined conceptually. This can lead to future reuse problems.

Clearly, an object-oriented design can be created to address this problem. But a Java-based solution does not have inheritance relationships which can model the conceptual relationships faithfully. In other languages, e.g. Eiffel [84], the unsuitable methods may be disinherited, thereby addressing the last point above.

Gardner has proposed Structured Inheritance Relationships (discussed in Section 2.4.2 on page 19) in order to overcome problems associated with inheritance, improve conceptual modelling and facilitate reuse [44]. Among her recommendations is a **View** inheritance relationship which allows a subclass to provide an alternate interface to target objects that is more appropriate to some clients than the interface provided by the default view. Languages that provide natural classifications of objects will produce more robust object-oriented programs than those that model one solution to the problem [81].

4.3.4 A Solution For Subject-Oriented Programming

The interaction problem can be addressed by a composition rule that allows objects to be distinguished by the subject that instantiates them. We propose the **view-merge** integration rule for dynamic selection based on subject of origin. The **view-merge** rule is inspired by the **View** structured inheritance relationship [44] but scaled to subject granularity.

The **view-merge** integration rule can be applied on a set of corresponding classes. When a **view-merge**d object is received as an operation parameter, the method body to execute is selected based on the subject which instantiated it. An object is considered to be instantiated by subject S when it is:

- instantiated on an instance variable in S that has no corresponding instance variables, or

- created in the body of a method in S with the exception to those methods which have corresponding `abstract` methods in other subjects.

When subjects $S_1 \ldots S_n$ are composed, an object is considered to be instantiated by all subjects when it is

- instantiated on an instance variable v_i in S_i and there exists some v_j in S_j with $i \neq j$ such that v_i, v_j correspond, or
- created in the body of method m_i in S_i and there exists some m_j in S_j with $i \neq j$ such that m_i, m_j correspond and m_j `abstract`.

The last of these deserves an additional explanation. Recall that in SOP, a method is declared `abstract` when another subject is responsible for its implementation. When a set of `abstract` methods is **merge**d with implemented operations, the effect is to share the implementations between all host classes. Objects created in these operations are classed as belonging to all subjects.

We believe that the **view-merge** rule can be implemented within the SOP rule framework. Whereas the standard **merge** rule in SOP creates a single class in the output from a set of input subjects, **view-merge** creates an inheritance hierarchy as follows:

> When **view-merge** is applied to $S_1.A$, $S_2.A$ and $S_3.A$, create class X by **merge** integration of $S_1.A$, $S_2.A$ and $S_3.A$. Also create three subclasses of X called $X\_S_i.A$, $i \in \{1, 2, 3\}$, by augmenting the interface of each $S_i.A$ with stub methods order to make them into proper subclasses of X (to satisfy Java type rules). When A is instantiated in S_1, create an instance of $X\_S_1.A$ if the point where instantiation occurs is exclusive to S_1, but create an X object instead if the instantiation point is shared by two or more subjects. For example, if A is instantiated in a method exclusive to S_2 then instatiate an object of type $X\_S_2.A$. But if A is instantiated in a method in S_2 that *does have* corresponding `abstract` methods then instantiate X instead.
>
> When an object of a **view-merge**d class is received as a parameter, the method body to execute is selected dynamically using the `instanceof` operator in Java to simulate multiple dispatch. The code of the kind given in Figure 4-6 for dynamic selection is injected automatically into all methods that have a parameter of type A.

The details of rule implementation are hidden from the rule user. In the Water Beans application, **view-merge** can be applied to the composition of `FlowEvent` and `RateChangeEvent` classes to create class `FlowAndRateChangeEvent`. When `Graphics` instantiates an event object and sends it to listeners, only the `Graphics` subject's handler method gets executed. Similarly, when `WaterPressure` instantiates an event and sends it to listeners, only the `WaterPressure` subject's handler method is executed. However, when a method belonging to both these subjects instantiates an event object, both handlers will be run.

Further development of this rule should extend it to operations with multiple parameters. Consider method `doSomething(A a, B b)` where `A` and `B` are **view-merge**d. The dynamic types of both `a` and `b` must be used to select the method body to execute. In order to continue the discussion on interaction problems, the design of this composition rule is not developed further at this point.

Other rules may be inspired by various uses of inheritance; subject composition rules describe relationships between abstractions that may be modelled with inheritance in object-oriented programming. Development of integration rules is open-ended in nature. It depends on the existence of a suitable composition framework – this is already the case with Subject-Oriented Programming. By increasing the composition vocabulary it becomes possible to compose more subjects non-invasively.

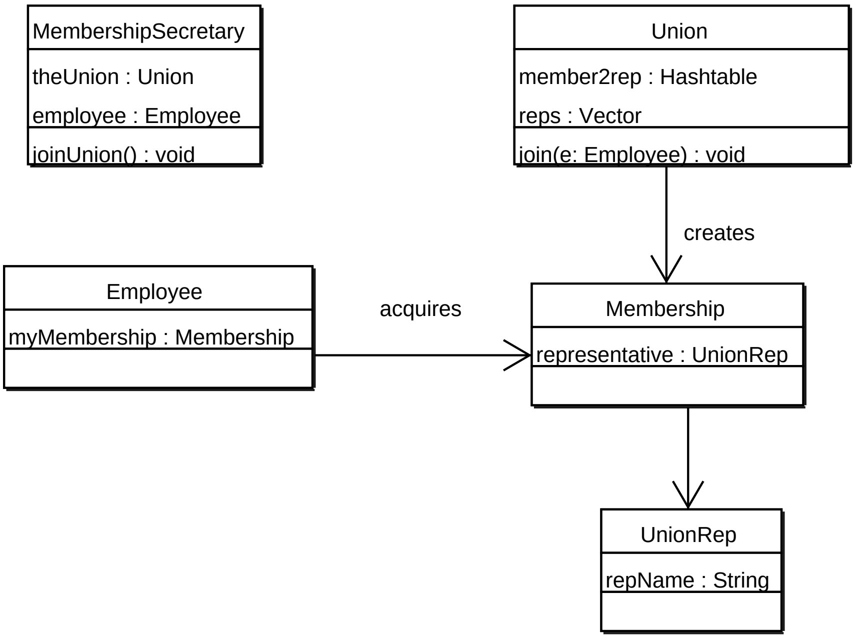

Figure 4-7: The `JoinUnion` subject class diagram

4.4 Union Members and Representatives

The third anomaly we present is not solved by changing the order of subject composition or by defining more powerful composition rules. The interaction problem is caused by incompatible domain views. The example concerns the development of software for a trade union. The initial description of system functionality is as follows:

> The employees of a company can become members of a union. The union is a large organisation consisting of workers and union representatives. Upon joining, an employee is assigned a representative who advises the member on his or her rights in case of an industrial dispute or if a member feels that he has been treated unfairly by the company which employs him. An employee deals solely with his union representative. The representative handles small cases personally. Bigger issues are taken back to the union committee who take a collective decision on behalf of all members. An issue which all union representatives get to deal with is dismissals. The representatives deal with dismissals on a personal basis. A dismissed employee contacts the representative who can investigate the causes of a dismissal and so on.

During analysis, joining the union and dismissal features are identified. In the design the `JoinUnion` subject describes the way an employee becomes a union member. The `Dismiss` subject describes the procedure involved in getting help when an employee is dismissed. These subjects are shown in Figures 4-7 and 4-8.

In the `JoinUnion` subject, the `MembershipSecretary.joinUnion` method sends an `Employee` object as argument to `Union` to register him as a member. `Union` generates a `Membership` object that is stored by the `Employee`. A `Membership` contains a reference to the employee's `UnionRep`.

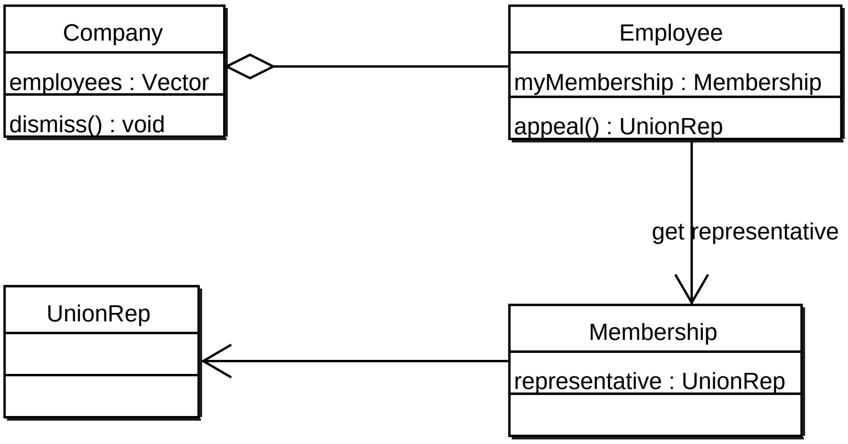

Figure 4-8: The `Dismiss` subject class diagram

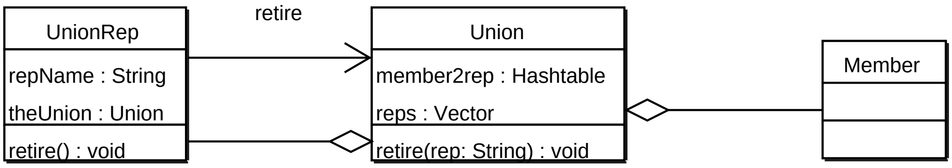

Figure 4-9: The `Retire` subject class diagram

The interaction in the `Dismiss` subject starts with the `Company` object that `dismiss`es an `Employee`. The `Employee` can call the `appeal` method. This returns the assigned union representative in the form of a `UnionRep` object to handle the allegations of unfair dismissal.

The composition specification joins all identically named artifacts. `Membership` classes are joined, `representative` field is the same union representative in both subjects. The two views of the `Employee` class share the `myMembership` field also.

An additional requirement is introduced into the design at a later stage. A union representative is now able to retire from the post. `UnionRep` sends a `retire` message to the `Union` giving his name. The members that the retiring representative served are assigned an alternative representative from the pool. This concern is elegantly captured by the `Retire` subject as shown in Figure 4-9. The collaboration involves updating the `member2rep` collection to reference some new representative, then removing the retiree from the `reps` pool. When composing, we equate classes `Employee` and `Member` for they represent the same abstraction from different perspectives.

4.4.1 An Interaction Problem

There exists an anomaly in the composed program that emerges during a particular interaction between the `Retire` and `Dismiss` subjects. The problem manifests itself when an employee is dismissed after his union representative retires. The union now associates the employee with a new union representative but the employee's membership still refers to the retired representative. The `Dismiss` subject returns the retired union rep instead of his replacement. The `JoinUnion` subject

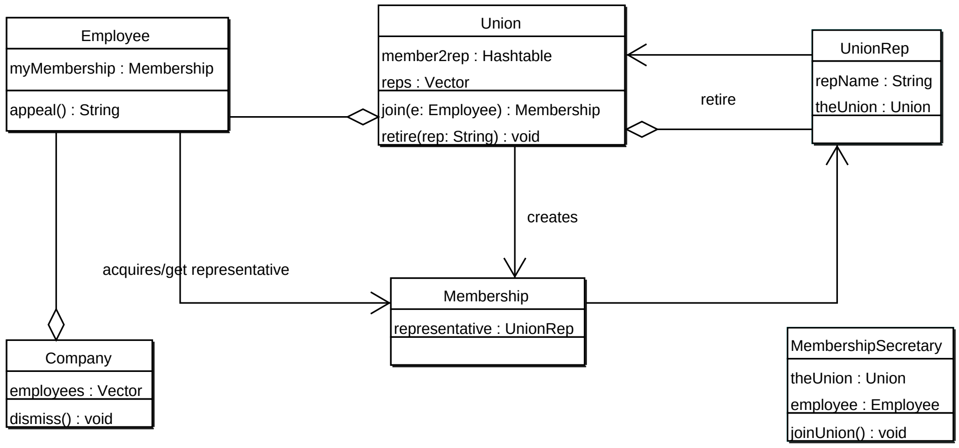

Figure 4-10: JoinUnion, Dismiss and Retire concerns as a (badly structured) OO program

informally expects the relationship between an employee and his union representative to be invariant. The `Dismiss` subject reinforces this assumption by relying solely on `myMembership` for information on the representative.

Invasive modifications to the Water Beans example seen earlier were avoided by defining composition rules which reflect the true relationship between abstractions. However, this anomaly cannot be solved non-invasively by addition of composition rules because the interaction cannot be expressed as a relationship between the composable elements of corresponding subjects.

We observe two ways of correcting this interaction with design. The first solution is to make the `Retire` subject update all references between the employee and the representative. The `Retire` subject becomes responsible for updating each `Employee`'s membership details in the `Union` object and all `Membership` objects. The second requires reconstruction of `JoinUnion` and `Dismiss` concerns such that `Union` class becomes the only source of information regarding the relationships between employees and representatives. The first solution extends the design which has outgrown its usefulness in a highly coupled way. To its credit, the first solution requires no modifications with respect to the `JoinUnion` and `Dismiss` subjects. According to Tekinerdogan et al [123], it is a composition anomaly because the additional code affects quality factors (see Section 4.1.2 on page 47). Also, the explicit capture of cross-cutting concerns in subjects should be the natural consequence of good modularity and not the result of a corrective measure due to a tangled implementation [29]. The second solution is a better design because it localises the employee-union representative relationship but it requires invasive modifications to all input subjects. Clearly, neither solution is ideal.

The cause of this interaction problem is understood best by looking at an object-oriented design for `JoinUnion`, `Dismiss` and `Retire` concerns.

4.4.2 An Object-Oriented Solution

An object-oriented design addressing these concerns is shown in Figure 4-10. The interaction problem experienced in the output of subject-oriented composition occurs also in this object-oriented program. This kind of problem in object-oriented programming has been blamed on uncontrolled object aliasing [57].

In object-oriented programs, objects are passed by reference. An object is said to be aliased when there are two or more references to it. Object state depends on the values of its variables and the state of other objects it references. Aliasing problems can occur when an object reveals references to the objects contained within through the interface. The client can obtain a reference and proceed to dispatch messages to the reference without going through the interface of the object that revealed it. This can pose problems when trying to understand object-oriented programs because a state change to one object affects all others that alias it. Confusion can arise when object state changes not as a result of a message sent to its interface but because of an alias into its state.

Aliasing problems are particularly acute in object-oriented programming because most objects have mutable state. Object state mutates as the result of changes to values and the state of referenced objects. Some objects are immutable: their state cannot be changed although the values of variables referencing them can. This includes atomically typed objects such as integers and booleans. Immutable objects can be aliased more freely because their state cannot be modified.

To address aliasing problems, a number of researchers have proposed *Alias Protection Systems* (APSs) [56, 6, 91, 5, 127, 23]. At the core of any APS is a concept of object representation. Representation consists of objects that are used in the implementation of abstractions (classes). Pragmatic approaches to alias protection do not hide the representation within one object but enable controlled exposure to support idiomatic uses of object-oriented programming. Flexible Alias Protection (FAP) [92] is one of the most advanced APSs. The inspiration for FAP came from the observation that problems are not caused by aliases per se; rather, they are due to non-local changes caused by aliases. Aliasing should be allowed so long as the visibility of changes to objects is controlled. The Union example could be redesigned, using FAP to enforce encapsulation, thus steering clear of the aliasing problem in Figure 4-10.

4.4.3 Redesigning the Object-Oriented Solution

Flexible Alias Protection takes the form of aliasing mode declarations (additional types) that are inserted into code by the programmer and checked statically by an automated checker. The full details of FAP are not relevant for our example[2]. Suffice it to say that in our example objects of non-value types have one of three modes:

- *Representation* objects (mode `rep`) can be manipulated freely inside the container but never exposed.
- A representation object can be passed to internal containers as an *argument* object (mode `arg`). To minimise its effect, an argument object must appear immutable from the perspective of objects that access it. The messages sent to argument objects should appear to be purely functional.
- *Variable* objects (mode `var`) may be manipulated and aliased freely anywhere in the program in the same way as objects in a mainstream object-oriented programming language.

Most objects require a single mode to describe the aliasing policy. Classes implementing ADTs, such vectors and hashtables, require additional modes: vectors require one additional mode for the elements stored inside; hashtables require two additional modes – one for the keys and one for the values. Being able to specify the mode of container elements parametrically makes ADTs more reusable. Assignment between expressions of different modes is forbidden in most cases, but an

[2] Chapter 5 on page 70 contains a detailed review of Alias Protection Systems.

object can be viewed using different modes at different times. A `rep` object can be viewed as an `arg` object when it is passed to another `rep` object for storage, e.g. a union representative can be stored in a data structure containing all other union representatives.

Figure 4-11 contains the improved object-oriented implementation for the `JoinUnion`, `Dismiss` and `Retire` concerns annotated with FAP aliasing modes. The aliasing problem has been elimintated in this design. The `Membership` class has been deleted and `Employee` objects use `Union` as the sole source of employee-union representative relationships. In the `Union` class, a hashtable is used to store member-to-representative associations (line 29). This hashtable is the private representation of its `Union` object as indicated by the leading `rep` mode. Inside the angle brackets, members can be referenced but not modified by the `Union` as indicated by `arg`. Union representatives have mode `rep`; they are treated as representation objects which must not be exposed externally.

During a dismissal appeal (line 9), when `getRepName` method is called by the employee, instead of returning a reference to a `UnionRep`, we return its `repName` field (line 38 followed by 23). This object is immutable; once created, it is accessed in a purely functional way, leading to no unexpected state changes in objects that reference it directly or transitively.

4.4.4 Towards a Solution for Subject-Oriented Programming

In object-oriented programming FAP aliasing modes serve two roles. The first role is alias protection; it is to protect representation objects from external access. The second role is annotational. Aliasing modes annotate object usage, describing the permissions to access and modify the state of an object. It is the second role which motivates our application of FAP to Subject-Oriented Programming. Aliasing modes may also be useful for describing the way subjects use objects. At present in Hyper/J, Java types are the only interface-level formalism for checking composition validity, and the composer must rely on informal documentation or implementation inspections to check that subject interaction is problem free. The addition of aliasing modes will improve the composer's ability to reason about interaction.

In Figure 4-12, FAP aliasing modes are applied to the `JoinUnion` (fig. 4-7), `Dismiss` (fig. 4-8) and `Retire` (fig. 4-9) subjects. Subjects are purely object-oriented and aliasing modes are applied without making any structural or functional changes. No changes are necessary because on its own each subject is a reasonable object-oriented design. The modes in each subject are chosen independently from other subjects. When two or more modes are applicable, the conceptually most descriptive mode is chosen.

The modes were selected based on the following justifications. In the `JoinUnion` subject:

- `MembershipSecretary.theUnion` has mode `rep` to indicate that only this `MembershipSecretary` can visibly change the state of `theUnion`.
- `MembershipSecretary.employee` has mode `var` because an employee need not be a member of a union. Mode `arg` is not appropriate because of changes to an employee's mutable state when setting membership.
- `Employee.myMembership` has mode `arg` because changes to the state of `myMembership` are not expected after it is created.
- `Membership.representative` has mode `arg` to indicate that the representative's state is immutable.

```
class MembershipSecretary {
  var Union theUnion;
  var Employee employee;
  void joinUnion() { theUnion.join(employee); }
}

class Employee {
  var Union theUnion;
  arg String appeal() { return theUnion.getRepName(this); }
}

class Company {
  rep Vector<var Employee> employees;
  void dismiss() {
    var Employee e = selectEmployeeToDismiss();
    arg String repName = e.appeal();
    ...
  }
  var Employee selectEmployeeToDismiss() { ... }
}

class UnionRep {
  arg String repName;
  var Union theUnion;
  void retire() { theUnion.retire(repName); }
}

class Union {
  rep Hashtable<arg Employee, rep UnionRep> member2rep;
  rep Vector<rep String> reps;
  void join(arg Employee e) { member2rep.add(e, selectRepRandomly()); }
  void retire(arg String r) {
    rep UnionRep old = getRepByName(r);
    changeRep(old, selectRepRandomly());
    reps.remove(old);
  }
  void changeRep(rep UnionRep oldRep, rep UnionRep newRep) { ... }
  arg String getRepName(arg Employee e) { return member2rep.get(e).getRepName(); }
  rep UnionRep selectRepRandomly() { ... }
  rep UnionRep getRepByName(arg String repName) { ... }
}
```

Figure 4-11: OO Program implementing the JoinUnion, Dismiss and Retire concerns annotated with FAP aliasing modes

```
subject JoinUnion {
  class MembershipSecretary {
    rep Union theUnion;
    var Employee employee;
    void joinUnion() { theUnion.join(employee); }
  }
  class Employee {
    arg Membership myMembership;
  }
  class Membership {
    arg UnionRep representative;
  }
  class UnionRep {
    arg String repName;
  }
  class Union {
    rep Hashtable<var Employee, arg UnionRep> member2rep;
    rep Vector<arg UnionRep> reps;
    join(var Employee e) { ... }
  }
}

subject Dismiss {
  class Company {
    rep Vector<rep Employee> employees;
    void dismiss() { ... }
  }
  class Employee {
    arg Membership myMembership;
    arg UnionRep appeal() { ... }
  }
  class UnionRep {
  }
  class Membership {
    arg UnionRep representative;
  }
}

subject Retire {
  class UnionRep {
    var Union theUnion
    arg String repName;
    void retire() { ... }
  }
  class Union {
    rep Hashtable<arg Member, rep UnionRep> member2rep;
    rep Vector<rep UnionRep> reps;
    void retire(arg String repName) { ... }
  }
  class Member { }
}
```

Figure 4-12: `JoinUnion`, `Dismiss` and `Retire` subjects annotated with FAP aliasing modes

- `Union.member2rep` is the representation of its container. It cannot be exposed outside. `Employee` objects used as the hashtable keys have mode `var` because they can be aliased and changed outside this object. `UnionRep` objects used as hashtable values have mode `arg` because they are aliased but not visibly changed within this class.

- `Union.reps` is also the represention of its container. The decision to use a vector to reference all union representatives is a design decision that should be hidden in an OO program. `UnionRep` objects stored in the vector have mode `arg` because they are aliased but not visibly changed in this class.

In the `Dismiss` subject:

- `Company.employees` is a vector of `Employee` objects. `rep` mode on the vector and the elements within indicates that the choice to use a vector to reference employees is an implementation decision that should be hidden from clients. Company employees cannot be contacted directly by the company clients.

- `Employee.myMembership` has mode `arg` to indicate that the state of `myMembership` is not changed by this `Employee` instance.

- `Membership.representative` has mode `arg` to indicate that representative's state is not changed by this class.

In the `Retire` subject:

- `UnionRep.theUnion` has mode `var`. By retiring, the representative changes the state of the union. However, more than one representative may retire, requiring the union to be modified by multiple representatives.

- `UnionRep.repName` has mode `arg` because the representative's name is not expected to change.

- `Union.member2rep` is the representation of its object so it has mode `rep`. The union associates members to representatives but does not change them in any way. Members have mode `arg` because employees can be a member of more than one union. Representatives have mode `rep` because they are exclusively the representatives of this union.

- `Union.reps` is also the representation of its container. `UnionRep` objects stored in the vector have mode `rep` because they are exclusively the members of this union and should not be referenced outside.

Having specified the subjects, attention now turns to their composition. Composition of aliasing modes has not been addressed in existing work and requires further investigation before being applied. Two choices are apparent: either only elements with the same modes may be composed or it should be allowed to compose elements with non-matching modes. Mode equivalence is meaningful because subjects in Figure 4-12 are different views of the same domain. Lets consider the semantics of pairwise composition of modes. This is easily extended to n-ary compositions:

`rep` with `rep`. All compositions take place in the context of corresponding classes. Hence, this composition means that both subjects can alias this object freely inside the encapsulating object but not expose it to external clients.

arg with arg. Both subjects can pass the object freely but never modify it in a way that is visible.

var with var. Both subjects can alias and modify the object freely.

Composition of elements with the above modes leads to the same mode in the output as in the input but only if equivalent modes appear at all join points. Modes in different classes are interrelated, and a change in aliasing mode in one class would have a cascading effect on modes in other classes. Composing the above subjects, the views of corresponding elements in Employee/Member (different views of the same entity), Membership and UnionRep classes are equivalent across all three subjects. For class Union there are two places where modes are not equivalent:

- In JoinUnion, employees have mode var. This mode is required in order legally to pass an Employee instance when calling Union.join and to be able to modify it within Union. In Retire, employees have mode arg because the Union does not affect or depend on their state. However, purely from the point of view of aliasing other FAP modes can be used in this position in Retire.
- In JoinUnion, union representatives have mode arg because they are assigned to Employee.myMembership's representative field in the body of Union.join (method body elided in Figure 4-12). Mode var is also legal in this case. In Retire, union representatives have mode rep because conceptually Union objects cannot expose them. Although other modes are also valid here.

Although not a join point, MembershipSecretary and UnionRep refer to the same Union object in separate subjects. JoinUnion uses mode rep to specify that MembershipSecretary is the sole object that can modify MembershipSecretary.theUnion. Retire uses var to specify that any object can change the state of UnionRep.theUnion.

The different modes in corresponding elements and, occasionally, in non-corresponding elements indicate differences in the way the domain is perceived from the perspectives of different concerns. In many cases there is more than one choice of modes. Mode var is the superset of uses described by rep and arg; it can be used in places where the other two modes are appropriate, and, during composition, var may replace modes rep and arg.

The requirement of mode equivalence prevents composition and averts the interaction problem. The use of FAP modes for conceptual modelling of subject aliasing properties has helped to show that the subjects have differing views of the domain. However, this composition may not have been preventable if other modes had been chosen. The application of FAP concepts to Subject-Oriented Programming raises a number of interesting questions:

- Do aliasing modes help to understand subject interaction for the purpose of avoiding interaction problems?
- Is mode equivalence the only meaningful composition or is it also meaningful to compose elements with different modes?
- What are the criteria for choosing aliasing modes?

4.4.5 The Role of Aliasing Modes in Understanding Subject Interaction

In the terminology of Multi-Dimensional Separation of Concerns, it can be said that data sharing between subjects leads to 'object encapsulation' becoming a dimension of concerns. Subjects scatter and tangle code which in an object-oriented design would be hidden in the representation of some class. In the Union example broken object encapsulation has been shown to lead to an interaction problem.

Composition of elements with equivalent aliasing modes helps to address the scattering and tangling. It constrains composition but in a structured way. FAP does not reduce the set of available join points. Aliasing mode equivalence shows an agreement on object aliasing policy between subjects. Modes used as part of the compositional interface support independent subject development and subject reuse by strengthening the typing of composable elements of each subject. Aliasing modes help the composer to reason about subject interaction and also fit well with our reuse position. Subjects are object-oriented programs and object aliasing is a concern in subject design. An alias protection system such as FAP is of value to the subject developer: it helps to create well structured subjects that encapsulate and protect object representation from access at the subject's functional interface.

The Union example has shown that different modes for corresponding elements do not necessarily indicate interaction problems. The selected modes are not mutually exclusive. However, this need not be the case for other Alias Protection Systems. For example, consider mode `val` which describes value types and immutable objects. Suppose we compose two `String` classes and integrate two corresponding `String`-type variables from each subject. In the first subject the `String` class is immutable. In the second subject `String` is mutable; it introduces the `setValue(..)` method which enables the value to be changed. Suppose that in the first subject the variable has mode `val` and the second subject mode `rep`. These `String` classes should not be merged because the first subject may depend on `String` being immutable. There is no problem with composing elements with different modes *per se*. The challenge is determining the mode of the output element and of all other elements affected by this composition. Clearly, FAP was never intended for subject composition. Further work is required to determine the best modes to use and the policies for mode selection and composition.

4.5 Conclusion

This Chapter has described interaction problems in subject-oriented programs. We defined an interaction problem as an unwanted subject interaction. Interaction problems are undesirable because they raise the cost of subject reuse and impact modular development of subjects. In the worst cases, interaction problems require either invasive subject modifications or patching. This Chapter has presented a range of interaction problems occurring in Subject-Oriented Programming and suggested ways of tackling them.

The first example demonstrated the importance of concerns and the way they relate to each other. We looked at the combinations of `Persistence` with `Transaction` and `Association` with `Transaction`. However, it is insufficient to evaluate interactions in a pairwise manner. Unless there exists no connection between concerns then any interaction analysis must involve all concerns together.

The second example demonstrates the tension between concerns with respect to a shared resource. The intention of having a single connection between the Water Beans causes a single event model to be shared for carrying two kinds of event. The composition rules available in the SOP language Hyper/J cannot resolve the interaction without changes to the input subjects. However, the SOP composition framework allows other rules to be defined. The `view-merge` composition rule is proposed for addressing the anomalous interaction non-invasively.

The Union example demonstrates that interaction problems can be caused by encapsulation issues. Subjects have dependencies on object state that may be subverted during inter-subject interaction, resulting in interaction problems. Some data dependencies can be made explicit in subjects with the aid of modes proposed as part of Flexible Alias Protection [92]. For instance, aliasing modes can describe where and how object state can be changed. By using Flexible Alias Protection as part of the composition interface it becomes easier to observe data dependencies and detect interaction problems.

Alias Protection Systems (of which Flexible Alias Protection is but one) help to create well structured object-oriented programs. By ensuring the absence of external aliases into object representation they enable modular reasoning. This property makes APSs useful to the original subject developer who is also an object-oriented programmer. Being useful to the original developer and the reuser potentially makes APSs an excellent reuse technology in projects where future reusability is not part of the initial requirements. Moreover, APSs annotate the objects which depend on or modify the state of other objects. The annotational properties facilitate the reuse of subjects. For these reasons, we believe that APSs are a topic worthy of further investigation. In the next Chapter, we review the background on Alias Protection Systems and discuss alias annotation in the context of Subject-Oriented Programming.

Chapter 5

Alias Protection and Subjectivity

The previous Chapter described the current understanding of interaction problems – unwanted interactions between subjects. Interaction problems affect the reusability of subjects and are an impediment to independent subject development. In the worst case, the anomalous interactions can be corrected only by invasively modifying subjects or by defining a patch subject. In our example the problem was caused by broken assumptions about object state. We observed that the opportunity to detect the anomalous interaction improved when annotations from Flexible Alias Protection [92] were applied within each subject. The way subjects use and modify objects is made explicit with alias annotations. We believe that alias annotations will be generally useful to subject composers (that is, subject reusers) for the purposes of understanding interactions and preventing anomalies.

In object-oriented programming, unstructured aliasing has been identified with understandability and reasoning problems. Alias Protection Systems (APSs), including Flexible Alias Protection, constrain object aliasing in a structured way, improving object encapsulation and facilitating modular reasoning. In this sense, APSs have value to the subject developer. The subject developer, as an object-oriented programmer, is concerned with creating well structured and maintainable subjects. APSs fit well with our position on reuse: we believe that a reuse technology has more chance of being accepted when it has value to the original developer.

The aim of this Chapter is to present our understanding of the way subjectivity affects aliasing properties. We lay the foundation for creating an APS for SOP. Section 4.4 on page 59 talked about Flexible Alias Protection in the context of an interaction problem. In the present Chapter, Section 5.1 describes the background to APSs and reviews the related area of effects annotation. SOP is different to OOP in the way it approaches certain design problems. Following the background, we commence our analysis of the way SOP affects alias annotations, that is, the way the alias protection policy is realised by aliasing modes. These Sections form a part of our contribution to the thesis. Section 5.2 presents the strategies for selecting aliasing modes and the meaning of mode equivalence and inequivalence. Section 5.3 looks at the problems caused by ownership parameters. In Section 5.4 the reusability of subjects is explored; we analyse a subject which should be useful within a family of applications. Section 5.5 describes the properties of APSs that are useful for understanding subject-oriented interaction.

5.1 A Review of Alias Protection Systems

Encapsulation is one of the corner-stones of object-oriented programming. Well structured objects hide their implementation and present an abstract interface to clients. Non-trivial object-oriented programs consist of many collaborating objects that send each other messages which include objects in parameters and return values. Object-oriented languages pass objects by reference. An object's state is made up of the values of its field variables and the state of all objects it references. When an object is referred to using two or more names, it is said to be *aliased*. Aliases are created during variable and field assignment, and when an object is passed in a method argument or returned by another method call. Aliases pose a problem particularly in object-oriented programming because objects have persistent local state [57]. When object A receives a reference to object B, A may send messages to B which modify its state. Execution of a method call affects the state of the receiver object B and all other objects which reference the receiver. The states of objects that reference B change seemingly without the affected objects being accessed.

Visibility modifiers such as `private` are not an adequate protection from aliases. They protect variables from direct access but fail to disallow object exposure. One can easily write a getter method that reveals a `private` object. For example, consider class `Rectangle`, implemented using `Point`s as shown below:

```
class Rectangle {
  private Point topleft;
  private Point bottomright;
  Point getTopleft() { return topleft; }
  void setTopleft(Point topleft) { this.topleft = topleft; }
  Point getBottomright() { return bottomright; }
  void setBottomright(Point bottomright) { this.bottomright = bottomright; }
  void shiftBy(Point p) {
    topleft.shiftBy(p);
    bottomright.shiftBy(p);
  }
}
```

There are times when a `Rectangle` client needs to know and change its rectangle's geometry. Accessor methods are provided for this purpose. However, the client who gets the `Point` objects from a rectangle should use them with care because they are a part of the rectangle's mutable state. For instance, suppose a client, who is unaware of the way `Rectangle` is implemented, has two `Rectangle` objects `r1` and `r2`. At some moment he wishes to resize and move `r2` such that the `topleft` coordinate of `r2` becomes the same as that of `r1`. Then, he shifts `r1` to another location by calling method `void shiftBy(Point p)`:

```
1    Rectangle r1, r2;
2    Point p;
3    Point r1_tl = r1.getTopleft();
4    r2.setTopleft(r1_tl);
5    r1.shiftBy(p);
```

The unforeseen result will be also to shift `r2` by the amount denoted by `p`. As a consequence of this interaction, after line 4, `r1` and `r2` share the same `Point` object and not just the same top left coordinate. This is problematic to a client who is expecting only the top left coordinate to be shared. The problem is solved *post hoc* by cloning `Point` objects either in the implementation of `Point` or in the client. To the reuser of `Rectangle`, in order to use the `Rectangle` well, the clients must be aware of the way it is implemented.

Most solutions for tackling problems such as this have been in the area of alias prevention and control. Flexible Alias Protection uses aliasing modes to describe what a `Rectangle` object and the clients of rectangles can do with `Point` objects. When the `Rectangle` designer's intention is to keep `Point` objects hidden, he should use mode `rep`. Objects of mode `rep` cannot be disclosed outside their container, hence getter methods must return clones of `topleft` and `bottomright`.

For the most part programmers avoid aliasing problems, probably because objects mostly communicate in close groups [56]. The problem faced by APS designers is to create APSs that provide a degree of alias protection while supporting common object-oriented idioms. The emphasis is on practicality: discouraging bad practice without forbidding the designer from creating all sorts of object-oriented programs. Presently, we discuss in detail two recent proposals which fit this description best. Clarke et al's Ownership Types [23] derive from a formalisation of the core of Flexible Alias Protection. Aldrich et al's AliasJava [5] is an alias annotation system that emphasises the description of aliasing properties over strong encapsulation.

The issue of alias protection is related closely to effects annotation. An effects system describes the way that the state of a component may be accessed during program execution. This information is useful to programmers for reasoning about data dependencies between computations [106]. An effects system has the ability to infer the effects of a computation, to declare the permitted effects and to check that the inferred effects are permitted. Aliases introduce additional dependencies between computations making precise effects description more difficult. This Section also describes the role of effects systems in understanding subject interaction.

5.1.1 Ownership Types for Flexible Alias Protection

Ownership Types [23] have been proposed as a way of encapsulating objects used in the implementation of classes. At the core of Ownership Types is the concept of *contexts*. Every object owns a context and is owned by a context. The context an object owns is known as the object's *representation*. The context that owns the object is the object's *owner*. Contexts partition objects into nested groups, making it possible to talk about the inside and the outside of an object.

Program execution begins within a default system context world[1]. Any object created with owner context world can be referenced everywhere in the program. Objects of value types implicitly have world as owner, indicating that they can be aliased anywhere in the program. Every new object created comes with a new representation context by default. In essence, the only objects that can access this representation context are the object itself and other objects nested inside this representation context but only as long as they have been granted a permission. Thus, an object *o*'s representation context is not accessible from outside *o*. This property is known as *representation containment*.

An *ownership type* is the data type of an object extended with an angle-bracketed list of contexts. Non-value types are derived from classes. Class names are followed by a sequence of *ownership parameters*. When creating a new object, the object owner is passed as the first ownership parameter. The owner can be either world, the current representation context or any context from the ownership parameters of the class containing the expression.

For example, the `Queue` class with field variable `head` of type `Link` may be defined as follows:

```
class Queue<owner, data> {
```

[1] The notation of Ownership Types changed in later work [21]. The more recent notation is used here.

```
2      Link<this, data> head;
3      ...
4    }
```

`data` in line 1 is the name of the second ownership parameter. It is bound by the client instantiating `Queue`. The owner of `head` is the current representation context, i.e. this queue instance, as indicated by `this`. Class `Link` requires an additional ownership parameter. `data` in line 2 is bound to the same context which binds `data` in line 1.

Ownership Types are flexible because the owner need not be the object which does the instantiating. In fact, the object owner does not need to reference the objects it owns and can reference objects owned by others.

The system of ownership parameterisation allows clients to customise the ownership properties of objects. An object's owner context and ownership parameters are bound at instantiation and remain invariant until the object is destroyed. For example, with ownership parameters it is possible to create two `Queue` objects where in the first, the queue and data within are owned by the current representation context, and in the second, the queue is owned by the current representation context but the data inside is owned by world:

```
Queue<this, this> q1;
Queue<this, world> q2;
```

Variables `q1` and `q2` can never be aliases for the same objects because they have different types. Although both are have the same owner, the data sets referenced by each are disjoint.

Representation containment is best understood in terms of object graphs. A snapshot of an object-oriented program at runtime can be represented by an object graph. Objects are vertices and solid edges denote inter-object references. As the program executes, the graph evolves, with new vertices and edges added and old ones removed. At the root of the graph is context world representing the system in which the program runs. The sequence of solid edges between the root vertex world and any other vertex of interest forms a *path*. In a well formed object graph, all vertices are reachable by paths but there may be multiple paths for each object.

The dashed edges relate objects to their owner contexts. In a graph that satisfies the representation containment property, every path to an object must pass through that object's owner. Consider the graph in Figure 5-1. Its properties are:

- Object $o4$ is owned by $o2$ and $o4$ is in $o2$'s representation context.
- $o2$ is owned by $o1$. $o2$ is in $o1$'s representation context.
- Global objects with owner world are $o1$, $o3$, $o5$ and $o7$.
- $o1$ owns $o6$ but does not reference it. All paths to $o6$ must pass through its owner context $o1$.
- The world owned object $o3$ can alias $o6$ so long as all path to $o6$ pass $o6$'s owner. $o3$ cannot be aliased outside $o1$ because that would lead to the exposure of representation of $o1$ – the edge marked with a cross.

The program in Figure 5-2 demonstrates Ownership Types applied to the development of the `Queue` abstract data type. `Queue` is implemented as a collection of `Link` objects. The first element in the queue is referred to by `head` and the last element by `tail`.

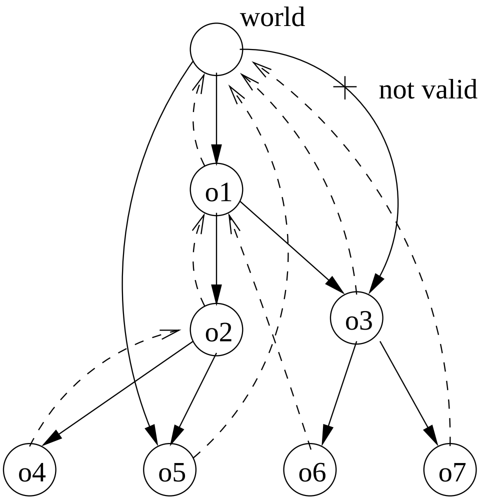

Figure 5-1: An object graph showing ownership arcs

Class names are followed by a sequence of ownership parameters where the first parameter is always `owner`. This refers to the owner context of the current instance. The `Queue` class also has ownership parameter `data` which refers to the owner of elements stored in the queue. In order for all elements to be treated as a single collection they must have the same ownership type, `Object<data>` in the example. In addition to `owner`, the `Link` class also has ownership parameter `d`. In collaboration with the queue, `d` gets bound to the owner of the data referred to by the link.

When links are created, their ownership parameters are bound. `Link` objects are owned by the queue, given by `this`, and the `Link`'s ownership parameter `d` is bound to the same context as Queue's `data`. Ownership Types uses the self reference `this` to state that links are the representation of their queue. Variables `head` and `tail` can be aliases for the same object because they both have the same ownership type.

The implementation of `Queue` requires links to refer to each other. This can happen only when all links have the same owner context. In the `Link` class we refer to the `next` link's owner parametrically with `owner`. The type of the `next` field is `Link<owner, d>` indicating that the owner of `next` is the same as the owner of this link, i.e. the queue which owns all the links. Figure 5-3 illustrates an object graph with ownership edges for a queue with 5 elements.

Object-oriented programming idioms such as Iterators [43] need short-term access to representation objects. In order for an efficient implementation to be possible, iterators must alias representations of the collections over which they iterate. The representation containment properties presented to now have enforced encapsulation fully making it difficult to create efficient iterators.

In an extension to Ownership Types, Clarke and Drossopoulou proposed support for *dynamic aliases* [21]. Dynamic aliases allow objects from the representation context temporarily to escape outside. The word *dynamic* refers to the way short-term aliases are implemented. In object-oriented languages, object references held in instance variables are stored on the heap while those held in a method's local variables are stored on the stack. All stack allocated variables are dynamic; they are destroyed when the method returns. Heap allocated references survive between method calls. In Ownership Types, external dynamic aliases to other objects' representation are allowed so long as the owner object is also in scope. Thus a representation object can be exposed but only in the

```
class Queue<owner, data> {

  Link<this, data> head = null;
  Link<this, data> tail = null;

  void put(Object<data> o) {
    Link<this, data> l = new Link<this, data>(o);
    if(head == null) {
      head = tail = l;
    } else {
      tail.next = l;
      tail = l;
    }
  }

  Object<data> get() {
    if(head == null) return null;
    Object<data> o = head.o;
    if(head == tail) {
      head = tail = null;
    } else {
      head = head.next;
    }
    return o;
  }
}

class Link<owner, d> {
    Object<d> o;
    Link<owner, d> next;
    Link(Object<d> o) { this.o = o; }
}
```

Figure 5-2: Program demonstrating Ownership Types

scope of its owner.

In the original Ownership Types [23], the ownership parameter bindings for an object could come only from the set of ownership parameters of the host class (including owner) and the set {this, world}. For dynamic aliases, the bindings can also come from any variable which is also in scope. For example, Figure 5-4 shows an iterator extension to the `Queue` class seen originally in Figure 5-2.

To obtain an iterator, the queue client calls `makeIterator`. The client gains sequential access to the data in the queue by repeatedly calling iterator's `next` method. In the following code fragment, first a queue is created and then an iterator is obtained from the queue:

```
Queue<this, d> q = new Queue<this, d>();
Iterator<q, d> it = q.makeIterator();
```

Note that the owner of the iterator is the queue. The owner context of `it` signifies that `it` is allowed to access the representation of `q` but only while `q` is in scope.

An alternative perspective on aliasing is taken by Aldrich et al in their work on AliasJava [5].

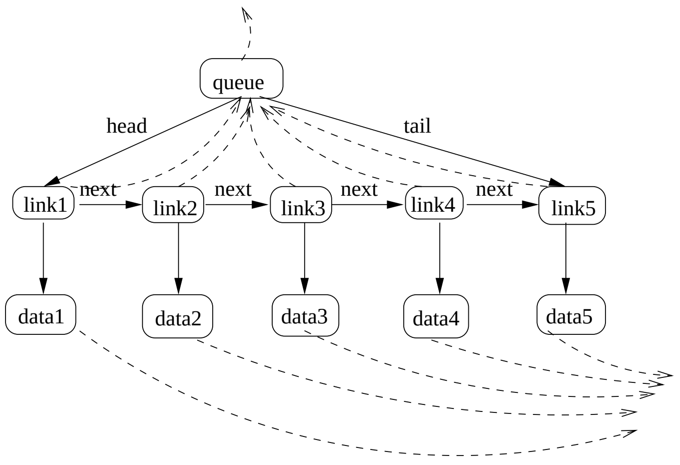

Figure 5-3: Ownership structure for a `Queue` instance

```
class Queue<owner, data> {
  ...
  Iterator<this, data> makeIterator() {
    return new Iterator<this, data>(head);
  }
}

class Iterator<owner, dt> {
  Link<owner, dt> current;
  Iterator(Link<owner, dt> first) { current = first; }
  boolean hasNext() { return current != null }
  Object<dt> next() {
    Object<dt> currentData = current.data;
    current = current.next;
    return currentData;
  }
}
```

Figure 5-4: `Iterator` extension to the `Queue` class with Ownership Types

5.1.2 AliasJava

AliasJava is an alias annotation system for specifying data sharing relationships in Java programs. It claims to capture several common forms of sharing that exist in object-oriented systems. The annotation system takes the form of a type system that lives alongside visibility modifiers and data types (like aliasing modes in FAP). There are five kinds of annotation found in AliasJava:

unique. A newly created object is considered *unique* – there is only one reference to it. After a `unique` variable is read, the source location must be set to another value before executing any other statement that may result in the original value being read a second time. `unique` values can be assigned to any other data sharing annotation but the inverse is not true as other annotations do not guarantee uniqueness.

owned. Objects that are confined to the scope of the enclosing object are considered *owned*. A reference to an `owned` object may be passed to another container if an explicit permission is granted. Unlike objects which are declared `private`, we cannot write a getter method to expose `owned` objects. `owned` objects may be aliased freely inside their container.

ownership parameters. An `owned` object may be shared with other objects by granting access at instantiation time with *ownership parameters*. An `owned` object can be passed several levels down the object hierarchy. When granting access, ownership is passed either directly by referring to the `owned` annotation, or by using any of the client's ownership parameters – these represent ownership properties associated with other objects. In classes which declare ownership parameters, objects with annotations taken from the set of ownership parameters are treated the same as `owned` objects.

shared. Objects which are intended to be aliased throughout the program are considered *shared*. Objects which are either global or do not have an owner are given the `shared` annotation.

lent. A `unique`, `shared` or `owned` object can be *lent* to another object for the duration of a method call. A `lent` object cannot be stored in an instance variable or returned from a method call. However, it can be passed on as a parameter to other operations as long as other operations also treat the reference as `lent`. A `unique` object can be aliased temporarily with `lent`. An `owned` object may be exposed externally with `lent`.

Figure 5-5 contains the `Queue` example with AliasJava annotations. The `head` and `tail` links are owned by their queue. `Iterator` instances are unique when created and can be bound to any mode in the client. The right to reference data objects is granted to `Queue` using ownership parameter `data`.

5.1.3 Understanding Aliasing Modes

Ownership Types and AliasJava approach encapsulation and aliasing from slightly different but intimately object-oriented perspectives. Both are concerned with the development of black-boxes which hide their implementation details. Ownership Types and AliasJava have the concept of object owner. Every object has exactly one owner that does not change over time (with the exception of unique references). Ownership parameterisation is used to separate the owner of the abstraction from the owners of elements stored within. The checking of types and modes is static, demonstrating the pragmatic nature of these approaches. Aldrich et al [5] go even further to suggest algorithms for inferring annotations in legacy software.

The containment properties of Ownership Types are stronger than those of AliasJava. In Ownership Types, the set of contexts forms a partial order. In order for object x to refer to object y, the representation of x must be inside the valid owners of y [22]. This property permits the creation of robust ownership structures where the representation of objects remains hidden behind the interface. This property has beed dubbed *deep ownership.* AliasJava instead enforces *shallow ownership* which guarantees that the owner of an object will not change. The stronger containment properties of Ownership Types are instrumental in making it possible to reason about the absence of aliases. By contrast, the unique aliasing mode of AliasJava ensures the absence of aliases but requires an unconventional programming style or explicit language support.

Beside Ownership Types and AliasJava, a number of other APSs have been proposed. Islands [56] and Balloon types [6] focus on full object encapsulation in which all representation objects are inaccessible outside the container. Objects can be moved in or out with unique references or using other techniques that prevent aliases escaping. The Universes approach [91] makes extensive use of

```
class QueueClient {
  owned Queue<owned> q = new Queue<owned>();
  void run() {
    owned Object o1 = new Object();
    owned Object o2 = new Object();
    q.put(o1);
    q.put(o2);
  }
}

class Queue<data> {

  owned Link<owned, data> head = null;
  owned Link<owned, data> tail = null;

  void put(data Object o) {
    owned Link<owned, data> l = new Link<owned, data>(o);
    if(head == null) {
      head = tail = l;
    } else {
      tail.next = l;
      tail = l;
    }
  }

  data Object get() {
    if(head == null) return null;
    data Object o = head.o;
    if(head == tail) {
      head = tail = null;
    } else {
      head = head.next;
    }
    return o;
  }

  unique Iterator<owned, data> makeIterator() {
    return new Iterator<owned, data>(head);
  }
}

class Link<queueAsOwner, dt> {
  dt Object o;
  queueAsOwner Link<queueAsOwner, dt> next;
  Link(dt Object o) { this.o = o; }
}

class Iterator<queueAsOwner, dt> {
  queueAsOwner Link<queueAsOwner, dt> current;
  Iterator(queueAsOwner Link<queueAsOwner, dt> first) { current = first; }
  boolean hasNext() { return current != null }
  dt Object next() {
    dt Object currentData = current.data;
    current = current.next;
    return currentData;
  }
}
```

Figure 5-5: `Queue` with AliasJava annotations

read-only references to specify a powerful APS. Boyapati et al [19] propose extending Ownership Types with dynamic aliases that are scoped to a group of related classes. Confined Types [127] focus on the security of objects. An object of a confined type is statically scoped within a package and any external references are disallowed. Confined Types are motivated by the need to prevent access by untrusted programs running in the same space. For instance, a Java applet can confine all objects of a certain type to the module denoted by the package which contains the applet. The realisation of Confined Types depends on *anonymous methods* that do not expose, manipulate or depend on the identity of the receiver object. In order to keep the identity of the confined objects hidden within the package, unconfined objects can only call anonymous methods.

Aliasing modes and properties can be roughly divided into those which describe the places where an object can be aliased and those which restrict access to the object's interface. Both AliasJava and the core of Ownership Types are concerned with the former: when an object acquires a reference, it has unrestricted access to its interface. Some APSs use interface restrictions to implement their aliasing policy. For instance, in Confined Types anonymous methods allow confined objects to be mutated while keeping their identity hidden. Flexible Alias Protection has the `arg` mode [92]. Representation objects (mode `rep`) which should be hidden from external clients can be passed to internal objects under mode `arg`. The client of an `arg` object only accesses the immutable interface. Messages sent to the immutable interface do not modify object state in a way that is visible. The client is protected from mutable state and the effect of an `arg` object on the client is constant.

Noble et al [92] partition modes into those which constrain external clients and those which constrain the implementation. Modes which constrain the external clients of an object are *upwardly restrictive*. Those which constrain the implementation are *downwardly restrictive*. Pragmatic considerations suggest that downwardly restrictive modes are preferable to upwardly restrictive ones. Components designed with the aid of an APS should be usable in existing systems but upward restrictions require other components to be aware of the APS used in this component's design. For example, anonymous methods are downwardly restrictive: the design of packages is constrained to disallow direct access to confined objects and their identities from objects outside the package. The `arg` mode downwardly constrains the receiver to using the immutable interface of the object in a parameter. In the Islands model [56], the `read` mode annotates variables to indicate read-only access to their state. The `read` mode is transitive: any reference obtained from the interface is also read-only. This mode is upwardly restrictive because it constrains the object's clients.

The modes introduced to now have annotated object references. Effects annotations are instead placed on methods. Effects are also of interest to us because, like aliasing modes, we believe that they can improve the understandability of subject interaction.

5.1.4 On Effects Annotations

Effects annotations describe the possible of method execution on state at signature level. Alias control is at the heart of effects systems in object-oriented programming. The choice of effects is driven by the goals of the effects system's designers. Greenhouse and Boylands's Object-Oriented Effects System [46] is motivated by the intention to perform semantics-preserving program transformations. Transformations often require the order of statements to be changed. Two computations do not interfere when one computation does not write state that is read or written by another. Therefore, Greenhouse and Boyland only track *read* and *write* effects. The FX language [79] also introduces the *alloc* effect which describes memory allocation and initialisation. The *alloc* effect adds to the descriptiveness of the effects system and proves useful for compiling programs that execute on parallel computers.

In Greenhouse and Boyland's system [46], effects are described on *regions* which are encapsulations of mutable state. Instance and static variables appear in regions which together form a hierarchy. Each variable has a default region and a special region called *All* is at the top of each class. Effects are specified on method declarations similarly to `throws` clauses in Java. The private state of an object is abstracted using the *unshared* annotation. Fields declared `unshared` have no aliases, that is, they are unique. Objects read from an `unshared` object and stored in local variables must not be revealed beyond the scope of the method call. Their effect is totally encapsulated within the object. In the terminology of regions, `unshared` objects appear in the regions of their enclosing container, and consequently their effects are hidden by the effects of the container as a whole. Unaliased objects, annotated with keyword `unique`, are used to insert and extract objects from containers. Static analysis ensures that parameters and return values are unaliased during the method call.

Trying to redesign the Queue example to use Greenhouse and Boyland's effects annotations shows some of the limitations of this approach. Suppose we try to use the linked list representation as before; in order to keep representation objects hidden, the program has to be redesigned to incorporate uniqueness. The main change involves the removal of field `tail` in order to make all links unaliased within the `Queue`. The `get()` method (which returns the last element) now has algorithmic complexity $O(n)$ over the elements in the queue because it must traverse all preceding elements to get to the tail. This compares unfavourably with the Ownership Types or Alias Java implementations which were $O(1)$. Other, more efficient implementations of the Queue abstraction are possible. However, it is important to note that the implementation must reflect the idiosyncrasies of Greenhouse and Boyland's effects system. In order to maintain performance, the preferred implementation cannot be used and another implementation is required.

In the absence of a `lent` mode as seen in AliasJava, the iterator is made integral to the `Queue` class. Method `resetIterator` now performs the function of creating a 'new' iterator. This implementation prohibits multiple simultaneous iterators from being created. Figure 5-6 shows the main design elements.

The JOE language (Java+Ownership+Effects) extends a Java-like language with Ownership Types and an effects system [21]. Instead of using regions for describing effects, JOE employs ownership contexts to describe *effect shapes*. As described earlier, the contexts in the scope of a class include this, world, owner and the other ownership parameters. There are two kinds of effect shapes. The *band* effect denotes the set of objects referenced by the instance variables of one object. The band is specified in relation to the current context this. For instance, suppose $\{p, \mathsf{this}, \mathsf{owner}, \mathsf{world}\}$ is the set of contexts in scope. The bands include:

- Each one of this, owner, world is a band.
- this.1 describes the band which has the present instance as owner.
- owner.1 is the same as this.
- this.2 describes the band which has this.1 as owner.
- $p.0$ is the same band as p; it denotes all objects referenced by the instance variables of p.

```
class Queue {
  region Data;
  unshared Link head = null;
  unshared int index;

  void put(Object o) reads nothing writes Data { ... }
  Object get() reads Data writes nothing { ... }
  void resetIterator() reads nothing writes nothing { ... }
  boolean hasNext() reads nothing writes nothing { ... }
  Object next() reads Data writes nothing { ... }
}

class Link {
  region Data;
  Object o in Data;
  unshared Link next;
  Link(Object o) reads nothing writes Data { ... }
}
```

Figure 5-6: Queue example extended with Greenhouse and Boylands effects annotations

The *under* effects denote a set of objects whose contexts are inside of and include a band. An under effect is written by wrapping a band within `under(..)`. For instance, the annotation under(this) denotes an effect which concerns all objects referenced by this and other objects in the representation context of this. The annotation under(p.1) denotes an effect which concerns all objects referenced by contexts represented by bands $p.i$ where $i \geq 1$.

In Figure 5-7 the Queue example is annotated with JOE effects. For instance, the `put` method declares the `writes under(this)` effect. The call to the `Link` constructor writes the newly created `Link` object as specified in the `writes this` annotation on the `Link` constructor. In `Queue` this is equivalent to `writes this.1`. The new object is assigned either to `tail` or to both `head` and `tail`, with effect `writes this`. Hence, the combined effect of the first statement in `put` is `writes under(this)`. All other statements in `put` either read or write the under effect denoted by `under(this)` (note that read is included in write).

5.1.5 Conclusion

APSs are a response to a call for better treatment of object aliases [57]. Uncontrolled aliasing has been shown to lead to programs which are hard to understand and maintain. The present Section has shown that APSs have in common the concept of an owned object. In Ownership Types the emphasis is on strong encapsulation. The core of Ownership Types is concerned with constraining object aliasing to a subset of objects in the program. In AliasJava the emphasis is on alias annotation. Instead of strong encapsulation, modes describe where the objects are aliased. Aliasing is managed by a combination of parameterisation, dynamic aliases to allow temporary access, and uniqueness which enables the object to change its owner. In all APSs, the aliasing annotations, modes or types work together to implement the containment policy. A mode is like a role that changes depending upon where the object is referenced.

We have described two computational effects systems. The effects systems are motivated by requirements for modular reasoning in order to perform program transformations or to enable parallelisation. While we expect that effects systems will be useful to understanding subject interaction,

```
class Queue<owner, data> {
  Link<this, data> head = null;
  Link<this, data> tail = null;
  void put(Object<data> o) writes under(this) {
    Link<this, data> l = new Link<this, data>(o);
    ...
  }
  Object<data> get() reads this.1 writes this { ... }
  Iterator<this, data> makeIterator() reads this writes this.1 { ... }
}

class Link<owner, d> {
  Object<d> o;
  Link<owner, d> next;
  Link(Object<owner, d> o) writes this { ... }
}

class Iterator<owner, dt> {
  Link<owner, dt> current;
  Iterator(Link<owner, dt> first) writes this { ... }
  boolean hasNext() { ... }
  Object<dt> next() reads this.1 writes this { ... }
}
```

Figure 5-7: Queue example extended with JOE effects annotations

compared to APSs, effects systems are not as universally useful to subject designers. Moreover, as exemplified by JOE, effects systems are built on top of alias protection systems. For these reasons, we choose to explore APSs as the means of improving the understandability of subject interaction. In the following Sections we look at the way the difference between subject-oriented and object-oriented programming impacts Alias Protection Systems.

5.2 The Impact of the Subject-Oriented Paradigm on APSs

Subject-Oriented Programming is based on a belief that in many cases there is no single intrinsic view of objects. Instead, the behaviour is determined by a combination of a number of possibly overlapping extrinsic perspectives. In the previous Chapter SOP was applied to a number of such examples. SOP decomposes software into subjects, and each subject uses classes to model the perspective assigned to it during decomposition. SOP introduces new concepts of correspondence and integration for synthesising the various views of abstractions and for reusing subjects.

Common abstract data types such as queues and hashtables have been used to demonstrate APSs. The object-oriented mechanisms of inheritance and delegation have proven well suited for conceptual modelling of ADT families and for reuse of ADTs. Stacks and hashtables do not exemplify SOP precisely because there exists a clear instrinsic understanding of the behaviour of these abstractions.

Multi-perspective development opens a question on how to deal with the different views that subjects have of object aliasing and ownership. Should subject designers agree on the aliasing policies or is there room for different views? What role does the composition specification play in determining the mode in the output subject? In the following, we apply a selection of ideas from APSs to the challenges posed by SOP examples.

5.2.1 The Car Mechanic Example

Suppose there exists a car hire company which leases vehicles out to clients. Each client is a driver who takes the vehicle for the period of the lease and returns it to the hire company when the lease expires, the car breaks down or the driver has an accident. Cars in good condition can be leased out again, however, cars which have been in accidents or are broken down must await the mechanic. The task of the mechanic is to restore cars to working condition. This is done by removing the working engine from crashed cars and using it to replace the broken engine in a car which has broken down. The example is decomposed into two subjects. The `HireCompany` subject contains the functionality associated with car leasing, driving, breaking down and crashing. The `Mechanic` subject contains the engine swapping functionality.

Without reference to any particular APS, this example invites a number of questions:

- Is the `Mechanic` subject an extension to the `HireCompany` base, or are these subjects peers? Aliasing modes may be treated differently in each case.
- Should the modes be equivalent, and if not, what is the meaning of different modes on corresponding elements?
- These subjects are being developed in concert. In view of the future composition, what are the criteria for mode selection?

Suppose AliasJava annotations are chosen to describe the subjects[2]. The annotated `HireCompany` and `Mechanic` subjects are shown in Figures 5-8 and 5-9 respectively. The composition specification that ties these subjects together is given by:

```
compose HireCompany, Mechanic;
mergeByName;
bracket ``Driver.rent'' with after Mechanic.afterHire;
```

Let us now look at the above questions in more detail.

5.2.2 Peer and Extension Subjects

Peer subjects are perspectives on to the same domain. They represent partial and potentially overlapping views which should not be contradictory. By contrast, an extension subject extends some base with optional or exceptional functionality. The extension may modify properties in a way that is contradictory with respect to the base view. But why does it matter if subjects are peers or related by evolution? – the strategy for determining the output mode may be tailored accordingly.

Looking at the Car Mechanic example we observe that the `HireCompany` subject is the main part of the application. Conceptually, it can be understood without reference to any other subject. The `Mechanic` subject represents an exceptional case that is meaningful only in relation to a base, e.g. the `HireCompany` subject. In the present case, it can be said that the `Mechanic` subject extends the `HireCompany` subject.

Having agreed that conceptually the `Mechanic` extends `HireCompany`, how does SOP specify when subjects are peers or related by extension? The composition specification has two purposes. The first is to specify the way subjects should be synthesised from the inputs. Secondly, the composition specification has a conceptual dimension which describes the way elements relate. Conceptually,

[2] we can equally well have chosen Ownership Types or another APS for this example.

```
class CarHireCo {
  owned Vector<owned> fleet;

  void addCar(unique Car c) {
    fleet.add(c);
  }

  void hireTo(shared Driver d) {
    lent Iterator it = fleet.iterator();
    while(it.hasNext()) {
      lent Car c = (lent Car)it.next();
      if(c.state == 0) {
d.rent(c);
return;
      }
    }
    // no working cars left to rent
  }


  void main() {
    shared CarHireCo f = new CarHireCo();
    f.addCar(new Car());
    f.addCar(new Car());
    shared Driver d1 = new Driver();
    shared Driver d2 = new Driver();
    f.hireTo(d1);
    f.hireTo(d2);
  }
}

class Car {
  // 0 = rentable, 1 == crashed but engine ok, 2 = broken engine
  shared int state;
  owned Engine e;
  void go() {
    e.start()
    ...
  }
}

class Engine {
  void start() { .. }
}

class Driver {
  void rent(lent Car c) {
    drive(c);
  }
  void drive(lent Car c) {
    c.go();
    // breakdown, crash or return the car unchanged
  }
}
```

Figure 5-8: The `HireCompany` subject with AliasJava annotations

```
class Mechanic {
  unique Engine spareEngine;
  shared Car brokenCar;
  void afterHire(shared Car c) {
    switch(c.state) {
      case 0: break;
      case 1:
        spare = c.extractEngine();
        if(brokenCar != null) doRepair();
        break;
      case 2:
        brokenCar = c;
        if(spareEngine != null) doRepair();
    }
  }
  void doRepair() {
    brokenCar.fitEngine(spareEngine);
    spareEngine = null;
    brokenCar.state = 0;
    brokenCar = null;
  }
}

class Car {
  // 0 = rentable, 1 = crashed but engine ok, 2 = broken engine
  owned int state;
  unique Engine e;
  unique Engine extractEngine() {
    unique Engine r = e;
    e = null;
    return r;
  }
  void fitEngine(unique Engine e) {
    this.e = e;
  }
}

class Engine { }
```

Figure 5-9: The `Mechanic` subject with AliasJava annotations

merge describes the joining of views with no implicit order or precedence and implies compatibility between aliasing modes. Compatibility need not mean equality although equality is the most straight forward measure of compatibility. Equality is meaningful for the modes of both Ownership Types and AliasJava. If, for example, all corresponding variables are declared `owned`, the output mode is also `owned`. The **override** rule specifies an ordering where the overriding element replaces the overridden element, e.g. the overriding operation replaces the overridden operation. Conceptually, the overriding element may totally change the aliasing policy in a way that is not compatible with the previous modes. So a parameter with a `unique` annotation may be overridden by a `shared` annotation from the signature of the overriding operation.

In SOP, the mechanics of composition at times clash with the conceptual model. One such case is when a composer has to use **override** to select one method body from two identical definitions while conceptually merging corresponding views. Looking at the composition specification for the Car Mechanic example, we observe that **mergeByName** is used to join subjects. The rule is necessary for describing the synthesis of subjects but the rule fails to convey the conceptual relationship between the concerns. The **bracket** relationship induces an order but only between classes and operations.

5.2.3 How to Treat the Modes of Corresponding Elements

The treatment of modes of corresponding elements may be related to the top level composition rule relating the subjects. One can require modes to be the same or introduce a level of variability that fits in with the SOP model of decentralised development.

There is no reason why different modes cannot be composed conceptually. Modes help to reason about the aliasing properties of objects in the output subject only when the output mode does not degenerate to unrestricted aliasing. For instance, in AliasJava, the `shared` mode conveys little useful information. Composition using **merge** inevitably increases object aliasing because each subject introduces behaviour which increases aliasing. If most objects become `shared` due to composition, aliasing annotation benefits that the APS brings will be lost.

Once again, consider the composition in the Car Mechanic example. In the `HireCompany` subject, cars are `owned` by `CarHireCo` and `lent` to the hirer for the duration of the method call. Engines are `owned` by cars with the driver having no direct access to the car's engine. In the `Mechanic` subject, cars are globally aliased objects as indicated by the `shared` annotation. To the mechanic each car has a single engine as indicated by `unique`. When the engine is replaced from a crashed car to one with the broken engine, uniqueness annotates with precision the effect of the swap. AliasJava does not allow an object to be simultaneously `owned` and `unique` as these are totally different aliasing properties. Likewise objects cannot be `shared` and `lent` at the same time. The combination of the properties of any of these modes leads to global aliasing as described by `shared`.

The problem lies in part with the choice of modes and in part with the way modes are selected from those available. An APS with a menu of finer grained modes than those offered by AliasJava may prevent all compositions of non-equivalent modes degenerating to `shared`. For example, if the APS allowed both the `CarHireCo` and the `Mechanic` to share the ownership of cars, some mode such as `co-owned` could replace `shared` when elements with `owned` and `lent` modes are composed. A `lent` object in one subject may become aliased as `owned` in another so long as no references are passed back to the original subject.

At times, more than one mode is capable of describing the actual aliasing properties. For example, a car's engine is `owned` by the car in `HireCompany`. Mode `unique` could have been used equally well without any changes to subject implementation. This topic is discussed in more detail in the next Subsection.

Consistency checking is a concern when composing different modes. In an APS like AliasJava, aliasing modes are used in concert. For instance, an `owned` object may be passed to another container using ownership parameters and later `lent` to other objects. Changing the mode in any one class will have a ripple effect on other classes that alias the object.

5.2.4 Criteria for Mode Selection

An APS like AliasJava makes it possible to use more than one mode in some cases. We have identified the following strategies for programmers to use when selecting modes:

- **Design the subject in order to conform with an APS.** Encapsulation is a cornerstone of object-oriented programming and one should design subjects with representation encapsulation in mind. APSs support encapsulation but with a certain programming style that may not fit with all applications of object-oriented programming. By following the APS's idiom strictly, the developer may be pressured into creating designs that do not satisfy other concerns. For example, in Figure 5-6 on page 81, the range of available modes affected efficiency: the program returns object clones when references would lead to a more efficient implementation.

- **Use the most constraining mode while still making it possible to create the same design as envisioned originally.** In this case, one selects the most constraining mode in order to describe the interaction. It suggests that one should not follow the idiomatic style of the APS but instead use the APS to annotate the latent relationships. This is the approach taken by any mode inference algorithm. Inference algorithms identify the most constraining aliasing mode.

- **Use the mode which is the most suitable conceptually.** In this case, conceptual modelling is identified as a priority. The choice of mode is influenced not by the domain of implementation but rather by the problem domain. Selection is an option when the available modes are not orthogonal and two or more modes can describe the interaction.

- **Use the strongest mode possible in view of composition.** One should bear in mind that subjects are often incomplete designs; subjects contribute to the behaviour of classes through composition. When subjects are designed as part of a collaborative effort, i.e. designed with a particular composition in mind, the choice of modes may be influenced by the subjects with which the present subject is going to be composed. For example, suppose that the mode most suitable conceptually is `owned`. However, composition introduces behaviour that produces external aliases, requiring a change to mode `shared`. Thus, mode selection is predicated on whether the problem domain is understood to be a single subject or a collection of subjects.

In conclusion, when composing elements with different modes, a fine grained APS is necessary in order to avoid all composition leading to a complete generalisation of properties, e.g. mode `shared`. Determination of the output mode may depend on both the input modes and the way the composition is specified. Mode compatibility is required for the **merge** composition strategy. For **override**, mode compatibility is not essential. However, the overriding of one mode by another requires some form of consistency checking to ensure that all mode changes are mutually consistent.

5.3 Problems with Ownership Parameters

Both AliasJava and Ownership Types use parameterisation to grant containers access to objects which are not part of their immediate representation. Ownership parameters are important for creating reusable classes. For example, the following two `Queue` objects have different aliasing properties (using Ownership Types):

```
Queue<this, this> s1;
Queue<this, world> s2;
```

Both queues are the immediate representation of the objects in which they are declared, but in `s1` the elements are owned by the current representation context while in `s2` the elements can be aliased anywhere.

Ownership parameterisation works very well for classes with an intrinsic view like `Queue` but not so well for classes created using SOP from a collection of overlapping domain views. Some objects and their owners are relevant only to a subset of composed subjects. These different views of classes translate to different ownership parameter lists. We present two examples of this problem.

First, consider `Finance` and `HR` (Human Resources) subjects in an office suite shown in Figure 5-10. Both subjects manipulate `Employee` objects. In `Finance`, employee expenses are reimbursed by sending `ExpensesSheet` objects to the `FinanceDept`. For this reason, during instantiation, `Employee` objects are parameterised by the owner of `FinanceDept`, giving rise to ownership parameter `fd`. In subject `HR`, the human resources department assigns line manangers to employees. In order for an employee to reference the line manager, the `Employee` class has ownership parameter `lm`. The program in Figure 5-10 uses Ownership Types annotations. Problems occur when one subject is responsible for instantiating `Employee`. Suppose subject `Finance` does the instantiating. Although both `Finance.Employee` and `HR.Employee` have ownership parameter lists of the same length, these parameters represent different concepts that may well denote different owners.

The second program performs graphical transformations on `Coordinate` objects. It consists of two subjects, shown in Figure 5-11, and uses AliasJava annotations. In the subject `AlgIn2D`, manipulations of coordinates are done in two dimensions, using only x and y values. In subject `AlgIn3D`, the algorithms apply to three dimensions, incorporating the z axis. Problems occur when a `Coordinate` instantiated in one subject is passed to another subject. For instance, both subjects declare class `X` with corresponding fields `someC`. Any object assigned to `someC` in one subject automatically becomes visible in another subject. When `AlgIn2D` creates a coordinate, it binds only the `n` and `m` parameters, and parameter `p` is unbound. It is not clear how the unbound parameter should be treated. When `AlgIn3D` creates a coordinate, it binds parameters `n`, `m` and `p`. If that coordinate is passed to `AlgIn2D`, the value bound to `p` will be lost. In order to restore the binding to `p` we must track its value while the object is aliased within `AlgIn2D`.

Naively, one may require that corresponding classes have ownership parameter lists that map one-to-one. Different names for corresponding ownership parameters should not pose a problem because the renaming facilities of SOP can be easily extended to include ownership parameters. In the first example, this entails introducing concepts from the Finance concern into the HR concern and vice versa. In the second example, this means introducing the z axis into all classes which refer to `Coordinate` objects in `AlgIn2D`. However, any such action violates a fundamental principle of Subject-Oriented Programming concerning clean separation of concerns. It would be inappropriate to have to include additional ownership parameters in order to satisfy some other concern.

```
subject Finance {
  class FinanceDept<owner> {
    void acceptExpenses(ExpensesSheet<owner> es) { ... }
  }
  class Employee<owner, fd> {
    FinanceDept<fd> finDept;
    Employee(FinanceDept<fd> finDept) { this.finDept = finDept; }
    void sendExpenses() {
      ExpensesSheet<fd> es = new ExpensesSheet<fd>(...);
      fd.acceptExpenses(es);
    }
  }
  class ExpensesSheet<owner> { ... }

  // example code using these definitions
  FinanceDept<q> finDept;
  Employee<p, q> emp = new Employee<p, q>(finDept);
  emp.sendExpenses();
}

subject HR {
  class HRDept<owner, lm> {
    Vector<this, lm> lineManagers;
    void addLineManager(LineManager<lm> lineMan) {
      lineManagers.add(lineMan);
    }
    void assignLineManager(Employee<owner> e) {
      e.setLineManager((LineManager<lm>)lineManagers.firstElement());
    }
  }
  class Employee<owner, lm> {
    LineManager<lm> lineMan;
    void setLineManager(LineManager<lm> lineMan) {
      this.lineMan = lineMan;
    }
  }
  class LineManager<owner> { ... }

  // example code using these definitions
  LineManager<g> lm;
  HRDept<f, g> hr;
  Employee<f, g> emp;
  hr.addLineManager(lm);
  hr.assignLineManager(emp);
}
```

Figure 5-10: Composition of subjects with incompatible ownership parameter lists

```
subject AlgIn2D {
  class Coordinate<n, m> {
    n int x;
    m int y;
  }

  class X<a, b> {
    owned Coordinate<a, b> someC;
  }
}

subject AlgIn3D {
  class Coordinate<n, m, p> {
    n int x;
    m int y;
    p int z;
  }

  class X<a, b> {
    owned Coordinate<a, b, a> someC;
  }
}
```

Figure 5-11: Composition of subjects with partially overlapping ownership parameter lists

Moreover, there is another way in which ownership parameters and SOP interfere. In SOP, corresponding classes can have different and non-corresponding superclasses. Each class has an ownership parameter list which the subclasses inherit. Therefore, the problem can occur also when classes with non-corresponding superclasses are composed.

As a reprieve, there is always at least some overlap in the ownership parameter lists. In AliasJava and Ownership Types, each object has an owner that is set at instantiation and does not change until the object is destroyed. In SOP programs, the owner is guaranteed to be bound for all objects in all subjects. Classes which have a single ownership parameter denoting the object owner can be composed without these problems.

In the following Subsections we look in greater detail at the problem of ownership parameters in SOP, starting from the development of abstract data types and moving on to the way ownership parameters contribute to creation of larger programs with SOP.

5.3.1 Ownership Parameters and ADTs

Common abstract data types like stacks, queues and hashtables are not candidates for decomposition along purely functional lines. These classes have clear intrinsic properties and we cannot improve their design by fragmenting further. ADTs can be associated with aspectual concerns such as synchronisation and persistence. SOP **bracket** relationship and other aspect-oriented technology can modularise these aspects. Aspectual concerns like synchronisation and persistence apply on a per instance basis; it should be possible to have two instances of the same basic ADT with different combinations of properties.

Most aspects affect behaviour but in a way that is transparent to the existing clients. So long as aspects do not introduce data which require parametric specification of ownership, all subjects have the same view of an ADT and all ownership parameters are bound no matter which subject instantiates such a class. In conclusion, ownership parameters do not pose a problem if all subjects that use parameterised components have the same view of their parametric properties. This is a good result for ADT reuse; ownership parameters support reusability by letting the client specify the aliasing properties of each instance.

But ADT reuse does not end with common components like stacks and hashtables. Programmers create arbitrarily complex components which use common ADTs in their implementation. For example, consider a `Spreadsheet` component. A client may want to create multiple spreadsheets with different ownership properties. But a Spreadsheet is a large and complex application supporting many features. SOP can simplify Spreadsheet development by enabling modular development of its features. By mixing and matching, the composer can tailor a Spreadsheet to the client by providing the required features. The Spreadsheet example would benefit from the modularisation potential of SOP *and* the customisation afforded by ownership parameters.

5.3.2 A Layered Architecture

Parnas [97] was among the first to suggest that modules should be arranged into a hierarchy, with modules higher up using modules lower down but not vice versa. An architecture that describes this layering is called a layered architecture. SOP enhances programs built of layers by supporting additional dimensions of decomposition. SOP can be used at each layer to separate concerns in the development of large components such as the `Spreadsheet` discussed above. Still larger applications built using SOP technology may use `Spreadsheet` objects in their implementation. And so on towards even larger components.

In order to achieve separation of concerns between layers, at each layer the component used in the implementation of a subject has to be a black-box. For example, `Spreadsheet` may support a degree of adaptation based on particular reusability requirements. The adaptations can be performed without looking inside the black-box. Functional changes which cannot be affected with parameters or other interface-level adaptations require the black-box to be opened up. The changes consist of one or more subjects and are applied using SOP composition rules.

In order to scale, an APS should help SOP to build large components by subject composition. At each layer:

- Aliasing modes should help composers avoid interaction problems by improving the understandability of interaction.
- Aliasing modes should protect the representation of the output component from access by external clients.
- Ownership parameters should facilitate client-end customisation of containers used in subject design.

5.3.3 The Two Roles of Ownership Parameters

A further problem with ownership parameters concerns the way they are used in design. Refer back to the Queue example annotated with Ownership Types in Figure 5-2 page 75. Suppose two subjects both define a `Link` class. Another way of defining this class is:

```
class Link<owner, l, d> {
  Object<d> o;
  Link<l, l, d> next;
  Link(Object<d> o) { this.o = o; }
}
```

In Figure 5-2, both the `Link` owner and the `next` link have ownership context `owner`. Above, the `Link` owner can be set separately from the owner of the `next` link.

Suppose two subjects both define the `Link` class but in different ways as shown here. These classes are conceptually composable but require significant modifications to subjects or extensive glue code. The problem is that in Figure 5-2, the `owner` parameter is reused when defining the owner of the `next` Link. This solution suffices for the implementation of the `Queue` class but makes the `Link` class less reusable than it can be. In a reuse setting, a client may require two links with different ownership types, e.g.:

```
Link<this, owner, world> link1;
Link<this, this, world> link2;
```

This example shows that while constructing class `Queue` for reuse, `Link` is treated as an implementation abstraction whose reusability does not concern the designer. Refer back to the criteria for mode selection in Section 5.2.4 on page 87. The original design of `Link` in Figure 5-2 used ownership parameters *to describe the conceptual relationship* between classes `Queue` and `Link`. In the context of `Queue`, this link and the `next` link have the same owner. The decision to use an additional ownership parameter is characteristic of an intention to achieve the best separation of concerns *by conforming with the APS*. The conceptual selection of modes was better at annotating existing usage but made `Link` less reusable.

This example shows also that ownership parameters play two roles in APSs:

- As the means of customising the ownership properties of ADTs, and
- As an implementation mechanism for passing access permissions.

The former is required because we want to reuse ADTs with different ownership properties. The latter is problematic within SOP because within each subject, classes define only those ownership parameters that are needed to realise the current concern. Defining additional parameters to satisfy other concerns is contrary to the spirit of SOP. Consequently, we believe that in subject-oriented programs, for class definitions that are distributed across subjects, ownership parameters are not the best way to pass access permissions. Some other system is required.

Finally, we also observe that ownership parameters pass access permissions but have little value as an annotational aid. An ownership parameter denotes that 'some other object owns this object' without making it clear which object it is. A concrete mode that pinpoints the actual owner or describes an aliasing policy is better for understanding subject composition because it conveys at a glance the extent of aliasing.

Client side customisation of ownership properties is necessary not only for ADTs but also for concerns implemented by subjects. This is the topic of the following Section.

5.4 Dealing with Incomplete Specifications

When decomposing a program into subjects, more often than not there are subconcerns which are common to more than one subject. In order not to duplicate code, a set of subjects delegate the

```
subject Composite {
  abstract class Component {
    abstract Object doAction();
  }

  abstract class Composite extends Component {
    Vector children;

    Object doAction() {
      Iterator it = children.iterator();
      while(it.hasNext()) {
        Component c = (Component)it.next();
        perChild(c);
      }
      return null;
    }
    abstract void perChild(Component c);
  }
}
```

Figure 5-12: Composite design pattern as a subject

implementation of the subconcern to just one subject. For instance, consider a banking application in which subjects implement the `OpenAccount` and `BalanceTransfer` concerns. Both subjects make use of operation `Account.deposit(..)`. In `OpenAccount`, it is called to set the initial balance when a new account is opened. In `BalanceTransfer`, it is called after the donor account has been withdrawn. Only one subject need implement `deposit(..)`.

In AliasJava and Ownership Types, ownership parameters make container classes more reusable by allowing clients to specify the ownership properties. In SOP, subjects are elements of reusable software that can implement patterns in a generic way. The subject designer may want the composer to specify the precise aliasing properties of a subject. Some subjects are made more reusable if their aliasing properties are not set in stone but allowed to vary based on the other subjects with which the reused subject is composed.

Reusability requirements and delegation are two reasons why an APS should have a way of specifying modes in some general way. Let us consider an example. A computer aided design (CAD) application creates pictures from primitive objects such as rectangles, lines and other pictures. The components making up a picture can be aliased by any other component. When a client needs to redraw the picture, `draw()` is called on all primitive elements and, recursively, on all pictures within. In an unrelated program, consider a file system consisting of files and directories. Directories contain files and other directories. Files or whole directories can be moved from one place to another. When a client calls the `size()` command on a file or a directory, the value associated with the size of the file or directory is calculated from the constituent parts.

Drawing in the CAD application and calculation of file system size are feature concerns that affect multiple classes in the base application. The behaviour associated with `draw()` and `size()` can be extracted into separate subjects, but it is possible to go still further. The concern that ties these subjects together is 'object hierarchy traversal'. The Composite design pattern [43] describes how to build object hierarchies consisting of primitive and composite objects. Primitive objects can be composed into more complex objects, which in turn can be composed, and so on recursively. Clients treat primitive and composite objects in the same way.

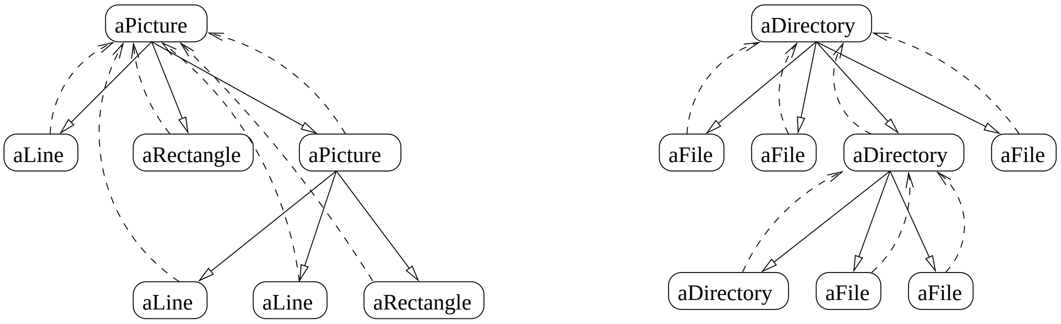

Figure 5-13: Ownership structure examples for the Draw concern in a CAD application (left) and for the Size concern in a File System application (right)

```
subject CADdraw {
  abstract class Component<owner> {
    abstract Object<world> draw();
  }
  class Picture<owner, topPic> extends Component<owner> {
    Object<world> draw() { /* to be composed with doAction() */ }
    void perChild(Component<topPic> c) {
      c.draw();
    }
  }
  class Line<owner> extends Component<owner> {
    Object<world> draw() { .. }
  }
  class Rectangle<owner> extends Component<owner> {
    Object<world> draw() { .. }
  }
}
```

Figure 5-14: `CADdraw` subject annotated with Ownership Types

With SOP we can implement the Composite pattern in a non-application specific way, and extend the generic pattern definition to create the subjects for doing `draw()` and `size()`. The `Composite` subject without aliasing modes is given in Figure 5-12. However, problems occur when we try to give an aliasing annotation or type to the objects referenced in the Composite pattern. The problem is illustrated by the object graphs in Figure 5-13. The solid edges denote references and dashed edges denote ownership relations. In the CAD application, all pictures, lines and rectangles are owned by the root picture. This structure allows for lines, rectangles and pictures to be shared between pictures at different levels. In the file system, the files and directories are owned by the directory that references them. The movement of a file from one directory to another changes the file's owner.

The different ownership structures required by these two problems translate into different aliasing modes in the design of the Composite pattern. The CAD drawing subject (Figure 5-14) requires the children of a composite object to be parameterised by the owner, which is the top-level picture. Annotated with Ownership Types, the `Composite` subject acquires the following modes:

```
abstract class Composite<owner, topPic> extends Component<owner> {
  Vector<this, topPic> children;
  ...
}
```

```
subject FileSystemSize {
  abstract class Component<owner> {
    abstract int size();
  }
  class Directory<owner> extend Component<owner> {
    int tempSize = 0;

    int size() { return tempSize; }

    void perChild(Component<this> c) {
      tempSize += c.size();
    }
  }

  class File<owner> extends Component<owner> {
    int fileSize;
    int size() { return fileSize; }
  }
}
```

Figure 5-15: `FileSystemSize` subject annotated with Ownership Types

In the file system (Figure 5-15), the directories and files inside another directory are owned by that directory, giving the following Ownership Type annotations:

```
abstract class Composite<owner> extends Component<owner> {
  Vector<this, this> children;
  ...
}
```

The problem occurs whether we use Ownership Types or AliasJava. In fact, the problem is more serious in AliasJava because the file system can employ the `unique` mode instead of ownership parameterisation. However, suppose that we stay with non-`unique` ownership. It should then be possible to design the `Composite` subject while allowing a degree of freedom when selecting the owner of the components referenced by the composite.

In related work, Clarke and Walker [26] discuss *composition patterns* which separate the design of cross-cutting requirements into reusable, extensible design models. Composition patterns are an extension to UML templates and composition semantics defining how both structural and behavioural design elements may be merged. Composition patterns use template parameters as placeholders for elements replaced by real elements in the composed design. The template parameters have constraints. For instance, when modularising a design pattern such as Observer [43] as a composition pattern, operations that are specific to pattern instantiation are specified as composition pattern parameters. The Observer pattern has already featured heavily in the examples of Chapter 3 on page 22. To recap, the Observer pattern describes a collaboration between a Publisher and a number of Subscribers. Subscribers dynamically register and deregister an interest in Publishers, so that when the Publisher's state changes all its registered Subscribers are notified of the change. The template parameters of this composition pattern are:

- The Publisher and Subscriber classes.
- The behaviour which constitutes a state change in the Publisher.
- The behaviour for performing updates in response to state change notifications.

- The behaviours for initiating the registration and deregistration of Subscriber objects with Publishers.

Each template parameter is typed as either a class or an operation, with operations having certain parameters of their own, e.g. this is the case with registration and deregistration behaviours. Clarke and Walker [27] have shown a mapping from composition patterns to Hyper/J and AspectJ. Template parameters are the client-specific elements of subject designs.

We believe that the reusability of composition patterns will be improved by the introduction of aliasing modes that are bound during composition. Aliasing modes should be in the list of template parameters. To best support the creation of reusable subjects and to enable delegation during design, it is necessary to specify the aspect of aliasing that should be parameterised, the constraints on parameterisation and a representation for parameters and constraints.

5.5 Towards an APS for Understanding Subject Interaction

Over the previous Sections this Chapter has described the main properties of Alias Protection Systems and the way the shift towards Subject-Oriented Programming affects those properties. The current Section brings these threads together in order to evoke the desirable properties for a subject-oriented APS. That is, to highlight those properties which best help subject composers to understand subject interaction.

According to Aldrich et al [5], a way of evaluating an Alias Protection System is by showing how annotations can help programmers answer questions that are difficult to answer in existing programs. We discuss some questions that are hard to answer presently in SOP programs and for each question describe:

1. The reason it is difficult to answer this question presently with SOP.
2. The APS properties that can help to answer the question.

- **Which objects may modify this object's state?** For example, suppose there is an interaction problem when a collection of subjects are composed. The problem is traced to an unwanted state change in an object.

    1. The cause of the state change is difficult to diagnose because almost any object in the system is potentially a client of the object whose state changes.
    2. Containment properties of Ownership Types are superior to the annotations of AliasJava because Ownership Types do not allow objects outside the owner to change the object's state. In AliasJava, only the owner is invariant but ownership can be granted to any object created subsequently. Ownership Types offers stronger representation containment guarantees. The objects that can modify another object depend also on dynamic aliases allowed by an APS.

- **How does one subject affect the objects of another subject?** Most complex behaviour is specified inside of and occurs within a subject. It is natural to think of a subject in terms of the object collaborations it implements. Objects passed as arguments to the collaboration from the outside and those created within appear to be owned by the subject. Of course, there is no actual subject ownership because subjects are merely packages.

   1. This question is difficult to answer at present with SOP because one is required to study operation implementations to understand their effect on shared state.
   2. A mode specified in the signature can help one understand the way a subject affects an object that is also referenced by another subject. For example, if an operation in subject $S1$ is merged with a `read-only` operation (with `read-only` transitive) from $S2$ then there are no unwanted state changes to objects 'owned' by $S1$ in calls to the merged operation.

- **What integration tests should be run on the output subjects created by this composition?** Before a component created by subject composition can be released it must be tested to check that it satisfies the requirements. Subjects interact when sharing control or data, hence compositions which integrate classes but never their members require no integration testing. Subjects that share control but not data, e.g. the Tracing concern in Chapter 3 on page 22, cannot be analysed through APSs because they do not pass object references over join points.

   1. Where subjects pass object references over join points, the modes of the shared data elements affect the range of tests required.
   2. Subjects that use only the immutable interface of shared objects, e.g. mode `arg` in Flexible Alias Protection, are not affected by state changes to those objects. When one subject depends on the mutable state of objects in another subject but is `read-only` on the objects it accesses, only the subject with `read-only` access needs to be tested for state changes. This is similar to the Spectators and Assistants model [28] defined for AspectJ. Aspects that only read but never modify objects at join points are *spectators*, the rest are *assistants*. In AspectJ an aspect is a class but a subject is a family of classes. The notion of spectator may be defined over the set of inter-subject join points by using aliasing modes that denote `read-only` access.

For understanding subject interaction, we believe that the initial challenge lies in getting control over object aliasing in a multi-subject environment in order that the first question above can be answered. The effect of one subject on another can be better understood only when the extent of aliasing is known.

5.6 Conclusion

This Chapter has reviewed Alias Protection Systems and discussed the challenges of developing an APS for SOP. An APS annotates the objects which depend on or modify the state of other objects. This property is useful to the subject composer because it can help to understand subject interactions and thus prevent interaction problems.

The differences in approach to software development between SOP and OOP impact the selection of aliasing modes. The main technical challenges are:

- Ownership parameters are problematic because each subject may need to reference objects from partially overlapping sets of owners.
- Ownership parameters are still required for creating traditional container classes.

- Certain subjects should be parameterisable by concrete aliasing modes of objects in other subjects.
- Subject-oriented composition may be used to create new black-box components.

In the following Chapters we present the Subjective Alias Protection System. SAPS addresses many of the above challenges: it is a fully fledged APS that also improves subject reusability.

Chapter 6

SAPS – Subject Design

The Subjective Alias Protection System is our proposal for improving reusability in a way that is also useful to the original developer of software. SAPS is an Alias Protection System for subject design and an annotation system for subject composition. The APS part of SAPS, also known as Subjective Ownership Types (SOT), helps subject designers to create well structured subjects that avoid problems which are known to result from bad uses of aliases. In this sense, SOT are of use to the original developer of software. SAPS is SOT plus subject-oriented composition rules. SOT annotate object aliasing at the points of subject interaction, helping the composer to understand the effect of subject interaction on state. Through explicit alias management, SAPS helps the subject integrator to prevent interaction problems.

The presentation of the SAPS is split over this and the following Chapter. The present Chapter describes the Subjective Ownership Types used in subject design. Chapter 7 discusses the composition of subjects annotated with Subjective Ownership Types.

Like Ownership Types [23], SOT enforced deep ownership properties. However, the traits of subject-orientation distinguish SOT from any object-oriented APS. For a number of reasons that will be explained inside this Chapter, SOT makes it possible to define two kinds of classes: composable classes and uncomposable classes. Subject definitions predominantly contain composable classes. We say that a subject has an ownership structure which is a model of the subject's ownership relationships. In composable classes the ownership structure is formalised by a system of explicit context naming. We will show that when separating concerns into subjects there are parts of the ownership structure which a subject either does not know or should not need to know about. For this purpose composable classes feature a new kind of context variable that has no equivalent in object-oriented programming. These so-called unknown contexts make it possible to specify subjects in a more reusable way than is possible with explicit contexts alone. Uncomposable classes are black-box abstractions that retain ownership parameters as the means for formalising their ownership structures.

This Chapter continues the presentation of our contribution to the thesis. Section 6.1 explains the principles of SOT and links them to the observations made in the previous Chapter concerning the differences in approaches to software construction between subject-oriented and object-oriented programming. Section 6.2 explains the principles of explicit context naming used in composable classes. Section 6.3 describes unknown contexts and their relationship to explicit contexts. Uncomposable classes and their relationship to composable classes are described in Section 6.4. Along the

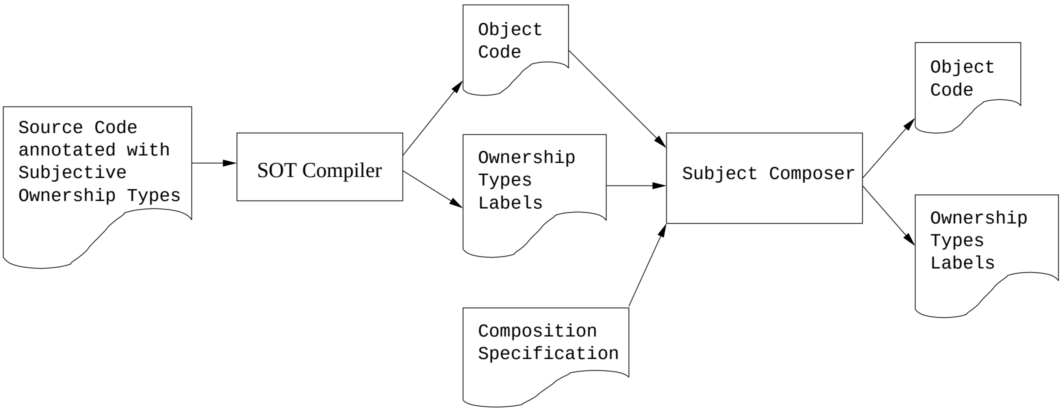

Figure 6-1: SAPS composition process

way we present the properties of correctness for Subjective Ownership Types. These include deep ownership checks. Section 6.5 concludes this Chapter.

6.1 Subjective Ownership Types and SAPS

This Section describes the principles of Subjective Ownership Types. In order to put SOT into context we will outline the SAPS process.

Subjective Ownership Types are part of a process that consists of two key stages. Figure 6-1 shows that subjects annotated with Subjective Ownership Types are individually compiled using the SOT Compiler (also known as the Subject Compiler). The SOT Compiler type checks the subject, generating Ownership Types Labels as one part of its output. For the second part, the SOT compiler strips out the Subjective Ownership Types and uses a standard programming language compiler to generate object code. The Ownership Types Label contains the stubs denoting the subject's composable elements, types and other attributes of constructs found in object code.

For the second stage, the Subject Composer takes as input object code, Ownership Types Labels and a composition specification. Initially, the composition specification is applied to Ownership Types Labels. The output label is generated if the Subjective Ownership Types found in the input labels permit composition. Finally, object code for the output subject is linked based on the composition specification and object code of the input subjects.

6.1.1 Deep Ownership

To create a useful Alias Protection System for Subject-Oriented Programming it is necessary to find the right balance between an APS for subject design and an annotation system for subject interaction. We believe that the right balance can be struck by devising a system that enforces deep ownership both for a single subject and across a set of subjects linked by a composition specification. That is, to extend SOP with additional types and composition rules for enforcing the same containment properties as Ownership Types [22].

In deep ownership only the object's owner and other trusted objects inside the owner can reference the object. This means that representation objects are totally hidden from external clients. In the setting of one subject, representation containment properties of Ownership Types better support modular reasoning at the object level than shallow ownership as used in AliasJava. AliasJava's constraints on `owned` objects do not prevent owned or representation objects from passing to external clients. The capability to reference objects owned by others can be passed using ownership parameters to an object with any other owner. Consequently, the effect on state is not constrained to the same degree as with deep ownership. The previous Chapter has argued that deep ownership is better suited to answering interaction questions about the effect of interactions on the state of objects. We anticipate that deep ownership will help to trace the source of an anomaly in many cases.

Object-oriented programming and SOP are different ways of addressing design challenges. In subject-oriented development, we have found the notation used by Ownership Types for communicating the ownership constraints inadequate for enforcing deep ownership. Instead, we propose a new system; one that is better suited to the SOP paradigm and its idioms. Before delving into the details of Subjective Ownership Types we summarise the motivation for its constituents.

6.1.2 The Origin of the Notation

The Subjective Alias Protection System is simultaneously inspired by a number of observations detailed in the previous Chapter. We review these in order to help explain the origin of our notation.

- **Observation 1: The Suitability of the Deep Ownership Model.**

  As described above, we want SOT to enforce deep ownership. We feel that it represents an agreeable compromise between a flexible alias protection model for subject design and an alias annotation system for subject interaction.

- **Observation 2: Customisation of Ownership Properties of ADTs.**

  APSs have been demonstrated in terms of ADTs such as one may find today in utility libraries. SOP will not be used to extend the definitions of these classes as for the most part inheritance and delegation are well suited to creating new types based on these abstractions. Common ADTs and types derived from them by inheritance or delegation will be used in subject definitions. Ownership Types and AliasJava have employed ownership parameterisation to enable clients to customise the aliasing properties of ADTs. The inherent flexibility of ownership parameterisation also should prove useful for subject design.

- **Observation 3: The Annotational Properties of Ownership Parameters.**

  SAPS is motivated by interaction problems. It should help programmers to steer clear of and subsequently detect anomalies. Consequently, in order to help the composer to understand the intra-subject relationships Subjective Ownership Types should annotate the ownership structure of subjects. Ownership parameters convey little information about the ownership structure; with the exception of the first parameter which denotes the object's owner, ownership parameters represent objects at possibly arbitrary points in the owner hierarchy. Parameterisation is useful for customising the ownership properties of ADTs (see Observation 2, above) but is less suited for annotating the ownership structure of subjects. The composer gains little useful information about the role an object plays in collaborations implemented by the subject when its owner is denoted parametrically.

- **Observation 4: Parameterisation is Characteristic of an Objective Perspective.**

A subject defines only those abstractions and functions that contribute to addressing its concern. The same principle should also extend to ownership concepts: a class in a subject should only have to define those ownership concepts that pertain to implementing its concern. We have shown in Section 5.3 on page 88 that subjectivity concepts interfere with ownership parameterisation. When instantiating, a subject does not and should not have access to all contexts which an object of that type may need to reference in all subjects. In many cases this results in a subject being unable to bind all ownership parameters declared in all other subjects. We believe that ownership parameterisation is characteristic of an objective perspective of software development where the client always sees the whole interface. To handle subjective perspectives, SAPS proposes an alternative to this permission passing mechanism.

- **Observation 5: Construction of New Components.**

  Subject-Oriented Programming enables the decomposition of programs by feature. Decomposition by feature applies not just to end-user programs but also to components built for reuse in component frameworks. These components are intended to be reused as black-boxes but may support certain anticipated extensions and adaptations. In order to extend the benefits of SOP to the design of components for use with existing frameworks, the restrictions on aliasing must be hidden within the output subject. If necessary, subjects may restrict each other but any aliasing modes emergent in the output subject should be downwardly restrictive.

- **Observation 6: Partially Specified Ownership Structures.**

  Subjects are often incomplete programs, delegating to other subjects certain implementation details. At other times, subjects implement collaborations with certain reusability requirements. Both cases require some form of genericity. With respect to ownership, Section 5.4 on page 92 identified an example where the ownership structure of one subject may be parameterised by other subjects.

The present Chapter is dedicated to explaining how the above observations have influenced Subjective Ownership Types. A combination of observations 2, 3, 4 and 5 have inspired us to formalise the separation of ADT utility library classes from those classes created as part of the subject definition. ADTs from utility libraries and other classes requiring customisation of aliasing properties are in the set of *uncomposable classes*. The vast majority of classes created as part of subject design are part of the set that we call *composable classes*. Interaction between objects of composable and uncomposable classes is possible in most cases.

Uncomposable classes do not participate in compositions; however, their instances can. ADTs are cohesive black boxes; we believe that for common ADTs, SOP cannot simplify their implementation. Based on observation 2, in order for clients to be able to customise the aliasing properties of ADTs, ownership parameterisation is used with these classes. Composable classes are defined using an alternative type system that does away with ownership parameters.

Observations 3 and 4 have inspired a new notation of *explicit context identifiers* for describing externally owned objects in composable classes. This notation replaces ownership parameterisation to enforce deep ownership. Based on observation 6, we introduce *unknown context identifiers*. These are used in composable classes for referring to objects whose ownership contexts are not known in the current subject. An unknown context identifier is part of another subject's design decision. They are bound when subjects are composed to form complete programs.

Observation 5 is concerned with our rules for subject composition. SAPS helps to hide the objects used in the implementation of components from the framework clients of the component. Subjects agree on the representation objects and each subject specifies representation objects using Subjective Ownership Types. In conjunction with subject composition rules that preserve types, the output subject continues to hide the common representation. SOT create no restrictions on the use of the output subject within a component framework. Observation 5 also concerns subject-oriented design in the large, i.e. the co-design of multiple subjects together. This aspect of Subjective Ownership Types is discussed in the following Chapters.

In this Chapter, we will present 12 properties that are required to ensure SOT correctness. The properties will be presented gradually and brought together at the end in order to describe checks for type correctness.

6.2 Explicit Context Identifiers

Explicit context identifiers are at the core of SOT and the design of composable classes. To make further discussion of explicit context identifiers more manageable, they will be referred to as `exps` (singular: `exp`). `exps` replace ownership parameters in composable classes. The `exp` annotations are used by the subject compiler to check that the subject satisfies the contraints of deep ownership.

The term *explicit* has been adopted because the ordering of contexts is explicit in the `exp` notation. To help explain the ordering of contexts and the origin of the notation we present object graphs more formally than they were described in the previous Chapter. A snapshot of an executing subject-oriented program can be represented as an object graph:

Definition: (Object Graph) An object graph is a finite directed graph whose ω_i vertices represent objects. References are denoted $\omega_1 \rightarrow \omega_2$. The root object ρ is a distinguished vertex with all objects reachable from root either directly or along a path formed by edges.

In order to support deep ownership all references to the object must come either from the object's owner or from other objects which are inside that owner. Graphically, this property can be understood in terms of paths between ρ and the object of interest. All paths from the root to the object must pass through the vertex representing the object's owner. In graph theory, the owner is the *immediate dominator* for the objects it owns. The immediate dominator comes from a set of *dominators* of an object:

Definition: (Dominator) For a given object graph, vertex ω_1 is a dominator for ω_2 if and only if every path from ρ to ω_2 includes ω_1. $\mathit{dominator}(\omega_2)$ is the set denoting all such dominators including ω_1.

A useful way of representing dominator information is in a tree, which in our case is called the *ownership tree*. The root vertex is ρ and each vertex dominates only its descendants in the tree [2]. Such a tree is induced by $\mathit{dominator}(\omega_2) \supseteq \mathit{dominator}(\omega_1)$:

Definition: (Ownership Tree) The ownership tree of an object graph is given by the partial

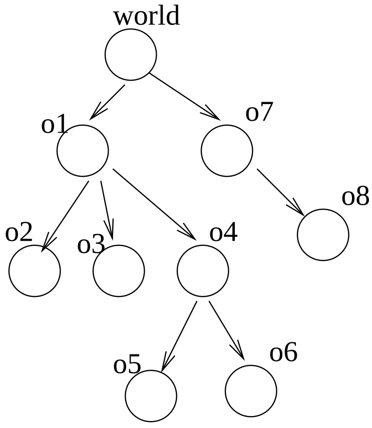

Figure 6-2: An Ownership Tree

order $\leq$ on objects:

$$\omega_2 \leq \omega_1 \text{ iff } \mathit{dominator}(\omega_2) \supseteq \mathit{dominator}(\omega_1)$$

ρ as the biggest element. Vertex ω_2 is the immediate dominator of ω_1 if and only if ω_2 is the least element of $\mathit{dominator}(\omega_1)$ not including ω_1. We write $\omega_1 < \omega_2$ when ω_2 is the immediate dominator of ω_1.

The owner of ω_i is defined as the immediate dominator of ω_i. As the program executes the object graph evolves. New objects and references are added, other objects and references are removed. The ownership tree co-evolves with changes to the object graph. Potter et al [102] observed that object graphs have an implicit domination structure. Although changes to the domination structure are inevitable, in well structured programs changes are limited. The purpose of an ownership type system is to formalise the domination structure in order to constrain the evolution of the object graphs, such that new references can be added only in a structured way.

Subjective Ownership Types formalise the ownership structure by numbering the dominators. The dominator set of ω_i forms a sequence with ω_i as the first, ω_j with $\omega_i < \omega_j$ as the second, until ρ as the last element. The elements of the sequence can be identified with their position in the sequence. Let $\mapsto$ denote the relationship between a sequence position index and the sequence element denoted by the index. Also let σ be a set of such tuples. The element identified by index i is denoted $\sigma(i)$ with $\sigma(1) = \omega_i$ as the first element. For example, consider the ownership tree in Figure 6-2. In the Figure, world owns objects $o1$ and $o7$; $o1$ owns $o2$, $o3$ and $o4$; $o4$ owns $o5$ and $o6$; object $o7$ owns $o8$. Object $o4$ is associated with sequence $\langle o4, o1, \mathsf{world}\rangle$ of dominators. Then for $o4$, σ is defined as:

$$\sigma = \{1 \mapsto o4, 2 \mapsto o1, 3 \mapsto \mathsf{world}\}$$

Object $o7$ is associated with sequence $\langle o7, \mathsf{world}\rangle$ of dominators. Its σ is defined as:

$$\sigma = \{1 \mapsto o7, 2 \mapsto \mathsf{world}\}$$

Now, suppose that Figure 6-2 is the intended ownership structure that we wish to formalise using types. We use the dominator indices $\mathit{dom}(\sigma) \cup \{0\}$ (the domain of σ including zero) to define

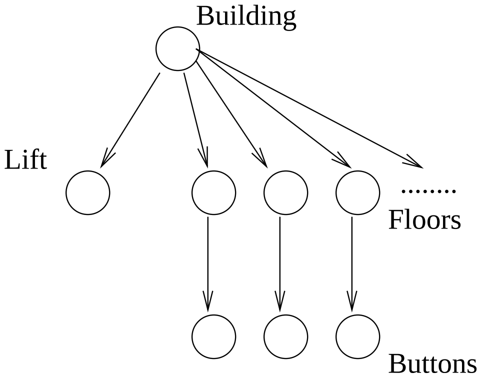

Figure 6-3: Ownership structure for subject `FloorPressButton`

explicit context identifiers for each object. With respect to $o4$, explicit context identifier 0 is shared by all objects that have $o4$ as owner, i.e. objects $o5$ and $o6$ in the representation context of $o4$. Explicit context identifier 1 refers to $o4$, the current context of interest. The owner of $o4$ is $o1$ – the immediate dominator of $o4$. We use explicit context identifier 2 to refer to the object that owns the current instance. Finally, explicit context identifier 3 refers to world, the ownership context of $o1$. From now on, we shall adopt notation exp_n to refer to explicit context identifier n. For example, for explicit context identifier 0 we will write exp_0, for explicit context identifier 1 we will write exp_1.

No two objects can have the same σ because exp_1 always refers to a different this. σ is based on perspective and an object can be referred to by two different exps. For example, world is referred to as exp_4 from $o5$ and exp_2 from $o7$.

Explicit context identifiers denote object owners in program texts. In the body of a composable class, a type is formed by extending the name of a composable class with an angle bracketed exp, world or an unknown context identifier. Like `this` in Ownership Types, exp_0 denotes objects in the current representation context. The owner context is always labelled using exp_1. The other explicit context identifiers refer to greater dominators in the ownership tree. We retain context identifier world for referring to global objects which can be aliased anywhere. Objects of value types have world as owner implicitly and do not require additional annotations. This scheme allows new objects and references to existing objects to be created in the current representation context, in the representation context of this object's owner, in the global context or in any other context that dominates this object.

6.2.1 exps in Action

To show the exp notation in action we use an example from a lift operation system. Suppose that one use case in the specification of a lift operation system describes the action of pressing a button at some floor. The floor on which the button is pressed should be added to the lift's list of floors to visit. One subject can modularise the implementation of this use case.

To understand the ownership structure of a subject, the subject designer can draw the intended ownership tree. Figure 6-3 shows that the lift and the floors within it are owned by the building. The buttons are owned by their respective floors.

Figure 6-4 shows that this structure can be formalised in subject code using exps. From line 2 we observe that in class `Building` the `lift` is owned by the `Building`. In line 3, the `floors`

```
class Building {
    Lift<0> lift;
    Floor<0,0>[] floors = new Floor<0,0>[10];
    void main() {
        for(int i = 0; i < 10; i++) floors[i].setLift(lift);
    }
}

class Lift {
    Floor<0,1>[] floorToVisit;
    int index = 0;
    void addFloorToVisit(Floor<1> f) {
        if(index < 10) floorToVisit[index++] = f;
    }
}

class Floor {
    Lift<1> lift;
    Button<0> button;
    void setLift(Lift<1> lift) {
        this.lift = lift;
        button.lift = lift;
        button.thisFloor = this;
    }
}

class Button {
    Floor<2> thisFloor;
    Lift<2> lift;
    void press() {
        lift.addFloorToVisit(thisFloor);
    }
}
```

Figure 6-4: Code for subject `FloorPressButton`

array and the elements within it are owned by this `Building` instance. In class `Lift`, `floorToVisit` array (line 10) is owned by this `Lift` instance but the `Floor` objects, passed as arguments in calls to `addFloorToVisit(..)` (lines 12–14) and stored in the vector, are owned by this `Lift`'s owner. When method `Button.press()` is called (lines 30–32), the floor on which the button is pressed is added to the lift's list of floors to visit. As seen in Figure 6-3, from the perspective of a button, `thisFloor` (line 28) and `lift` (line 29) both have owners given by $\texttt{exp}_2$.

6.2.2 Context Identifier Arithmetic

The numeric notation that we have adopted for `exps` is used both by programmers and the Subject Compiler to type check programs. SAPS programmers use data flow between objects to mentally check the validity of explicit context identifiers, i.e. to check that two references in different classes are mutually consistent if objects of those classes can ever reference the same object. References are generally passed between objects as arguments or return values in method calls, during field variable update and access. When references are passed as method parameters or in a field update, data flows out from source to target. For method call return values and field variable access, data flows the other way. To help programmers do the mental calculations we introduce functions Δ_1 and Δ_2 for outward and inward data flow respectively. The Subject Compiler relies on Δ_2 to type check expressions.

In the outward direction, two factors contribute to calculating the context identifier at the target:

- The `exp` representing the ownership context of the object to which the reference is passed, k.
- The `exp` or `exps` of the object whose reference is being passed, m.

The calculation of the context identifier at target is arithmetically simple, though it takes a knack to grasp the concept. To explain the calculation, first one must remember that `exps` number the dominators of `this` object. In order to prevent representation exposure, only the owner object m and other objects inside m are allowed to access the representation of m. The receiver context k must be inside the context of the argument object m, i.e. $k \leq m$. Conversely there is representation exposure: by definition of dominators, if an object with owner k also references an object with owner m then m is not a dominator for that object.

Whatever the value of k is at source, in the target the ownership context of the self reference is given by $\texttt{exp}_1$. There exists a difference of $m - k$ contexts between the argument and the receiver. Putting the difference in relation to the ownership context of the self reference in the target, the passed reference has context identifier $m - k + 1$ in the target object. This calculation is captured by Δ_1:

$$\Delta_1(k, m) \overset{\text{def}}{=} m - k + 1$$

Let us demonstrate Δ_1 in action using the code in Figure 6-4 on page 106. Consider the loop shown in line 5 which passes a `Lift` reference to each of the floors. In the body of the loop each floor object has type `Floor<0>`; thus $k = 0$. In class `Building`, the lift object whose reference is being passed has type `Lift<0>`; thus $m = 0$. By Δ_1 in class `Floor` the type of this lift object must be `Lift<1>`. As seen at line 20, this is indeed the case.

In the return direction, the `exp` value of the reference in the current context is obtained from field variable accesses or from the return value of method calls. Two factors contribute to calculate the context identifier at the destination:

- The exp representing the ownership context of the object from which the reference is obtained, k.
- The exp or exps in the source of the object whose reference is being obtained, n.

The ownership context of the self reference in the source object is given by exp_1. In order to prevent representation exposure, n must be greater than zero. Conversely, the object with owner n is the representation of n, and it may not be accessed by any object other than the one bound to `this`. There exists a difference of $n-1$ contexts between the owner of the object and the self reference. Putting the difference in relation to the the receiver's context k, the obtained reference has context identifier $n-1+k$. This calculation is captured by Δ_2:

$$\Delta_2(k,m) \overset{\text{def}}{=} k+n-1$$

To demonstrate Δ_2 turn once more to the code in Figure 6-4 on page 106. The field update expression at line 22 sets the `Button.lift` field. During type checking, the types of expressions on both sides of assigment must be equal. On the left hand side, the type of `button.lift` is determined by obtaining k and n. The type of `button` in the class of the expression is `Button<0>`; thus $k=0$. The type of the field in source class is `Lift<2>` as shown at line 29; thus $n=2$. By Δ_2, the type of expression `button.lift` at target is `Lift<1>` as expected.

6.2.3 Type Checking and Representation Containment

An important purpose of exps is to prevent representation exposure by enforcing deep ownership constraints at compile time. In deep ownership, in order for object x to reference y, x must be inside the set of valid owners of y [22]:

$$x \to y \Rightarrow x \leq \mathsf{owner}(y)$$

So far we have presented only exps. So the only valid types we can form at this time are those derived by substituting an exp for the owner context. To prevent representation exposure, substitution must satisfy two properties:

- **Property 1**. If exp_0 is in the type of a parameter, return value or field variable then this is the only valid receiver expression.
- **Property 2**. The context in the actual parameter must be as given by Δ_2.

In a static check, only this is guaranteed to be the owner of the representation. Any other expression may denote other objects whose representation context is different to this. Consequently, this is the only valid receiver when the type of a method parameter, a method return value or a field variable contains exp_0. Figure 6-5 shows examples of valid and invalid accesses. In line 5, field variable `e` has exp_0 in its type. However, it can be updated because `this` is the implicit receiver expression. The same is not true of line 6 because `d` may contain a reference to any `D` object (not just `this`. Lines 7 to 10 show valid and invalid expressions involving method calls.

For well-typed exressions the explicit context identifiers must be mutually consistent. The Subject Compiler uses Δ_2 to check types for mutual correctness. Figure 6-6 shows Δ_2 being applied to field access, field update and method call expressions. In line 6, a newly declared variable `e0` is initialised with an object returned by a field access. By Δ_2 the type of the field access expression is

```
class D {
  D<0> d;
  E<0> e;
  void foo() {
    e = new E<0>();    // Valid. Equivalent to 'this.e = new E<0>();'
    d.e = ...          // Invalid. e is in the representation of d
    e = getSomeE();    // Valid. Implicit receiver 'this' in rhs expression
    .. = d.getSomeE(); // Invalid. getSomeE returns a representation object
    setSomeE(e);       // Valid. The expected and actual parameter type is E<0>
    d.setSomeE(e);     // Invalid. Only 'this' can access this method

  }
  E<0> getSomeE() { .. }
  void setSomeE(E<0> someE) { .. }
}

class E { }
```

Figure 6-5: Static visibility check exemplified.

```
class C {
  D<0> d0;
  D<1> d1;
  D<2> d2;
  main() {
    E<0> e0 = d0.e1;
    d1.e1 = new E<1>();
    E<2> e2 = d2.getE1();
    d1.setE1(new E<1>());
  }
}

class D {
  E<1> e1;
  E<1> getE1() { return e1; }
  void setE1(E<1> e1) { this.e1 = e1; }
}

class E { }
```

Figure 6-6: Δ_2 applied to different kinds of expression.

```
class LibraryClass {
  T libfun(V v) { .. }
  void register(Object c) { /* e.g. for async notification */ }
}

subject S {
  class C {
    D<O> d;
    void foo() {
      LibraryClass<world> lc;
      V<world> v;
      T<world> t = lc.libfun(v);
      d.v = v;               <- ok to pass world owned objects
      // lc.register(this);  <- representation exposure!
    }
  }
  class D {
    V<world> v;
  }
}
```

Figure 6-7: Using world owned objects.

E<0>. Line 7 contains a field update expression. By Δ_2 the type of the expression on the left hand side of assignment is E<1>. In line 8, a newly declared variable e2 is initialised with object returned by a method call. The exp in the return type is given by $\Delta_2(2, 1) = 2$. Finally, line 9 shows a method call that passes a reference into d1. Here Δ_2 is used to check the type of the actual parameter. exp$_1$ is the context identifier of the owner in the receiver expression. exp$_1$ is also the declared context identifier in class D. Thus, $\Delta_2(1, 1) = 1$ is the expected context identifier in the type of the actual parameter.

Function Δ_2 is used also when this is the receiver expression. For some class D, this has type D<1> and Δ_2 is applied conventionally to expressions involving this (whether used implicitly or explicitly) to determine the expected context identifier in the type of the expression.

world versus exps

world denotes the global context but by definition of exps there is always one exp that denotes the global context also. The decision to keep world for objects of non-value types is a pragmatic one. It enables SAPS programs to interact with existing libraries by treating as world-owned all objects created from library classes or obtained from the interfaces of library objects. Implementations of existing libraries are unaware of ownership concepts and deep ownership, so representation objects passed to objects of library classes may be exposed.

Despite the overlap with exps, we class world as representing a context that is external to all other contexts. This decision enables objects of library classes to be referenced and passed freely within SAPS programs but also restricts world owned objects from referencing other objects whose context is specified by an exp. The Subject Compiler must ensure that only world owned objects are received from and passed to the interface of a world owned object. Assignment between exp owned and world owned objects is not allowed. This leads to our third property for SOT correctness:

- **Property 3**. world is external to all contexts denoted by exps.

```
1    subject S {
2      class A {
3        B<0> b;
4        C<1> c;
5        void foo() {
6          c = b.c;
7        }
8      }
9      class B {
10       C<2> c;
11     }
12   }
13
14   main() {
15     A<world> a;
16     B<world> b;
17   }
```

Figure 6-8: Example showing out of range exps

Figure 6-7 shows the use of a LibraryClass object within a subject. In line 10, lc is declared with owner world. This enables objects to be obtained from and safely passed to the lc object (line 12), but prevents us from being able to register self with the library (line 14).

Out of range exps

References whose types feature exp$_0$ and exp$_1$ are always well formed because they always represent objects that are known to exist. However, it is possible to use the other exps to create references to non-existent contexts, e.g. type T<999> can be created from class T. Such types are problematic because they are meaningless, i.e. the exp in the type does not refer to a real dominator. exps must prevent representation exposure but out of range exps do not cause representation exposure as they always refer to objects outside the current representation context. Consequently, we do not check for out of range exps.

Figure 6-8 shows an example of out of range exps. Lines 15 and 16 show two objects being created in the global context. In this program all exps in class A refer to contexts that exist: exp$_0$ is the representation context of main's a; exp$_1$ refers to the ownership context of main's a, i.e. the global context otherwise referred to by world. In class B, exp$_2$ refers to the context that owns this B instance. There are two possible bindings for exp$_2$: through object a or through object b in main. Through a, exp$_2$ binds to the global context. However, through b exp$_2$ denotes the owner of the global context, i.e. the owner of context world. But world is the root of the ownership tree and has no dominators. Hence, exp$_2$ is an out of range context identifier.

Attempting to access an object with an out of range context is a conceptual error although not a type error. In the example of Figure 6-8, most probably the designer of subject S intends for objects of type B to be used as part of a collaboration with objects of type A and clients should not create instances of B directly. Judicious use of visibility modifiers should help prevent unauthorised access. By declaring class B private, it is possible to disallow the instantiation of B outside S.

6.3 Unknown Context Identifiers

For many applications of subjects, `exps` make it possible to refer directly to all the contexts an object needs to reference, e.g. Figure 6-3 on page 105. However, there exist concerns implemented by subjects which need to refer to contexts which are not known in advance for one of two reasons:

- The context is external to the ownership structure of the subject, such as when a collaboration implemented by the subject refers to data objects external to the set of collaboration participants.
- The decision about the context should be delegated to another subject. Subjects implement concerns that cross-cut the structure of other subjects. In order for a subject to adapt to the ownership structure of another subject the ownership context has to be stated more generally than is possible with `exps`.

Unknown context identifiers address both issues. In order to condense the presentation we shorten 'unknown context identifer' to `unk` (plural: `unks`). `unks` abstract explicit context identifiers in the sense that each `unk` represents one `exp` per class. They are context variables with subject scope but class level binding.

To motivate `unks`, consider the development of the SuperTax concern. A super tax is a flat levy on taxable objects in a tax declaration. It is envisaged that this concern will apply to a number of tax departments. During analysis the internal organisation of tax departments is translated to ownership structures within the design. Some of the requirements and their ownership interpretations (in an *italicised font*) are given below:

- The declaration artifact and the goods specified within belong to a tax declaration. *TaxCalculation owns the declaration and the Goods objects.*
- The tax declaration is part of an overall tax calculation for a trader. *TaxCalculation owns the declaration.*
- The tax declaration is the responsibility of the tax assessor who works with all aspects of trader's tax liabilities. *TaxAssessor owns TaxCalculation objects and the associated declarations.*
- The tax declaration belongs to the Customs and Excise office which employs the tax assessors. *CustomsAndExciseOffice owns the TaxAssessors.*
- The goods listed in the tax declaration belong to the Customs and Excise office. *CustomsAndExciseOffice owns the Goods objects in the declaration.*

In realising the SuperTax concern we create the `SuperTax` subject. This subject classifies `Goods` as either `Taxable` or `NonTaxable`, levying a flat duty of 200 on every `Taxable` object. When a client calls `calculateTax(..)` with a `declaration` array as argument, the duty is calculated and stored in the `TaxCalculation`'s `amount` field. Figure 6-9 shows the key parts of the implementation. It adopts an ownership structure where the `declaration` and the `Goods` objects are owned by the `TaxCalculation`. The same subject with an alternative ownership structure is shown in Figure 6-10. Here the `declaration` is owned by the `TaxAssessor` object and the `Goods` objects are owned by the

```
subject SuperTax {
  class TaxAssessor {
    TaxCalculation<0> tc;
  }
  class TaxCalculation {
    int amount;
    void calculateTax(Goods<0, 0>[] declaration) {
      for(int i = 0; i < declaration.length; i++) {
        amount += declaration[i].calculateSuperTax();
      }
    }
  }
  abstract class Goods {
    int calculateSuperTax();
  }
  abstract class Taxable extends Goods {
    int calculateSuperTax() { return 200; }
  }
  abstract class NonTaxable extends Goods {
    int calculateSuperTax() { return 0; }
  }
}
```

Figure 6-9: `SuperTax` subject with examplar ownership structure 1.

```
subject SuperTax {
  class CustomsAndExciseOffice {
    TaxAssessor<0> ta;
  }
  class TaxAssessor {
    TaxCalculation<0> tc;
  }
  class TaxCalculation {
    int amount;
    void calculateTax(Goods<1, 2>[] declaration) {
      for(int i = 0; i < declaration.length; i++) {
        amount += declaration[i].calculateSuperTax();
      }
    }
  }
  ...
}
```

Figure 6-10: `SuperTax` subject with examplar ownership structure 2.

```
subject SuperTax {
  unk k, m;
  ucirc k <= m;
  class TaxCalculation {
    int amount;
    void calculateTax(Goods<k, m>[] declaration) {
      for(int i = 0; i < declaration.length; i++) {
        amount += declaration[i].calculateSuperTax();
      }
    }
  }
  abstract class Goods {
    int calculateSuperTax();
  }
  abstract class Taxable extends Goods {
    int calculateSuperTax() { return 200; }
  }
  abstract class NonTaxable extends Goods {
    int calculateSuperTax() { return 0; }
  }
}
```

Figure 6-11: SuperTax subject implemented using unks

CustomsAndExciseOffice object. The two solutions differ only in terms of ownership structures as expressed by the exps in the types.

unks enable the ownership structure to be stated more generally than is possible with exps. A single program in Figure 6-11 can replace the two programs shown in Figures 6-9 and 6-10. In the SuperTax subject unks facilitate two kinds of ownership variations: the owner of the declaration array and the owner of the Goods referenced in the declaration. The variation is introduced through unks k and m (referred to as unk$_k$ and unk$_m$ henceforth). unk$_k$ denotes the owner of the declaration and unk$_m$ denotes the owner of the Goods. Figure 6-11 shows the SuperTax subject with exps replaced by the new unks.

An unk is a context variable that binds to one exp per class. Figures 6-9 and 6-10 presented two possible bindings for unk$_k$ and unk$_m$. For Figure 6-9, in class TaxCalculation unk$_k$ binds to exp$_0$ and unk$_m$ binds to exp$_0$. For Figure 6-10, in class TaxCalculation unk$_k$ binds to exp$_1$ and unk$_m$ binds to exp$_2$. Now, thanks to unks, a single SuperTax subject can replace a family of subjects that vary purely in terms of the ownership structure formalised by their explicit context identifiers. Figure 6-11 also contains a ucirc declaration. This will be explained once we have described the unk concept in greater detail.

6.3.1 Understanding unks

unks are characterised by the following list of properties:

- As seen in the SuperTax example, an unk generalises an explicit context identifier in a class. It represents a choice of explicit context identifiers which enables a subject to adapt to a greater number of ownership structures.

- An unk gets bound by composition. To avoid confusion, we use *resolution* to refer to 'unk binding by subject composition', reserving *binding* for function and ownership parameters.

```
subject S {
  unk k;
  class A {
    B<0> b;
    void passC(C<k> c) {
      b.doStuffToC(c);
    }
  }
  class B where 1 <= k {
    void doStuffToC(C<k> c) { ... }
  }
  class C { }
}
```

Figure 6-12: An example involving an unknown context identifier.

- unks are scoped at subject level. Conceptually an unk represents a single unknown context, that is, one object in the collaboration defined in the subject.
- unks resolve on a per class basis. Different resolutions in different classes are inevitable: two objects at different ownership tree depths will use different exps to refer to a common object.

The global scope and class level resolution mean that an unk is a set of tuples of the form $\langle$ClassName, unk$\rangle$, where each tuple maps onto a set of exps. This is illustrated by the example in Figure 6-12. A k-owned object c is passed to an A object which collaborates with a B object by doing something with c.

unk$_k$ resolves to an exp in A, B but not C because C does not contain any expressions whose type contains unk$_k$. unk$_k$ can resolve to any exps in classes A and B so long as the resolutions are mutually consistent such that the subject type checks successfully. Valid resolutions are shown in the following table below. Any row represents a valid resolution. The ellipsis indicates that other values matching this pattern are also acceptable.

A	B
0	1
1	2
2	3
...	...

By using Δ_1 the reader can mentally check that any resolution for A will produce the corresponding resolution for B. In this Chapter, the focus is on exp–unk interaction within subjects. Subject composition and unk resolution are presented in the next Chapter.

6.3.2 unk Resolution Constraints

In a few cases an unk can resolve to any exp; however, mostly the set of values to which an unk can resolve is constrained by other relationships. This introduces the notion of an *unknown context identifier resolution constraint*, formalised in SAPS by ucirc[1] declarations. A ucirc is a predicate that expresses a constraint on unk resolution.

[1] pronounced "you-serk"

```
subject S {
  unk k;
  unk m;
  class A where 1 <= k, k <= 1 { ... }
  class B where m <= 1, 1 <= k, 2 <= m { ... }
}
```

Figure 6-13: unks and resolution sets.

There are two kinds of `ucirc` appearing in code both of which have made an appearance in the preceding examples:

- Subject level ucirc declarations express an inter unk constraint. These appear at subject level because they express a context ordering that should hold for all classes in which the two unks appear together. For example, `ucirc k <= m` expresses that the context denoted by unk$_k$ is always inside the context denoted by unk$_m$.
- Class level ucircs are specified in `where` clauses of classes [30]. They express a constraint on unk resolution applicable to that class and classes derived from it. For example, `class A where k <= 2` expresses that in class `A` unk$_k$ may resolve to exp$_0$, exp$_1$ or exp$_2$. Also, `class B where 1 <= m, m <= 2` expresses that in class `B` unk$_m$ may resolve to exp$_1$ or exp$_2$.

The Subject Compiler performs checks against ucircs at both class and subject level. But before the body of a class is checked, the ucircs themselves are checked for consistency.

ucirc Consistency Checking

A consistency check ensures that ucircs are well formed and offer meaningful constrains on unk resolution. An unk implies a choice of exps and it is misleading to use an unk when ucircs imply one or no exps. We use the term *resolution set* to refer to the set of exps to which an unk may resolve in a class as determined by the ucircs. An empty resolution set indicates a type error. The unk should be replaced by the exp when its resolution set is a singleton. In Figure 6-13, for class `A`, unk$_k$ should be replaced by exp$_1$. For class `B`, unk$_m$ has an empty resolution set but the resolution set for unk$_k$ is valid.

The ucircs are checked per class, but first the class level ucircs are extended with those at subject level. In order to be consistent the ucircs of each class should satisfy the following conditions:

- All unks appearing in ucircs are declared.
- There are no cycles in ucirc declarations.
- The combination of class and subject level ucircs produces valid resolution sets.

Looking at cycles, consider $r = \{(u \leq v), (v \leq w), (w \leq u)\}$. This set has a cycle. Clearly, $u = v = w$ is the only resolution which satisfies all constraints. This set of contexts and the associated constraints should be replaced by a single unk. We require cycles to be removed from designs because they may cause confusion. That is, at a glance unks and ucircs lead the subject to reuser to believe that there exists a choice contexts when there is no such choice.

When subtypes are introduced, the resolution constraints of the subtype must be valid subconstraints of the supertype. An unk in the subclass may never resolve to a value that is outside the resolution set specified in the superclass. The subclass may strengthen the ucircs defined in the superclass but it may never weaken them. That is, the superclass constraints must imply the constraints of the subclass. Consistency checking of ucircs is formalised by the following property:

- **Property 4**. The resolution sets of all unks must be well formed.

6.3.3 Checking Classes Against ucircs

ucircs can be understood as sanity conditions for unks. They have three responsibilities:

- To check that types are well formed.
- To communicate valid resolution sets.
- To type check expressions involving objects whose types contain unks.

The checking of types derived from uncomposable classes will be described later in this Chapter. However, we are in a position to describe arrays.

Checking Arrays

In general, the contents of an array should be accessible everywhere the array is accessible. For one dimensional arrays two context identifiers are required: the owner of the array must be inside the owner of the elements of the array. In arrays of higher dimension, each earlier dimension must be inside the later ones. The constraints on context identifier substitution for arrays introduces our next property for SOT correctness:

- **Property 5**. The context substitutions for ownership parameters of arrays must allow access to lower dimensions wherever higher dimensions are accessible.

Recall the `SuperTax` example of Figure 6-11 on page 114. The `Goods` objects in the `declaration` should be accessible wherever the `declaration` object is accessible. From left to right, the two context identifiers in the type of `declaration` denote the array and the elements. The ucirc relating unk_k and unk_m guarantees that the `declaration` array will be accessible only where its elements are accessible.

Communicating Valid Resolution Sets

Property 1 ensures that objects in the representation context cannot be accessed externally. Therefore, if an unk-owned class member *is* accessed externally, then exp_0 is not in the realisation set of that unk. In order to communicate valid resolution sets to subject reusers, SOT requires ucircs for all unk-owned class members with external clients. Declarative completeness of subjects makes it possible to validate all members for external access.

Recall the example in Figure 6-12 on page 115. This example features a `where` clause on class `B` that constrains the resolution set of unk_k. This ucirc is required in order to prevent representation exposure by expression `b.doStuffToC(c)` in the body of method `A.passC(..)`. In order to prevent representation exposure in the body of class `A`, class `B` requires a resolution constraint.

In order to understand the motivation for this constraint, consider the effect of unk_k resolving to exp_0 in `B`. Then expression `b.doStuffToC(c)` would lead to representation exposure. Thus `ucirc 1 <= k` is a sanity check that constrains the resolution set of unk_k in `B` order to prevent representation exposure during composition.

```
subject S {
  unk k;
  unk m, n;
  ucirc m <= n;
  class U where 2 <= k {
    T<2> t;
    V<k> v;
    W<m> w;
    X<n> x;
    void foo() { t.bar(v); }
    void fee() { w.x = x; }
  }
  class T where 1 <= k {
    void bar(V<k> v) { ... }
  }
  class W where 1 <= n {
    X<n> x;
  }
  class V { ... }
  class X { ... }
}
```

Figure 6-14: Checking expressions involving unks.

Checking Expressions

ucircs are also used when checking field access, field update and method call expressions. Within a subject, an object whose context is given by an unk must always be referred to using the same unk. It is an error for an unk to bind to an exp (and vice versa) in a method call, field update, or any other expression.

There are number of cases, we will go through each one in turn.

- First, in Figure 6-14, suppose that we are checking the expression in the body of method `U.foo(..)`. Consider the analogy with exps. Recall that exps number the dominators of an object. Suppose `U.v` has type `V<1>`. By definition or exps, `v` is inside exp$_1$. An exp$_2$ owned object `t` cannot reference `v`; conversely, contrary to the definition of exps, exp$_1$ does not dominate `v`. In order for `t` to reference `v`, `t` must be inside `v`.

  Now examine Figure 6-14. In the body of class `U`, in order for `t` to reference `v`, `t` must be inside `v`. The resolution set of unk$_k$ may contain neither exp$_0$ nor exp$_1$. The resolution constraint in the `where` clause of `U` formalises this constraint. Note this ucirc may be omitted when an unk's resolution set is unrestricted.

  One special case concerns the treatment of the self reference this. The owner of this is always given by exp$_1$. Although it is technically valid to pass the self reference in relation to an unk whose resolution set is $\{0, 1\}$, in the present work this can be passed only in relation to exp$_0$ and exp$_1$. This is a simplification that is intended to improve clarity with minimal impact on subject reusability.

- Secondly, in Figure 6-14, suppose that we are checking the expression in the body of method `U.fee(..)`. By the same principle as in the previous case, in order for `w` to reference `x`, `w` must be inside `x`. The owners of these objects are given by unks, so this constraint is expressed at subject level by `ucirc m <= n`.

```
subject S {
  unk k;
  class F {
    H<k> h;
    J<2> j;
    void foo() { h.j = j; }
  }
  class H {
    J<3-k> j;     // o-o!
  }
}
```

Figure 6-15: Further checking of expressions involving unks.

```
subject S {
  unk k, m;
  ucirc k <= m;
  class F {
    H<k> h;
    J<m> j;
    void foo() { h.j = j; }
  }
}
class H where 1 <= m {
  J<m> j;       // that's better!
}
```

Figure 6-16: Yet more checking of expressions involving unks.

- Thirdly, we disallow expressions which place an unk owned object as the receiver against types containing exps in method parameters, return types, or field variables. The resulting type contains context expressions which may be correct but difficult for the reader to comprehend. Class `H` in Figure 6-15 shows the context expression produced by exp arithmetic. According to Δ_2 the type of `F.j` is correctly given as $\Delta_2(k, 3 - k) = 2$. To avoid context expressions involving unks, the programmer should introduce unk$_m$ and rewrite the ucircs as shown in Figure 6-16.

Type checking expressions whose types contain unks introduces two further correctness properties:

- **Property 6**. The ucircs must imply the inter-unk ordering required by the type or expression.
- **Property 7**. The ucircs must imply the resolution constraints required by the type or expression.

6.3.4 Per-Class Checks

unks are abstractions over exps and so inherit all the properties associated with exps. unks represent a choice of exps. So a program that uses unks should have a choice of exps to use in the place of its unks. An ownership structure that cannot be expressed using exps also cannot be expressed using unks. For example, type checking fails in `R.foo()` in Figure 6-17 because one of the two field update expressions is invalid. This program should also fail type checking when unks are used in the place of exps. The unk checks we have described to now are insufficient. They check each semicolon delimited expressions separately but this condition requires the recording of history. As shown in the following

```
1    subject S {
2      class R {
3        T<0> t0;
4        T<1> t1;
5        V<2> v;
6        void foo() {
7          t0.v = v;   <- valid
8          t1.v = v;   <- invalid
9        }
10     }
11     class T {
12       V<3> v;
13     }
14     class V { ... }
15   }
```

Figure 6-17: Additional checks for unks. Case 1.

table, there are two problem cases. Columns **Case 1** and **Case 2** show the replacement types for declarations in Figure 6-17.

Declaration	**Case 1**	**Case 2**
`t0` in line 3	`T<0>`	`T<a>`
`t1` in line 4	`T<1>`	`T<b>`
`v` in lines 5 and 12	`T<k>`	`T<k>`

In case 1, given expressions in lines 7 and 8 of Figure 6-17 and a valid resolution for unk_k in R, unk_k will resolve to different exps in T. In case 2, if unk_a and unk_b resolve to different exps in R then unk_k will resolve to different exps in T. Conceptually, two unks refer to two different contexts, so unk_a and unk_b should be replaced by a single unk.

To detect these cases the Subject Compiler uses an `ExpressionRepository` that records the types of receiver objects of affected expressions. Entries have the following form:

$$\langle k, C, v\rangle$$

where k is an unk whose usage is recorded. It appears in a field, return value or parameter type. C is the class containing the expression. v is either an unk or exp denoting the last usage. Method calls, field accesses and updates trigger a look-up and a possible update to the `ExpressionRepository`. Errors are detected as follows:

- In case 1 at line 7, the `ExpressionRepository` object has no prior entry for $\langle \mathsf{unk}_k, \mathtt{T}\rangle$, so tuple $\langle \mathsf{unk}_k, \mathtt{T}, \mathsf{exp}_0\rangle$ is inserted. Upon checking code in line 8, we lookup $\langle \mathsf{unk}_k, \mathtt{T}\rangle$ which returns exp_0. But, $\mathsf{exp}_0 \neq \mathsf{exp}_1$, so line 8 fails type checking.

- In case 2 at line 7, the `ExpressionRepository` object has no prior entry for $\langle \mathsf{unk}_k, \mathtt{T}\rangle$, so tuple $\langle \mathsf{unk}_k, \mathtt{T}, \mathsf{unk}_a\rangle$ is inserted. Upon checking code in line 8, we lookup $\langle \mathsf{unk}_k, \mathtt{T}\rangle$ which returns unk_a. But, $\mathsf{unk}_a \neq \mathsf{unk}_b$, so line 8 fails type checking.

Our next correctness property states:

- **Property 8**. All type checking of expressions whose types contain unks uses the `ExpressionRepository`.

```
class Queue<data> {

  Link<0, data> head = null;
  Link<0, data> tail = null;

  void put(Object<data> o) {
    Link<0, data> l = new Link<0, data>(o);
    if(head == null) {
      head = tail = l;
    } else {
      tail.next = l;
      tail = l;
    }
  }

  Object<data> get() {
    if(head == null) return null;
    Object<data> o = head.o;
    if(head == tail) {
      head = tail = null;
    } else {
      head = head.next;
    }
    return o;
  }
}

class Link<d> {
    Object<d> o;
    Link<1, d> next;
    Link(Object<d> o) { this.o = o; }
}

class QueueClient {
  Queue<0, 1> q1;
  Queue<0, 0> q2;
}
```

Figure 6-18: `Queue` class implemented using Subjective Ownership Types

6.4 Classes with Ownership Parameters

Abstract data types are implemented as classes with ownership parameters in SOT. These classes are special because the owners of the data they reference should be specifiable independently from the owner of the abstraction itself. Multiple instances of the same kind of abstraction, possibly with different aliasing properties, may be required in the body of the same class. `unks` are insufficient to specify this kind of diversity because `unks` are resolved per class while we require per object variability.

Our solution is to adopt ownership parameters for additional contexts which should be specifiable parametrically. Linked lists, queues and stacks have a single ownership parameter for their data. Hashtables have two ownership parameters: one for the keys and the other for the values. Figure 6-18 shows class `Queue` implemented using ownership parameters. Multiple instances of `Queue` with different ownership properties can be created in class `QueueClient`. Contrast this design with the one for Ownership Types shown in Figure 5-2 on page 75.

The set of contexts that can be referenced in the implementation of `Queue` is $\{0, 1, \mathsf{data}, \mathsf{world}\}$. exp_0 and exp_1 denote the representation context and the owner of the current instance. No other exps are allowed. Observe that classes do not declare the owner parameter in the list of ownership parameters.

6.4.1 Composable and Uncomposable Classes

Classes with ownership parameters are *uncomposable* in SAPS: subject-oriented composition rules which describe join points inside these classes are disallowed, but delegation and inheritance can still be used to extend uncomposable classes. Classes that use exps and unks in their definition are *composable*. These can participate in inheritance, delegation and all subject-oriented composition rules.

The inheritance hierarchies of composable and uncomposable classes are separate except for the uncomposable default class `Object` found at the root of both composable and uncomposable hierarchies. A class which does not declare a superclass implicitly extends `Object`. Uncomposable classes are distinguished syntactically by having a (possibly empty) comma separated sequence of ownership parameters following the class name as shown in Figure 6-18. Both kinds of classes share the same name space and it is illegal to have an uncomposable and a composable class with the same name in one subject.

As detailed at the beginning of this Chapter, the motivation for separation comes from a series of observations detailed in Chapter 5. Common to these is the tug-of-war between wanting to cleanly separate concerns while still providing encapsulation. In Section 2.3.3 on page 15 we compared black-box and white-box reuse strategies. Programmers as experts of their trade often require access of certain key aspects of implementation. We concluded that in order to be adaptable to unanticipated changes a component needs to provide facilities for changing from the inside. But there is often a limit to the details that will benefit the expert. Section 2.3.3 used the racing driver analogy to motivate this model. In Section 3.1.3 on page 27 it was noted that the MDSOC model aims to improve the modularity of scattered and tangled concerns but does not improve encapsulation. Subject-Oriented Programming is a flexible model for separating and recomposing concerns. It provides access to implementation that programmers require but it has no programmatic way of marking a component as an implementation abstraction which should not be decomposed further. SOP lacks the encapsulation mechanisms that stop the programmers being overwhelmed by implementation details.

In SAPS, uncomposable classes are a way of marking a class as an implementation abstraction. The most immediate example of an implementation abstaction is a class such as `Queue` shown above. The vast majority of software developers will not want to know how `Queue` is implemented, they are black-box users of `Queue`. Later on, the same will probably be true of subject-oriented programs implemented using `Queue` objects. New components constructed by composing subjects will be used as basic building blocks in larger systems. And so the cycle will continue.

The subject designer chooses whether to define a class as composable or uncomposable. Overall we expect that developers will be defining composable classes. Uncomposable classes are used mostly to define new kinds of container abstractions. Unlike abstractions of "real world" entities, containers are characteristic of an objective perspective and have been the hallmark of object-oriented programming. In general, the implementations of these abstractions cannot be improved by subject-oriented decomposition.

We anticipate that common ADTs will be reused from uncomposable class libraries. Definition of new uncomposable classes should be relatively rare and limited to the following cases:

- To define new kinds of container abstractions.
- To define classes whose instances need to be reused in conjunction with different ownership properties, and the system of `exps` cannot provide the required flexibility.
- To create abstractions that should be sealed, thereby artificially restricting the composition interface of a subject.

Class `Queue` shown in Figure 6-18 exemplifies the first point. For the second point we propose the following heuristic:

> Create an uncomposable class when requiring an abstraction that may have many instances, possibly in many subjects, each with different ownership properties. Create a composable class otherwise.

In a later Subsection we will present the limitations of the `exp` notation compared to parameterisation. The evaluation Chapter 8 contains examples of uncomposable classes being used for interface restriction and for sealing.

`unks` or Ownership Parameters?

`unks` are not the same as ownership parameters and they can never be confused for one another. `unks` are used in composable classes and ownership parameters are used in uncomposable classes. They are conceptually similar in the sense that both provide a way of parameterising programs, allowing third parties to customise ownership structures. Beyond this, `unks` and ownership parameters work very differently:

- The scope of an ownership parameter is a class and the classes derived from it. The scope of an `unk` is a subject – a collection of heterogeneous classes.
- `unks` are resolved by subject composition and ownership parameters are bound during object instantiation.
- An `unk` can bind an ownership parameter of an uncomposable class, but an ownership parameter is never resolved.

6.4.2 Interaction Between The Hierarchies

Interaction between hierarchies encompasses both inheritance and aggregation, the so-called *is-a* and *has-a* hierarchies respectively. Apart from a common root, the two inheritance hierarchies cannot be mixed. It is not possible to create an uncomposable class by inheriting from a composable class or vice versa. Although there may be implementation related reasons for doing so, conceptually the two hierarchies serve different purposes and so should not be mixed.

When defining a new composable class, instances of both composable and uncomposable classes can be used in the implementation. The evaluation Chapter 8 relies extensively on the uncomposable class `Vector` whose core interface (and that of the associated `Iterator`) is shown in Figure 6-19.

```
class Vector<data> {
  void add(Object<data> o);
  Object<data> remove(Object<data> o);
  Iterator<1, data> iterator();    // a version that respects deep ownership
}
class Iterator<data> {
  bool hasNext();
  Object<data> next();
}
```

Figure 6-19: `Vector` class core interface.

```
class Pair<m, n> {
  X<m> fst;
  Y<n> snd;
}

class PairQueue<p, q, r> {
  // p binds owner of Pair, q binds Pair.m, r binds Pair.n
  void put(Pair<p, q, r> obj) { ... }
  ...
}
```

Figure 6-20: Class `PairQueue` specialised to uncomposable classes.

When types are created exps and unks bind the ownership parameters of uncomposable classes but the way ownership parameters are used in uncomposable class definitions is hidden from clients.

Uncomposable classes are defined using instances of both composable and uncomposable classes. `Queue`s and `Vector`s store instances of `Object`. Objects of any class can be stored because all classes derive from `Object`. Specialised subclasses of `Queue` or `Vector` may be usable only with uncomposable classes. For instance, in Figure 6-20 class `PairQueue` is usable only with objects of uncomposable class `Pair`.

Composable class instances can be used in the definitions of uncomposable classes only when the implementation is restricted to accessing members whose types feature contexts exp_1 and world only. exp_0 is the representation context which cannot be accessed. unks and all other exps are undefined in uncomposable classes. Thus, container classes which store references to objects of composable classes but never access the interface can always be used with composable class instances. This restriction introduces another property necessary for SOT correctness:

- **Property 9**. Expressions in uncomposable classes must be restricted to contexts $\{1, \mathsf{world}\}$ in the members of composable classes.

Subclasses of uncomposable classes can declare additional ownership parameters to those which are inherited. Uncomposable classes can be declared in subjects; however, it is best to declare them in class libraries. The libraries can then be imported by all subjects which need to use these abstractions in their implementations. Those which are declared in subjects are not modified by composition, but depending on the composition rules either forwarded to the output subject unchanged or discarded.

```
1    class E<p, q> where p <= q {
2      Queue<p, q> s;  // requires p <= q
3      X<0, p> x;
4              Y<p> y;
5      void foo() {
6        x.y = y;
7      }
8      class X<p> {
9        Y<p> y;
10     }
11     class Y<> { }
12   }
```

Figure 6-21: Ownership parameter ordering.

6.4.3 Ownership Parameter Ordering

In uncomposable classes the owner parameter is implicitly inside other parameters. Therefore, any substitution must satisfy the implicit constraint of uncomposable classes that requires the owner to be inside all other contexts. This leads to our next property:

- **Property 10**. In a substitution for the parameters of an uncomposable class the context binding owner must be inside other contexts.

Whereas for exps the ordering of contexts is explicit in the notation, the ordering of contexts represented by ownership parameters may need to be made explicit. To help explain the issue, consider the example in Figure 6-21. Line 2 requires `p <= q` because the `Queue` owner must be inside the data referenced by the `Queue`. Line 6 requires `1 <= p` because contexts that bind to ownership parameters must be outside the context that binds exp_1, i.e. the owner context. In order to disambiguate the context ordering in view of such types and expressions, we again employ where clauses. These clauses are required only where the context ordering needs disambiguating. So if no type or expression in `E` depends on `p <= q` then the where clause is not required. Constraint `1 <= p` is implicit in SOT and does not require a where clause. The requirement for ordering of ownership parameters introduces our next property for representation containment:

- **Property 11**. A substitution for the ownership parameters of an uncomposable class must obey the context ordering in its where clause.

As in composable classes, cycles in the constraints specified in where clauses should be removed because they misleadingly represent a choice of contexts. Subclasses of uncomposable classes may define additional ownership parameters and constraints on ownership parameter substitution. To ensure representation containment we require the final property:

- **Property 12**. The ownership parameter ordering of uncomposable classes must be well formed.

The use of types derived from an uncomposable class `Map` in both composable and uncomposable classes is shown in Figure 6-22. A map is an ADT that stores key, value pairs and supports operations for addition and removal of pairs. This abstraction has a containment property which states that values may be accessed only where the keys can be accessed, formalised in line 1 by a constraint on ownership parameters of class `Map`. `Map` is defined as an uncomposable class; it is a traditional

```
class Map<key, value> where key <= value {
  void put(Object<key> k, Object<value> v) { ... }
  Object<value> get(Object<key> k) { ... }
}

subject SafetyBoxFeature {
  class SafetyBox<data> {
    Map<0, 0, data> datamap;
    Password<0> password;
    Object<data> getItem(String pw) {
      if(password.accepted(pw)) return datamap.get(password);
    }
  }
  class Password {
    bool accepted(String pw) { ... }
  }
}

subject AddNewAccount {
  unk k, m;
  ucirc k <= m;
  class AccountPortfolio {
    Map<0, k, m> accounts;
    addAccount(Account<k> acc, Integer<m> amount) {
      accounts.put(acc, amount);
    }
  }
  class Account {
    String holder;
    ...
  }
}
```

Figure 6-22: Using types derived from uncomposable classes.

object-oriented abstraction whose design cannot be improved by subject-oriented decomposition. Ownership parameters enable multiple maps to be created, each with different ownership properties.

Figure 6-22 defines two clients of `Map`. The uncomposable class `SafetyBox` uses a map to hide data. Data can be accessed by providing the right password. `SafetyBox` is an example of an abstraction constructed from and built on top of an existing `Map` class. Its designer envisages that a reuser will require multiple safety boxes to implement a security policy subject, each with a different aliasing policy. In line 8, to create a type from `Map` the available contexts are substituted for the ownership parameters of `Map`. The first identifier denotes the owner, the second and third bind `key` and `value` respectively. `Map` requires that `owner <= key <= value`. The substitution satisfies this condition.

Another client of `Map` exists in the `AddNewAccount` subject. This subject implements a feature for adding a new account to a portfolio. Class `AccountPortfolio` is composable; this subject will be composed with others to create a suite of financial tools. In order to maximise reusability the subject defines unk_k and unk_m to denote the owners of the `Account` object and the amount it contains. These will be resolved by other subject with which `AddNewAccount` will be composed. A map is used in `AccountPortfolio` to store the accounts and their amounts. exp_0, unk_k and unk_m bind the owner context and other ownership parameters of `Map`. Properties 10 and 11 ensure that only substitutions satisfying the ownership parameter ordering are accepted. Validity depends on `ucirc k <= m` (line 21) because `Map` requires that the context binding to `key` is inside the context binding to `value`. Correctness property 6 ensures that it is possible to observe this constraint from the ucircs. Conversely, this would be an invalid substitution.

6.4.4 Strengths and Limitations of the System of exps

exps are well suited to annotating the ownership properties of subjects because SOP shifts from classes to collaborations of classes as units of interest. In object-oriented programming the class is the modular unit; an object of a class may appear at any point in an object graph including at the top with world as owner. No assumptions can be made about objects outside the owner. In subject-oriented programming, subjects often implement collaborations where objects are tightly coupled. The subject is the pertinent modular unit. We think of objects as playing very particular roles in collaborations; the subject creator has *a priori* knowledge of the existence of objects external to the owner. This is very well demonstrated by the `FloorPressButton` subject: it is possible to sketch the ownership structure of `FloorPressButton` from the types of `inBulding` and `thisFloor` in Figure 6-4. The labelling of dominators numerically contributes to making SOT into an elegant system for specifying programmer intent.

The ordering of contexts inherent in the exp notation can be achieved with ownership parameters by specifying an order for the contexts bound to ownership parameters. This ordering is explicit in Boyapati et al [19]. However, ordering alone does not address the fundamental problems caused by the combination of ownership parameters and subjectivity (see Section 5.3, page 88).

Compared to Ownership Types, SOT can be classed as more permissive. The system of exps allows one object to reference another with a different owner without prior permission. In Ownership Types, a permission in the form of ownership parameterisation is always required. Seen from a collaboration perspective this is a strength: pre-established collaborators generally do not seek permission to communicate; subjects pre-establish the boundaries of object collaboration.

Compared to ownership parameterisation, explicit contexts appear to hardwire the ownership

```
subject S1 {
  class A {
    Pair<0, 0, 1> p1;
    Pair<1, 1, 1> p2;
    X<0>                f1() { return p1.fst; }
    Y<1>                f2() { return p1.snd; }
    X<1>                f3() { return p2.fst; }
    Y<1>                f4() { return p2.snd; }
  }
  class Pair<m, n> {
    X<m> fst;
    Y<n> snd;
  }
  class Main {
    void main() {
      A<0> a;
      Y<0>              y1 = a.f2();
      X<0>              x2 = a.f3();
      Y<world>          y2 = a.f4();
    }
  }
}
subject S2 {
  class A {
    Pair<0> p1;
    Pair<1> p2;
    X<0>                f1() { return p1.fst; }
    Y<0>                f2() { return p1.snd; }
    X<1>                f3() { return p2.fst; }
    Y<1>                f4() { return p2.snd; }
  }
  class Pair {
    X<1> fst;
    Y<1> snd;
  }
  class Main {
    void main() {
      A<0> a;
      // Y<0>           y1 = a.f2(); representation exposure
      X<0>              x2 = a.f3();
      Y<0>              y2 = a.f4();
    }
  }
}
```

Figure 6-23: Example with `Pair` composable/uncomposable

- **Property 1**. If exp_0 is in the type of a parameter, return value or field variable then this is the only valid receiver expression.
- **Property 2**. The context in the actual parameter must be as given by Δ_2.
- **Property 3**. world is external to all contexts denoted by exps.
- **Property 4**. The resolution sets of all unks must be well formed.
- **Property 5**. The context substitutions for ownership parameters of arrays must allow access to lower dimensions wherever higher dimensions are accessible.
- **Property 6**. The ucircs must imply the inter-unk ordering required by the type or expression.
- **Property 7**. The ucircs must imply the resolution constraints required by the type or expression.
- **Property 8**. All type checking of expressions whose types contain unks uses the `ExpressionRepository`.
- **Property 9**. Expressions in uncomposable classes must be restricted to contexts $\{1, \mathsf{world}\}$ in the members of composable classes.
- **Property 10**. In a substitution for the parameters of an uncomposable class the context binding owner must be inside other contexts.
- **Property 11**. A substitution for the ownership parameters of an uncomposable class must obey the context ordering in its where clause.
- **Property 12**. The ownership parameter ordering of uncomposable classes must be well formed.

Figure 6-24: Correctness Properties for SOT

structure into the design of each class. Subject design with exps forces the developer to make decisions about the ownership structure. This was a motivating factor in the introduction of unknown context identifiers. When exps cannot provide the required flexibility, uncomposable classes must be used. To demonstrate the limitations of the exp notation consider Figure 6-23. The Figure shows two subjects: in `S1` class `Pair` is uncomposable and in `S2` it is composable. In `S1` the ownership parameters of `A.p1` and `A.p2` are bound to different contexts. Method calls to `A.f2()`, `A.f3()` and `A.f4()` from `Main.main()` are valid but a call to `A.f1()` would cause representation exposure. Turning to the case where `Pair` is composable, note that no choice of exps within `A` and `Pair` can produce the same types for the methods of `A`. `S2` shows one of a number of failed attempts. exps cannot adapt to the subtle differences in ownership structures. unks cannot address this problem: an unk resolves to one exp per class and not per object. The subject designer must use the uncomposable class `Pair` if the full range of ownership structures is required within the subject.

6.4.5 Types and Type Checking

We are now in a position to summarise the properties and describe valid types. SOT correctness is ensured by the checks performed when enforcing the properties given in Figure 6-24. Before expression checking commences work is carried out to check property 4.

A Subjective Ownership Type is created by substituting the available context identifiers for the owner and any other ownership parameters of a class. Different types are created in the bodies of composable and uncomposable classes. In composable classes, the set of contexts to choose from includes the exps, the unks and world. In uncomposable classes, the selection is made from the current set of ownership parameters, exp_0, exp_1 and world.

The validity of substitution relies on context identifier ordering in the class where the type is formed and on the substitution constraints of the uncomposable class from which the type is formed. Inside a composable class properties 3, 6, 7, disambiguate the order of substituted contexts and properties 5, 10 and 11 ensure that the substitutions are valid. In an uncomposable class properties 3, 10 and 11 disambiguate the order of substituted contexts and properties 5, 10, 11 ensure that the substitutions are valid.

When checking composable classes, the check for property 1 guarantees representation containment. Checks for properties 2, 3, 6, 7 and 8 ensure that expressions are well formed. When checking uncomposable classes, representation containment is ensured by property 1 also. Property 12 ensures that ownership parameters are ordered in a structured way. Property 9 makes certain that objects of composable classes are accessed correctly by expressions. Expression checking proceeds by extracting the substitution for the ownership parameters and checking that the type of the actual parameter matches the expected type in the field or method parameter. Ownership parameters are adopted from Ownership Types, and the checks performed by the Subject Compiler mirror the description in [23].

6.5 Conclusion

This Chapter has introduced Subjective Ownership Types for use in subject design. Subjective Ownership Types provide alias protection for objects and support deep ownership. With SOT two kinds of classes can be created:

- *Composable classes* use a new system of explicit and unknown context identifiers. We believe that explicit contexts identifiers or exps are well suited to specifying the intended ownership properties of collaborations implemented by subjects. The domination structure is explicit in the notation. Unknown context identifiers or unks make it possible to defer the selection of an explicit context until composition. The full range of subject-oriented composition rules can be used on composable classes.

- *Uncomposable classes* use ownership parameters to create black-box components where the ownership properties can be set by the client on a per-object basis. Interface-level subject-oriented composition rules which affect instances of uncomposable classes are allowed.

Composable and uncomposable classes exist in separate class hierarchies but have a common superclass. SOT allow instances of one kind of class to be safely used in the definition of the other kind. Uncomposable classes enable the creation of black-boxes which are used as building blocks in subject construction. The design of subjects uses composable classes predominantly. Features implemented by subject can be composed to form larger grained components.

The following Chapter presents the second part of the Subjective Alias Protection System: extensions to Subject-Oriented Programming to support the composition of subjects annotated with Subjective Ownership Types.

Chapter 7

SAPS – Subject Composition

This Chapter is concerned with all aspects of subject composition. Its overall aim is to present a model of composition that synthesises subjects annotated with Subjective Ownership Types (SOT). This aim can be broken down into three top level objectives:

- Review the composition rules and describe the underlying model for subject composition.
- Describe the effect of composition on SOT concepts presented in Chapter 6.
- Specify SAPS by defining the necessary extensions to the underlying model to support the composition of SOT annotated elements.

Chapter 3 has already presented much of the syntax of the SOP composition language. After reviewing composition rules, we will show that composition primarily involves three activities: bringing together artifacts which should be composed, reconciling their differences and synthesis. But before artifacts can be brought together each subject must be decomposed into its composable elements. We will present the system of labels – a model for describing the composable elements of subjects. Composition rules written by the subject integrator are parsed into a series of composition directives. *Groupers* are directives that bring together elements and perform reconciliation. *Combinator* perform synthesis.

For the second item, we will discuss the challenges of composing SOT annotated elements. One way to achieve deep ownership in the output subject is for composition to preserve the ownership declarations in the input subjects. Correctness depends on the notion of type equivalence and SOT-aware composition rules. Equivalence is not as restrictive as it may sound. Ownership types have data and context identifier components; class composition leads to datatype equivalence, and explicit contexts can be composed with unknown context identifiers. Unification of explicit and unknown contexts yields resolutions: the unknown context identifier becomes bound to the explicit context identifier. In the model we propose all unknown context identifiers must be eliminated; that is, composition of subjects should map all `unks` to `exps`. With SOT-aware composition rules composition elements are either forwarded to the output unchanged or modified in a semantically consistent way. No declarations are removed. We will argue that a consistent mapping of this kind leads to desirable deep ownership properties in the output subject.

For the third item, we will specify the elements required to implement the model described as part of the second item. In order to support SOT concepts, the system of labels will be extended with new primitives. We will specify type combinators for checking element composability and introduce functions that collect `unk` resolutions. An important part of SAPS is checking that a collection of `unk` resolutions is consistent. We will present an algorithm that, given a set of resolutions, attempts to resolve all `unks` in a subject. The algorithm performs `unk` resolution propagation. It relies on static links between classes to forward resolutions from one class to the next.

Section 7.1 is analysis of past work on SOP and Hyper/J. It summarises the composition rules and redescribes the effect of composition on subjects. Section 7.2 presents the system of labels – a model of primitives on top of which we define composition directives. Our model is an extensive reworking of the original [94, 95] and forms a part of the contribution to the thesis. Section 7.3 defines groupers which manipulate elements from the system of labels and bring together elements from different subject that should be composed. The above three Sections conclude the presentation of subject composition in general, i.e. not specific to composition of subjects annotated with Subjective Ownership Types. The following Sections are concerned with extensions that will enable SOT annotated elements to be composed, they are also part of our contribution. Section 7.4 describes the meaning of composing SOT annotated elements. Section 7.5 defines type combinators and resolution mapping functions. Combinators check that the types of grouped elements can be composed and, where necessary, determines the type in the output subject. Resolution mapping functions collect `unk` resolutions and store them as attributes of labels. At the end, the results of resolution mapping functions are used by resolution propagation functions defined in Section 7.6. Section 7.7 concludes this Chapter.

7.1 Composition Rules

In this Section we review and analyse the composition rules first seen in Chapter 3. The SOP language is an extensible collection of composition rules. Applied to a collection of input subjects the composition rules cause an output subject to be created. Composition takes place statically, before the program is run. Each input subject realises some concern by defining a set of classes, field variables and operations whose execution will produce the desired behaviour. The output subject also is a set of classes, field variables and operations. The behaviour of the output subject depends on the input behaviours and the semantics of the composition rules used in its creation. The subject integrator must choose the right mix of composition rules to produce the intended behaviour. Most compositions can be achieved with a relatively small selection of rules. However, more exotic rules can be defined for particular cases. For instance, in Section 4.3 on page 52 we proposed the **view-merge** rule which directly addressed the needs of a tricky interaction issue.

For the most part, composition is about bringing together definitions from the input subjects. The main difference between rules concerns the action to take at the point of contact, i.e. at the join point. In the core set of most commonly used rules there are four actions:

unify By far the most common action is to unify the elements, e.g. create a single `Employee` class based on a set of `Employee` classes from input subjects.

slct Select one element from a range of alternatives, e.g. choose one `setAccountNo(..)` operation implementation from a set of implementations in the input subjects.

exec Prepend a set of behaviours with another behaviour, or append a new behaviour to an existing set, e.g. prepend `Caching` behaviour in order to save the values passed as parameters to operations.

call Insert a trigger such that a behaviour is invoked before or after a set of behaviours, e.g. trigger a `balanceCheck(..)` operation for account withdrawals of bank customers with limited borrowing.

The first two actions are *symmetric* and the last two are *asymmetric*. In a symmetric action the elements at a join point are different parts of the same concept. A single element will be created in the output. Unifying the elements brings together all the definitions from all input subjects and produces one definition in the output. Selection nominates one definition above all others and puts that into the output subject. When the output subject is run, a reference to any input element involved in a symmetric action will return the output element. We call this process *forwarding* because references to input elements are forwarded to the output element. In a symmetric action, all input elements forward to the same set in the output.

In an asymmetric action, a new element e is partnered with each element in an existing set of elements $\{s_1 \ldots s_n\}$. The new element adds to existing concepts. A set of elements $\{\langle s_1, e\rangle \ldots \langle s_n, e\rangle\}$ will be created in the output. In asymmetric actions forwarding takes place from the input elements s_i to the output elements $\langle s_i, e\rangle$, e forwards only to e. For example, `Caching` behaviour is added to multiple abstractions. When messages are dispatched to these abstraction, the cache behaviour is invoked. In most cases, it is not expected that invocation of cache behaviour directly will invoke the cached abstractions.

Symmetric and asymmetric actions may be combined. In the simplest case, an asymmetric action will apply after the symmetric action. For instance, a set of elements $\{s_1 \ldots s_n\}$ may be composed using a symmetric action to form S. A new element e will be composed with S using an asymmetric action to produce some output element $\langle S, e\rangle$. Other more exotic composition rules are also conceivable.

There is an important difference between the two kinds of asymmetric actions shown in the above list. **exec** does not distinguish the source of the request for behaviour. The behaviour is adapted universally and is the same for all clients. In **call** the behaviour is adapted more specifically to the needs of the client. For instance, when registering with a medical centre, the nurse asks all new patients to register their personal details. The procedure is universal and **exec** should be used. But, suppose a bunch of patients with flu symptoms attend the medical centre. The treatment dispensed by the doctor is not the same for all patients with flu symptoms. For instance, patients who recently returned from abroad may undergo additional tests to those who have not had any recent foreign trips. This interaction may use **call** to specialise behaviour.

The concepts of bringing elements together, forwarding, symmetric and asymmetric actions underlie subject-oriented composition rules. A core set of composition rules is summarised in Figure 7-1. We will use these composition rules when writing composition specifications.

The subjects to compose and the name of the output subject are specified using **compose**. **equate** establishes that the named input elements should be composed in some way but does not specify the action to take during composition. **merge** builds on **equate**. In addition to grouping a set of elements it also specifies an action. Applied to a set field variables, **merge** creates a single field variable in the output. Applied to a set of operations, **merge** creates a single operation in the output. Applied to a set of classes, **merge** creates a single class in the output. The **merge** of operations and field variables is meaningful only in the context of a class, and the Subject Composer

compose. Specifies a sequence of subjects to compose and the name of the output subject, e.g. `compose S1, S2 into S;`.

equate. Groups together elements of the same kind, giving the grouped element a new name in the output subject. Overloaded by element type. For example, `equate S1.A, S2.B, S3.C into S.D;` equates classes.

merge. Specifies the **unify** action on a set of elements of the same kind, giving the output element a new name. It is overloaded by element type, e.g. `merge S1.A.v, S2.A.w into S.A.x;` merges instance variables.

override. Specifies the **slct** action on a set of operations. One operation is nominated above others, this is the overriding operation. The output operation can be given a new name, e.g. in `override S1.B.foo, S2.B.bar with S1.B.foo into S.B.bee;` it is applied to operations.

bracket. Specifies either an **exec** or a **call** action on a set of operations. The operations to be bracketed (or wrapped) can be specified exactly or using wildcards. The brackets (or wrappers) can include **before** and **after** parts which are executed either immediately before or immediately after the wrapped operations. The intergrator may specify either **before** or **after** and need not specify both parts.

The **exec** form of brackets has three parts: the wrappee specification, the **before** and **after** wrappers. At most one of **before** and **after** wrappers is allowed to be null. For example, consider `bracket ‘‘*.foo’’ with before S1.A.bar after S1.A.bee;`. This rule will cause operation `S1.A.bar` to execute immediately before any operation matching pattern `*.foo` and `S1.A.bee` to execute immediately after any operation matching the same pattern.

The **call** form of brackets has an extra **from** part that narrows the set of matched operations. It can be either a list of classes or operations. For example, `bracket ‘‘*.foo’’ from S2.B, S2.C with before S1.A.bar after S1.A.bee;` is identical to the **exec** form above with the exception that the wrappers are run only when called from the methods of classes `S1.B` and `S2.B`.

mergeByName. Brings together all identically named elements and applies the **unify** action throughout.

overrideByName. Brings together all identically named elements and applies the **slct** action throughout. The first subject in the **compose** clause is the overriding subject (the source of the overriding elements).

order. Specifies an order for operation composition, e.g. `order S1.A.foo after S2.A.foo;` Used in conjunction with **merge** on operations. By default **merge** does not imply an order.

Figure 7-1: Composition rules summary.

```
subject S1 {          subject S2 {           subject S3 {
  class A {             class A {              class B {
    int value = 1;        int value = 1;         int value = 1;
    void f() {            void f() {             void g(){
      value += 3;           value *= 3;            value *= 5;
    }                     }                      }
    void S1_op() {        void S2_op() {         void S3_op() {
      f();                  f();                   g();
    }                     }                      }
  }                     }                      }
}                     }                      }

// subject external clients:
A a1 = new A();
A a2 = new A();
B b = new B();
a1.S1_op();
a2.S2_op();
b.S3_op();
```

Figure 7-2: Example showing

is required to check that the output field variable or operation has a valid destination. The **merge** of classes sets the output class but does not group the class members.

override also builds on **equate** by specifying an action. It applies only to operations, selecting one operation implementation over others. **override** on field variables and classes works the same as **merge**.

bracket is a composition rule that combines both a grouping facility and actions. The operations to be bracketed are grouped with the bracket operations, and either the **exec** or the **call** action is applied to each grouping. Behind the scenes, the **bracket** composition rule uses the **unify** action to compose the classes containing the wrappers and the wrappees, and the **slct** action to compose wrapper and wrappee operations.

The composition rules presented to now provide fine grained control over the composition elements. But using these rules to compose non-trivial programs would produce very lengthy composition specifications. In order to make composition specification more concise, SOP introduces top-level composition rules which group elements based on a strategy of some kind. For subjects designed in concert, grouping by name is useful. Identically named elements of the same kind across all input subjects can be brought together: classes are grouped with classes, operations with operations and field variables with other field variables. **mergeByName** and **overrideByName** are two composition rules that combine grouping based on name equivalence with an action. Whenever **mergeByName** or **overrideByName** are used, the other composition rules become exceptions to the groupings and actions implied them.

To help ground the presented concepts consider the three subjects shown in Figure 7-2. Suppose that these subjects are to be composed: `compose S1, S2, S3 into S;`. Each subject contains method `S*_op` that calls either `f` or `g`. Figure 7-2 shows three subject-external clients of the output subject which call operations `S*_op` in turn. Before the call, `a1.value` = 1, `a2.value` = 1 and `b.value` = 1. Figure 7-3 shows the `value` field after the execution of each `S*_op` based on the composition specification shown in the row.

(1) shows that without any additional rules the calls are malformed. Without addition rules

	composition specification ↓	`a1.value`	`a2.value`	`b.value`
(1)	no additional specs	invalid	invalid	invalid
(2)	`overrideByName;`	4	4	5
(3)	`mergeByName; order S2.A.f after S1.A.f`	12	12	5

(4)	`overrideByName; merge S1.A.f, S2.A.f into S.A.f;`
	`order S2.A.f after S1.A.f;`
		12	12	5

(5)	`mergeByName; bracket ‘‘A.f’’ with before S3.B.g;`
	`order S2.A.f after S1.A.f`
		24	24	5

(6)	`mergeByName; bracket ‘‘A.f’’ with after S3.B.g;`
	`order S2.A.f after S1.A.f`
		60	60	5

(7)	`mergeByName; bracket ‘‘A.f’’`
	`from S1.A with before S3.B.g;`
	`order S2.A.f after S1.A.f`
		24	12	5

(8)	`mergeByName; bracket ‘‘A.f’’`
	`from S1.A with after S3.B.g;`
	`order S2.A.f after S1.A.f`
		60	12	5

(9)	`overrideByName; bracket ‘‘A.f’’ with before S3.B.g;`	8	8	5
(10)	`overrideByName; bracket ‘‘A.f’’ with after S3.B.g;`	20	20	5

(11)	`overrideByName; bracket ‘‘A.f’’`
	`from S1.A with before S3.B.g;`
		8	4	5

(12)	`overrideByName; bracket ‘‘A.f’’`
	`from S1.A with after S3.B.g;`
		20	4	5

Figure 7-3: Results of applying composition rules.

classes `A` and `B` are not been formed in the output. Clearly, `S3` is unaffected by any composition, so we will concentrate on the behaviour of `S1` and `S2` only. (2) specifies that `S1` overrides other subjects. When any `f` is called only `S1` contributes to changing `value`. (3) combines all views of `f` and disambiguates the order of operation execution. In (4), the top-level composition rules is specialised by a **merge** to cancel out the effect of **overrideByName** on `f`. A bracket in (5), (6), (7) and (8) affects all places matching the pattern except the subject which is the source of the wrapper. In (5) the sequence of calls is: `S3.A.g`, `S1.A.f`, `S2.A.f` for both `S1_op` and `S2_op`. In (7) the sequence of calls is: `S3.A.g`, `S1.A.f`, `S2.A.f` for `S1_op` but `S1.A.f`, `S2.A.f` for `S2_op`. In (9) to (12) `S2.A.f` is overridden by `S1.A.f`. So calls to either operations will execute only the body of `S1.A.f`.

7.2 A System of Labels

Subject-oriented composition rules are specified in terms of an open and extensible framework known as the system of labels [94, 95]. This Section presents a model for describing the composable elements of subjects and shows how composition rules map on to the model.

A *subject label* is the composition interface of the subject, it contains all information about a subject needed for specifying and carrying out composition. The composition process is a function from the input subject labels to result or output labels. The output subject is created by linking code based on the result label. In order to support the above composition rules, subject labels should contain the following information:

- Classes defined or used by the subject.
- Instance variables including their types.

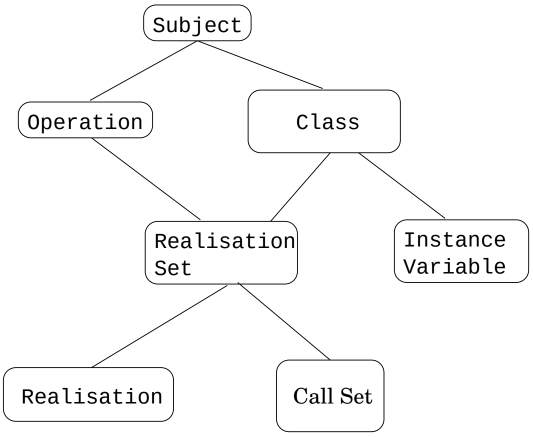

Figure 7-4: Subject label represented as a tree of nodes

- Operations including their signatures.
- Method execution mappings that map operations and classes to method bodies to be executed.
- Method call mappings that map operations and classes to execution mappings.

Two points require clarification: method execution and method call mappings. Method execution mappings describe the effect of method dispatch. In an uncomposed subject a mapping is similar to a method definition in an object-oriented language: a single method body is executed in response to a method call on a receiver. In a subject created by composing other subjects many method bodies may need to be executed in response to a method call. We use the term *realisation* to distinguish an abstraction of a method body from an operation which describes a target for a method call. A *realisation set* contains the realisations to execute in response to a method call on a receiver.

Method call mappings also describe the effect of method dispatch but from the call end. By default a single call will execute the code associated with a single realisation set. But bracket relationships that perform the **call** action described in the last Section cause multiple realisation sets to execute. A *call set* contains the realisation sets to execute in response to a method call.

When implementing concerns, each subject defines its own class hierarchy. Composition of subjects with different class hierarchies is an important part of subject-oriented programming. Inheritance makes sense within a subject but not when multiple subjects are considered together. Also, combination of inheritance hierarchies so as to preserve their effect can lead to cycles [129]. To alleviate these problems, class hierarchies are done away with by flattening or inheritance expansion [96]: all inherited information is made explicit in each class by copying declarations from the class where they are defined to the classes which inherit the declarations.

The subject label can be represented by an abstract syntax tree (AST) of nodes shown in Figure 7-4. A subject label is a collection of operations and classes. Instance variables are nested inside classes. Operations are not nested inside classes but shared between a set of classes based on the subject's inheritance relationships. Realisation sets are nested simultaneously within composable classes and operations, indicating that to gain access to a realisation set we must know its subject, the operation and class name. Realisations are nested inside realisation sets. Perhaps surprisingly, call sets are nested inside realisation sets instead of realisations; after all, calls emanate from method bodies. Call sets do not change the method call at source but act as dictionaries for redirecting control flow to

S :	**subject**
$S.cls$:	**classes**
$S.ops$:	**operations**
$S.map$:	**mappings**
$S.cls.c$:	**class**
$S.cls.c.v$:	**instvar of type** t
$S.ops.o$:	**operation with signature** $\langle t_0, t_1 \ldots t_n \rangle$
$S.map.o.c$:	**realisation set returning** u
$S.map.o.c.r$:	**realisation**
$S.map.o.c.m.r$:	**call set** $\langle \ldots, S.map.m.r, \ldots \rangle$

Figure 7-5: Label clauses.

various realisation sets. The redirection is affected per class or per operation making it appropriate to nest call sets inside realisation sets.

The AST may be an elegant model for representing and searching for composable elements but it is not well suited for explaining composition rules. We would like to use a single notation both for subject labels and for composition concepts. Fortunately, an AST can be equally well represented as a flat set given that fully qualified names are used in place of nesting.

We propose a clausal notation. All activity takes place within the clause universe U. Clauses can be specified in arbitrary order. Label clauses describe both the subjects' composable entities and the output subject. Control clauses describe the details of composition. In the following Subsections we describe the clausal representation of labels and the label composition model.

7.2.1 Clausal Representation of Subject Labels

Subject labels are represented by an unordered list of clauses. An element from the subject label is an attribute-value binding for some compound name:

$$\mathit{CompoundName} : \ \mathit{AttributeName} \ \mathit{OptionalValue}$$

A compound name is a dot-separated list of identifiers. In general, compound names are interpreted hierarchically, where the leftmost element is the most general and the rightmost element is the most specific with dots specifying node tree depth. An attribute name is some identifier. Some attributes have no values while others may be arbitrarily complex. A single compound name can have several attribute-value pairs.

Figure 7-5 shows all clauses necessary to define the composition rules shown in Figure 7-1 on page 134. The root of the AST for a subject is given by the **subject** clause. S is the name of some subject. A subject has three subgroups which are always defined. The *cls* group contains all classes, the *ops* group contains all operations and the *map* group contains all method mappings, i.e. realisation sets and their sub-elements. At the next level down, the **class** clause shows class c in subject S. Instance variables are given as subelements of classes: v is an instance variable of class c in subject S. t is an attribute of the **instvar** clause that denotes the type of v. **operation** clauses have signature types as attributes. t_0 is the type of the return value and t_1 to t_n are the parameter types. In the clause universe all operations have distinct names. This is not a constraint on implementation; rather, the clause universe is an abstraction and the Subject Composer maintains a mapping between actual operation names and their labels in the clause universe.

A **realisation set** clause specifies the result of calling an operation on an instance of a specific class; **realisations** are specified separately. o is the operation and c is the class on which the operation is called. u is the return value specification. For input subjects there is at most one return value but for subjects which are the product of composition the return values must be amalgamated in some way. In general, u is passed to some function f which uses a strategy to select or calculate the return value. The ordering of operation name before class name is not intended to convey nesting. Instead, the o and c components of a realisation set clause form a two-dimensional matrix for selecting a set of realisations and call sets. A **realisation** clause specifies a method body to execute within a realisation set.

A **call set** clause is a subcomponent of its realisation set. The compound name states that realisation set given by $o \times c$ in subject S contains a call to realisation set $m \times r$ where m is the method and r is the class identifier. Subjects are declaratively complete so there is no need to include an additional subject name in the label. The Subject Compiler has proven that realisation set $m \times r$ is defined in S. All method call sites in all operations are nodes because each is a potential join point. Composition rules work at class member level, so for specifying composition rules it is sufficient to know that a realisation set contains a particular call without exposing the details of control flow inside. The **call set** attribute is a sequence of realisation sets to call. This set usually includes $m \times r$. By default, the return value is taken from realisation set $m \times r$.

Control Clauses

The intuitive concepts of grouping elements and applying actions are formalised in the clause universe by control clauses. We distinguish between three kinds of control clauses:

- *Correspondence clauses* directly specify grouping between label clauses.
- *Grouper clauses* are a more powerful way of grouping labels. Groupers are a means of automatically determining correspondences.
- *Combinator clauses* combine the attributes of corresponding clauses and help generate the output clauses.

Top level composition rules are specific collections of attribute combinators and node groupers. Other composition rules directly manipulate correspondence clauses.

7.2.2 Correspondence Clauses

Correspondence clauses specify which labels are to be combined to produce an output label.

$$n : \textbf{composed-of}(q, F)$$

The correspondence clause has three parts. n is the *output label* which will be added into the universe. q is the *input sequence of labels of corresponding elements* whose attributes are to be combined in some way. The order in the sequence sets the order for attributes. F is called the *forwarding set*; it's main role is to realise the concept of forwarding introduced earlier. Forwarding concerns **class** and **operation** labels:

- For classes it describes which of the classes in the input set are to be replaced by the composed class. When a class in the forwarding set is instantiated, an object of the new class in the output label will be created.
- For operations it describes which of the operations in the input sequence get replaced by the output operation. When an operation in the forwarding set is called, all input operations will be executed.

Clearly, an input operation or class must exist in exactly one forwarding set, lest it is ambiguous which operations to execute or which class to instantiate. Overridden elements forward once to a special *null* subject which tells the Subject Composer to unlink the code represented by it. Entities which have no correspondences in other subjects are involved in an identity correspondence. The effect is to forward the node without changes, e.g.:

$$\begin{aligned} \mathit{S.ops.foo}: &\quad \textbf{composed-of}(\langle \mathit{S1.ops.foo}\rangle, \{\mathit{S1.ops.foo}\}) \\ \mathit{S.cls.A}: &\quad \textbf{composed-of}(\langle \mathit{S1.cls.A}\rangle, \{\mathit{S1.cls.A}\}) \end{aligned}$$

For operations and classes, the separation of input labels from the forwarding set facilitates creation of advanced interactions. For example, suppose when creating printing software we have two subjects: `NormalPrint` and `HeaderPrint`. Calling `print(Account, Doc)` on a `Printer` object in `NormalPrint` activates `HeaderPrint`'s method followed by the `NormalPrint`'s method. The former debits the account, prints the summary and account info, and the latter prints the actual document. However, calling `print(Account, Doc)` in the scope of `HeaderPrint`, only prints the account info without debiting the account. A correspondence clause to express this relationship is given by:

$$\begin{aligned} \mathit{TotalPrint.ops.print}: &\quad \textbf{composed-of}(\langle \mathit{HeaderPrint.ops.print}, \mathit{NormalPrint.ops.print}\rangle, \\ &\quad \{\mathit{NormalPrint.ops.print}\}) \end{aligned}$$

The forwarding set is used also with realisation sets to denote those elements that contribute to the return value calculation. For instance, this feature is used to implement the **bracket** rule: only the bracketed methods contribute to the return value. The forwarding set is not used with any other kinds of element.

7.2.3 Groupers

A grouper generates correspondence clauses automatically based on some strategy. A common way of determining correspondences is by name. Elements representing the same concept in different subjects should be grouped together. This stategy is used in both **mergeByName** and **overrideByName** composition rules. Once a grouper completes its work other rules can create alternative correspondence clauses to add to or to replace those created by the grouper.

Figure 7-6 shows a synopsis of groupers used to implement the composition rules in Figure 7-1 on page 134. Groupers **name-match** and **select-first** work in the context of specific constructs. In order to group classes or operations, their subjects must correspond. Instance variables can be grouped only in the context of corresponding classes. **corresponding-rs** is used solely with realisation sets. Realisation sets can be grouped only if their subcomponents are already known to correspond. **name-match**, **select-first** and **corresponding-rs** can be described as symmetric groupers in the sense that they group elements representing the same concept from all input subjects.

- **name-match** Draws into a new correspondence clause those label clauses that have the same last name component.
- **select-first** Draws into a new correspondence clause the input label from the first input subject that contributes to the output label.
- **corresponding-rs** Draws into a new correspondence clause those realisation sets whose constituent classes and operations correspond.
- **bracket-exec** Updates a collection of correspondence clauses including the realisation set of each wrappee with the realisation sets of the wrappers and the classes of each wrappee with the classes of the wrappers.
- **bracket-call** Updates a collection of call sets of correspondence clauses including the call set of each wrappee with calls to the wrappers and the classes of each wrappee with the the classes of the wrappers.

Figure 7-6: Grouper synopsis.

For classes and operations, **name-match** forwards all elements to the output. **select-first** only forwards the overriding element. The overridden elements forward to the special *null* subject. These groupers have no error conditions.

bracket-exec and **bracket-call** groupers generate correspondence clauses for bracket relationships. They create correspondences for method execute and call locations respectively. The activities performed by these groupers can be summarised in terms of the following steps:

1. Bracket relationships use a pattern to specify those locations that should be wrapped. These groupers generate a list of classes containing the join points based on the pattern. In order to prevent recursive bracketing, this list never includes the classes in subjects which are the source of the wrapper methods. Recursive bracketing can occur when both wrappees and wrappers are the same element, and should be prevented.
2. "Clones" of classes containing the wrapper methods are composed with classes containing the bracketed locations. We have put *clone* above in quotes because the appearance of cloning is created using correspondence clauses and there is no explicit clone operator. Figure 7-7 shows the difference between **bracket-exec** and **bracket-call**: the former composes the wrapper class "clone" with the class containing the wrapped operation, while the latter composes the wrapper class "clone" with the class containing the wrapped method call.
3. The wrappees are set to execute around the wrapped elements. For **bracket-exec**, the resolution set of each wrapped method is augmented with the resolution sets denoting the wrappers. For **bracket-call**, the relevant call set of each wrapped realisation set is augmented with new calls to the wrapper realisation sets. All members of wrappee and wrapper classes are set to correspond. "Cloned" wrapper classes' member operations forward to the output operation.

In conjuncton with inheritance the two forms of bracket relationship realised by these two groupers can produce different interactions. **bracket-call** selects call points based on the declared type, so a call will be bracketed irrespective of the dynamic type of the receiver. Thus, **bracket-call** affects all classes below it in the inheritance hierarchy. **bracket-exec** selects execution points by class, so when the pattern specifies the class, classes below it in the hierarchy will not be bracketed.

There are two error conditions for these groupers, both detected during step 1 above:

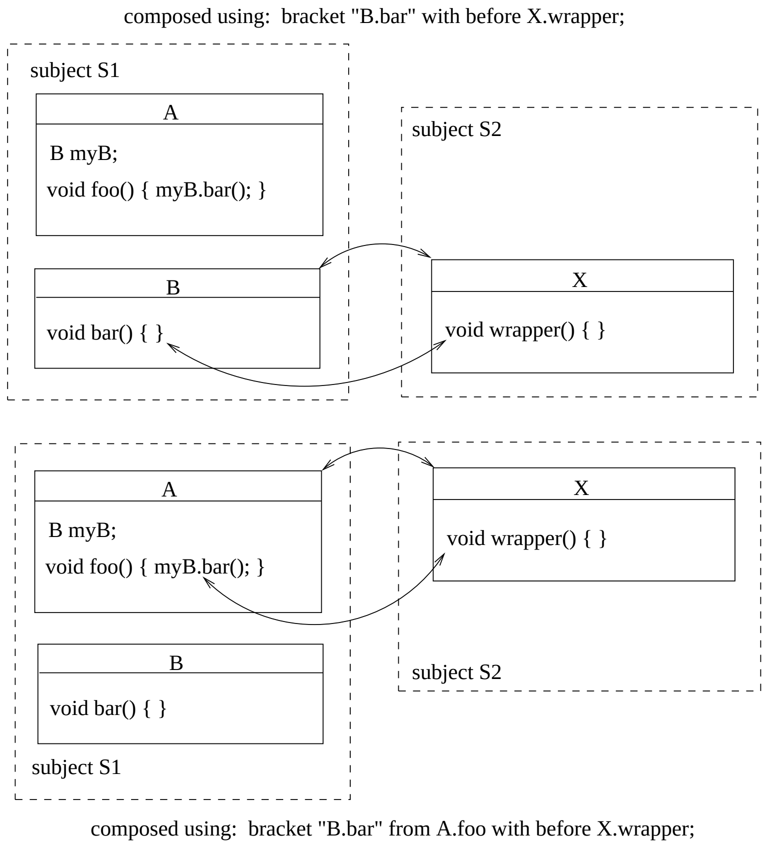
composed using: bracket "B.bar" with before X.wrapper;
subject S1
A
B myB;
void foo() { myB.bar(); }
B
void bar() { }
subject S2
X
void wrapper() { }
A
B myB;
void foo() { myB.bar(); }
B
void bar() { }
subject S1
X
void wrapper() { }
subject S2
composed using: bracket "B.bar" from A.foo with before X.wrapper;

Figure 7-7: Correspondences created by bracket relationships: **bracket-exec** top; **bracket-call** bottom.

- **identity**. The sequence of attributes must be of length 1 or an error is flagged; the output is the attribute of the sole element.
- **equivalent**. All values must be equivalent. The result is the first return value. Equivalence needs to be defined for each type of attribute.
- **first** or **last**. The first or last values is returned from the sequence.
- **union**. Return a set containing the union of all attributes.
- **forwarding-set**[]. This is a parameterised combinator that takes another combinator as a parameter. The return value is selected from the forwarding set of this correspondence clause, e.g. **forwarding-set**[**equivalent**] uses the return value from the first realisation set in the input sequence if all return values in the forwarded realisation sets are equivalent and the forwarding set is not empty.

Figure 7-8: Return value combinators.

- It is an error for the pattern not to match any elements. This is a sanity check that prevents ineffective compositions.
- It is an error for a class containing a wrapper method to have corresponding classes from prior composition rules. Bracketing in conjunction with prior correspondences can create forwarding cycles.

7.2.4 Combinators

Once all correspondences are established, combinators are applied to the attributes of corresponding elements. A combinator determines the output label's attribute from a sequence of input attributes. From Figure 7-5 on page 138 we observe that there are three kinds of attribute to combine:

- The types of corresponding instance variables.
- The sequences of types for signatures of corresponding operations and signatures of operations specified in **call set** clauses.
- The return value of corresponding realisation sets.

In order for instance variables to be composable, either they must have the same type or the clause universe should contain a correspondence clause that forwards all input types to the same output type. In general, in order for operations to be composable they must have equivalent signatures. Exceptions include operations used as wrappers in bracket relationship which may accept meta parameters in the signatures of wrapped operations (e.g. see the example in Figure 3-5 on page 41) and may have `void` return types. Otherwise, signature combination repeatedly applies the type combinator used for variables. For **call set** clauses, the combinator only checks that operations can be combined. Composition aborts if a sequence of input types cannot be combined.

A combinator for realisation sets describes the way the return value should be computed from all corresponding realisation sets, that is, from all return values. The attribute of the output realisation set label is itself a combinator. For instance, in Hyper/J bracket relationship semantics, only the bracketed operations contribute to the return value. Figure 7-8 lists a number of general purpose combinators that can be used in composition rules. Composition aborts if a return value combinator generates an error. The combinator for **call set** label attributes must determine the call set for the output label from a sequence of input call sets.

7.2.5 On the Correctness of the Composition Model

Confidence in the correctness of the system of labels can be attained if it can be shown that for all composition specifications all expressions which may get executed are well formed. Composition does not change realisations but does change what realisations are executed during method dispatch. Expressions are well formed when they have valid targets. Thus for correctness it is sufficient to check that all references extending outside realisations still have valid destinations after composition.

Program created by subject composition will crash if a reference to a class or any of its members cannot be resolved. There are two kinds of problems: either the class or its member cannot be found in the output subject or a reference to an element is ambiguous. Ambiguity results from having more than one choice of class to instantiate or member to access. We can address these problems by enforcing two properties:

- Monotonic composition: no composition rule removes declarations.
- Correct forwarding: each reference to an element in the input must resolve to one element in the output.

The first property ensures that references to classes or members always have targets. To enforce it, all classes, operations and instance variables in input subjects must also exist in the output subject. With respect to forwarding: for classes it ensures that types resolve to a distinct class; for operations it ensures that a method call has one target operation; and for instance variables it ensures that a field reference has a distinct target.

In order to enforce the above properties each kind of realisation artifact that extends links outside evokes constraints on the composition rules that may be created:

- **Types**. Realisations form types from classes. It is important that all types that can be created before composition, can still be created afterward. In the system of labels the relation between input and output types is governed by the input sequences and forwarding sets of correspondence clauses. Monotonic composition is ensured by requiring each input class to be in the input sequence of one or more output classes. By definition, the forwarding set only contains elements in the input sequence. Correct forwarding requires that no input class forwards to *null*. Moreover, for all correspondence clauses containing the class in their input sequence, only one may forward it. By following these principles when devising composition rules the output will contain exactly one class for each input class.
- **Field Access/Update**. The nesting properties of the system of labels ensure that instance variables always correspond in the context of corresponding classes. To ensure both monotonicity and forwarding, for each distinct instance variable in corresponding classes there should be one instance variables in the output class.
- **Method Calls**. In order for a method call to succeed it must have a valid receiver type and method name. There are no constraints on what is executed during a method call. Constraints on class composition ensure that all receiver types remain valid after composition. Monotonic composition requires that for each realisation set in the input subjects there is at least one

- Each input class must appear in exactly one forwarding set of a correspondence clause.
- If class *C* appears in the input set of correspondence clause *S.cls.C'*, then each instance variable *C.v* of *C* must appear in the input set of correspondence clause *S.cls.C'.v'*.
- Each input realisation set must appear in the input sequence of a correspondence clause.
- Each input operation must appear in exactly one forwarding set of a correspondence clause.

Figure 7-9: Correctness properties of control clauses.

realisation set in the output subject. But realisation execution is guarded by the forwarding set of operation clauses, so correct forwarding requires that each distinct input operation appears in exactly one forwarding set. Note that this places no constraints on the output label and an operation may forward to *null*.

In summary, composition rules must have the properties shown in Figure 7-9. The following composition rule definitions enforce the above properties.

7.2.6 Mapping Control Clauses to Composition Rules

In this Section we explain the way in which composition rules are implemented in terms of the system of labels. Rules in the composition specifications are evaluated from the top down. There are two passes: first groupers create and update correspondence clauses, secondly attribute combinators are applied. Composition specifications start with a **mergeByName** or another top level composition rule The other composition rules specify exceptions. **mergeByName** is used with most composition specification and for this reason the composition process is outlined in that Section.

compose

The **compose** directive inserts a single correspondence clause into the clause universe. For example, `compose S1, S2, S3 into S;` inserts:

$$S : \textbf{composed-of}(\langle \mathit{S1}, \mathit{S2}, \mathit{S3} \rangle, \emptyset)$$

equate

Typically, the **equate** directive is used to bring together elements that represent the same concept but have not been brought together by a top level composition rule. For this reason the input elements are symmetrically forwarded to the output. Parameters given to **equate** must come from different (and corresponding) subjects. **equate** is mindful of existing correspondence clauses: it checks if any parameters already participate in composition. There are two error conditions:

- The output element exists but contains none of the input elements. So the present **equate** is in a race with an existing clause.

- An input element already forwards to an output label that is different to that required by **equate**. This is an error because each input element is allowed to forward to at most one output element.

These correctness criteria are the same whether **equate** is applied to classes, operations or variables. For example, suppose that we are parsing `equate S1.A, S2.B into S.C;`, then composition will abort if the clause universe already contains any of:

$$\begin{array}{rl} S.cls.C: & \textbf{composed-of}(\langle S3.cls.E, S4.cls.F\rangle, \{S3.cls.E, S4.cls.F\}) \\ S.cls.D: & \textbf{composed-of}(\langle S1.cls.A, S3.cls.E\rangle, \{S1.cls.A, S3.cls.E\}) \end{array}$$

In the first case above, the output class `C` is already formed by composing classes unrelated to parameters of **equate**. So, the present **equate** is in a race with an existing clause. In the second case, a parameter to **equate** already forwards to a different output label. Each input element is allowed to forward to at most one output element.

equate succeeds in all other cases:

- No prior clauses exist.
- An input element participates in an "identity" correspondence.
- One or more of the input elements already forward to the output element required by **equate**.

For example, in the following, the first two clauses denote "identity" correspondence clauses.

$$\begin{array}{rl} S.cls.A: & \textbf{composed-of}(\langle S1.cls.A\rangle, \{S1.cls.A\}) \\ S.cls.B: & \textbf{composed-of}(\langle S2.cls.B\rangle, \{S2.cls.B\}) \\ S.cls.C: & \textbf{composed-of}(\langle S3.cls.E, S2.cls.B\rangle, \{S3.cls.E, S2.cls.B\}) \end{array}$$

In the third clause, one or more of the parameters to **equate** already participates in a composition. It is acceptable because it allows an omitted element to be added to an existing clause, creating:

$$\begin{array}{rl} S.cls.C: & \textbf{composed-of}(\langle S1.cls.A, S3.cls.E, S2.cls.B\rangle, \\ & \{S1.cls.A, S3.cls.E, S2.cls.B\}) \end{array}$$

For instance variables, **equate** additionally checks that all classes containing the corresponding variables contribute to the same output class, i.e. *contribute* means "appear in the same input sequence". **equate** on operations sets up correspondences for operations, classes and realisation sets. Consider `equate S1.A.foo, S2.B.bar into S.C.fee;`. With no prior clauses, **equate** creates the following correspondence clauses:

$$\begin{array}{ll} S.cls.C: & \textbf{composed-of}(\langle S1.cls.A, S2.cls.B\rangle, \{S1.cls.A, S2.cls.B\}) \\ S.ops.fee: & \textbf{composed-of}(\langle S1.ops.foo, S2.ops.bar\rangle, \{S1.ops.foo, S2.ops.bar\}) \\ S.map.fee.C: & \textbf{composed-of}(\langle S1.map.foo.A, S2.map.bar.B\rangle, \{S1.map.foo.A, S2.map.bar.B\}) \end{array}$$

merge

merge brings together elements and applies the appropriate combinators. For classes, it does the work defined for **equate**. **class** label clauses do not have attributes, so there are no attributes to combine. For instance variables, **merge** additionally checks that all classes containing the corresponding variables contribute to the same output class and applies the type combinator to set the type of the output variable. For operations, **merge** does the work defined for **equate**. The signature combinator creates the signature of the output operation. The attribute of output realisation set clause is set to **forwarding-set**[**equivalent**].

override

override creates correspondence clauses for the overriding elements and deletes the clauses of overridden elements. Consider `override S1.A.foo, S2.B.bar with S1.A.foo into S.C.fee;`. This directive indicates that all calls to `S1.A.foo` and `S2.B.bar` should forward to `S1.A.foo` (which is renamed to `S.C.fee` in the output subject). **override** creates/updates the following correspondence clauses:

$$\begin{array}{ll}
\mathit{S.cls.C}: & \textbf{composed-of}(\langle \mathit{S1.cls.A}, \mathit{S2.cls.B}\rangle, \{\mathit{S1.cls.A}, \mathit{S2.cls.B}\}) \\
\mathit{S.ops.fee}: & \textbf{composed-of}(\langle \mathit{S1.ops.foo}\rangle, \{\mathit{S1.ops.foo}\}) \\
\mathit{null}: & \textbf{composed-of}(\langle \mathit{S2.ops.bar}\rangle, \{\mathit{S2.ops.bar}\}) \\
\mathit{S.map.fee.C}: & \textbf{composed-of}(\langle \mathit{S1.map.foo.A}, \mathit{S2.map.bar.B}\rangle, \{\mathit{S1.map.foo.A}, \mathit{S2.map.bar.B}\})
\end{array}$$

The first clause helps to ensure that operation overriding takes place between corresponding classes. The second clause states "execute `foo` whenever `foo` is requested." The third clause composes into *null* which indicates that no node should be created for the input elements. When running the output program, calls to `bar` will execute no code. The last clause states that only that all input realisation sets contribute to the output subject. In a nutshell, a call to `S2.B.bar` will forward to the elements in the input sequence of `S.map.fee.C` which, through forwarding, executes `S.C.fee`. *S2.ops.bar* forwards to *null*, so no realisations are executed on the behalf of *S2.map.bar.B*.

override has two error conditions that conceptually mirror those of **equate**:

- For the first and second clauses above, no input element may forward to an output that is different to that which is required by **override** parameters. The overridden elements will be reforwarded to *null*. Realisation sets use forwarding clauses for other purposes (see Section 7.2.2 on page 139.

- For the first, second and fourth clause above, if the output label clause required by **equate** exists already, then at least one of the input elements must be related to **override** parameters. This error condition prevents a race for an output label with other rules. The output label *null* is unaffected by races.

When the output labels are created, the signature of the overriding operation is taken as the attribute of the output operation; the signature combinator checks that corresponding operations have compatible signatures. The attribute of output realisation set label is set to **forwarding-set**[**equivalent**].

mergeByName and overrideByName

The **mergeByName** composition strategy specifies that all correspondences should be established based on construct name. Figure 7-10 shows it as a bundle of groupers and combinators to be applied at each kind of node or to its attributes. **overrideByName** also uses name equivalence as

Composition Construct	**Attribute**	**Grouper**	**Combinator**
subject		(explicit match)	
class		**name-match**	
instance variable		**name-match**	
	type		type combinator
operation		**name-match**	
	signature		signature combinator
realisation set		**corresponding-rs**	
	ret. val. spec.		**last**
realisation		**name-match**	
call set		**name-match**	
	call set attrib.		**union**

Figure 7-10: Table showing the elements used in the definition of the **mergeByName** composition rule

Composition Construct	**Attribute**	**Grouper**	**Combinator**
subject		(explicit match)	
class		**name-match**	
instance variable		**name-match**	
	type		type combinator
operation		**select-first**	
	signature		signature combinator
realisation set		**corresponding-rs**	
	ret. val. spec.		**first**
realisation		**name-match**	
call set		**select-first**	
	call set attrib.		**first**

Figure 7-11: Table showing the elements used in the definition of the **overrideByName** composition rule

$S1$:	**subject**
$S1.cls.A$:	**class**
$S1.cls.A.value$:	**instvar of type** `int`
$S1.ops.f$:	**operation with signature** $\langle \texttt{void} \rangle$
$S1.ops.S1\_op$:	**operation with signature** $\langle \texttt{void} \rangle$
$S1.map.f.A$:	**realisation set returning** identity
$S1.map.S1\_op.A$:	**realisation set returning** identity
$S1.map.S1\_op.A.f.A$:	**call set** $\langle S1.map.f.A \rangle$
$S1.map.S1\_op.A.S1\_A\_S1\_op$:	**realisation**
$S1.map.f.A.S1\_A\_f$:	**realisation**
S :	**composed-of**$(\langle S1, S2, S3 \rangle, \emptyset)$
$S.cls.A$:	**composed-of**$(\langle S1.cls.A, S2.cls.A \rangle, \{S1.cls.A, S2.cls.A\})$
$S.cls.A.value$:	**composed-of**$(\langle S1.cls.A.value, S2.cls.A.value \rangle, \emptyset)$
$S.ops.f$:	**composed-of**$(\langle S1.ops.f, S2.ops.f \rangle, \{S1.ops.f, S2.ops.f\})$
$S.map.f.A$:	**composed-of**$(\langle S1.map.f.A, S2.map.f.A \rangle, \{S1.map.f.A, S2.map.f.A\})$
$S.map.f.A.S1\_A\_f$:	**composed-of**$(\langle S1.map.f.A.S1\_A\_f \rangle, \emptyset)$
$S.ops.S1\_op$:	**composed-of**$(\langle S1.ops.S1\_op \rangle, \{S1.ops.S1\_op\})$
$S.map.S1\_op.A$:	**composed-of**$(\langle S1.map.S1\_op.A \rangle, \{S1.map.S1\_op.A\})$
$S.map.S1\_op.A.f.A$:	**composed-of**$(\langle S1.map.S1\_op.A.f.A \rangle, \emptyset)$
...	

Figure 7-12: Label clauses for `S1` and correspondences created by **mergeByName**.

S :	**composed-of**$(\langle S1, S2, S3 \rangle, \emptyset)$
$S.cls.A$:	**composed-of**$(\langle S1.cls.A, S2.cls.A \rangle, \{S1.cls.A, S2.cls.A\})$
$S.cls.A.value$:	**composed-of**$(\langle S1.cls.A.value, S2.cls.A.value \rangle, \emptyset)$
$S.ops.f$:	**composed-of**$(\langle S1.ops.f \rangle, \{S1.ops.f\})$
$null$:	**composed-of**$(\langle S2.ops.f \rangle, \{S2.ops.f\})$
$S.map.f.A$:	**composed-of**$(\langle S1.map.f.A, S2.map.f.A \rangle, \{S1.map.f.A, S2.map.f.A\})$
...	

Figure 7-13: Clauses created by **overrideByName**.

the basis for bringing elements together and Figure 7-11 also shows it as a bundle of groupers and combinators.

A composition specification consisting of a **compose** statement and either **mergeByName** or **overrideByName** establishes correspondences that satisfy the correctness properties given in Figure 7-9 on page 145. Other composition rules preserve the status quo, changing correspondences to preserve the correctness properties.

We will use this opportunity to explain the composition process. Composition is set in motion by specifying corresponding subjects using the **compose** rule. Input subjects are required to have distinct subject names. Composition involves the application of groupers at successively finer levels of construct granularity. Correspondences arising from a top level composition rule are created first. Output labels for realisation sets and call sets are created. These specify how to compose the attributes. Next, the remaining composition rules are applied in the sequence given by the composition specification. These rules create, modify and delete correspondences; and possibly change the combinators set by the top level composition rule. Finally, a walk over the clause universe applies the combinators and creates the output labels. Combinator selection is driven by the type of corresponding elements.

$S3.cls.B$:	**class**
$S3.cls.B.value$:	**instvar of type** `int`
$S3.ops.g$:	**operation with signature** $\langle$`void`$\rangle$
$S1.ops.S3\_op$:	**operation with signature** $\langle$`void`$\rangle$
$S3.map.g.B$:	**realisation set returning** identity
$S3.map.S3\_op.B$:	**realisation set returning** identity
$S3.map.S3\_op.B.g.B$:	**call set** $\langle S3.map.g.B\rangle$
$S3.map.g.B.S3\_B\_g$:	**realisation**
$S3.map.S3\_op.B.S3\_B\_S3\_op$:	**realisation**
$S.cls.A$:	**composed-of**$(\langle S3.cls.B, S1.cls.A, S2.cls.A\rangle, \{S1.cls.A, S2.cls.A\})$
$S.cls.A.value$:	**composed-of**$(\langle S1.cls.A.value, S2.cls.A.value, S3.cls.B.value\rangle, \emptyset)$
$S.ops.f$:	**composed-of**$(\langle S3.ops.g, S1.ops.f, S2.ops.f\rangle, \{S1.ops.f, S2.ops.f\})$
$S.map.f.A$:	**composed-of**$(\langle S3.map.g.B, S1.map.f.A, S2.map.f.A\rangle,$ $\{S1.map.f.A, S2.map.f.A\})$
$S.map.f.A$:	**realisation set returning** forwarding-set[last]
...	

Figure 7-14: Clauses created by a bracket relationship on execute sites.

To exemplify **mergeByName** we return to the composition of subjects in Figure 7-2 on page 135. To top part of Figure 7-12 shows the input label clauses for subject `S1`. The bottom part of Figure 7-12 shows some of the correspondence clauses created by composition (3) in Figure 7-3 on page 136. Note that operations that have no corresponding elements, e.g. `S1.A.S1_op`, are in "identity" correspondences.

Figure 7-13 presents some of the correspondence clauses created by composition (2) in Figure 7-3 on page 136. Note that only overridden **operation** labels forward to the *null* subject. In accordance with the correctness principles, realisation sets, classes and instance variables are unified.

bracket

The two forms of bracket relationship are realised by groupers **bracket-exec** and **bracket-call**. Essentially, there is a one-to-one mapping between a **bracket** composition rule in the composition specification and the groupers. **bracket-call** is selected if the rule has a **from** part.

Figure 7-14 shows some of the correspondence clauses created by composition (5) in Figure 7-3 on page 136. This composition specification first applies **mergeByName** and then a bracket relationship of the **bracket-exec** variety. The top part shows the labels of the wrapper class `B`. The bottom part shows some of the correspondence clauses created by the **bracket-exec** grouper. In the bottom part, the label of the wrapper class is added to the existing correspondence clause. The wrapper class is instantiated when the bracketed classes are instantiated but not vice versa. The members of the wrapper class are set to correspond with the members of the class whose operations are bracketed. Note that the realisation set denoting the wrapper operation is prepended to the sequence of input realisation sets. The order of input realisation sets determines the order of operation execution. The bracket relationship changes the default attribute of the output label for affected realisation sets: wrappers do not contribute to return value calculation.

Figure 7-15 shows some of the correspondence clauses created by composition (7) in Figure 7-3. This composition specification first applies **mergeByName** and then a bracket relationship of the **bracket-call** variety. The Figure shows the correspondence clauses created by the **bracket-call**

$$
\begin{array}{ll}
\mathit{S.cls.A}: & \textbf{composed-of}(\langle \mathit{S3.cls.B}, \mathit{S1.cls.A}, \mathit{S2.cls.A}\rangle, \{\mathit{S1.cls.A}, \mathit{S2.cls.A}\}) \\
\mathit{S.cls.A.value}: & \textbf{composed-of}(\langle \mathit{S1.cls.A.value}, \mathit{S2.cls.A.value}, \mathit{S3.cls.B.value}\rangle, \emptyset) \\
\mathit{S.ops.g}: & \textbf{composed-of}(\langle \mathit{S3.ops.g}\rangle, \{\mathit{S3.ops.g}\}) \\
\mathit{S.ops.S3\_op}: & \textbf{composed-of}(\langle \mathit{S3.ops.S3\_op}\rangle, \{\mathit{S3.ops.S3\_op}\}) \\
\mathit{S.map.S3\_op.A}: & \textbf{composed-of}(\langle \mathit{S3.map.S3\_op.B}\rangle, \{\mathit{S3.map.S3\_op.B}\}) \\
\mathit{S.map.S3\_op.A.g.A}: & \textbf{composed-of}(\langle \mathit{S3.map.S3\_op.B.g.B}\rangle, \emptyset) \\
\mathit{S.map.S1\_op.A.f.A}: & \textbf{call set } \langle \mathit{S3.map.g.B}, \mathit{S1.map.f.A}\rangle \\
\ldots &
\end{array}
$$

Figure 7-15: Clauses created by a bracket relationship on call sites.

grouper. The wrapper class and its members are composed with the class containing the bracketed operations. From the top, class `A` is formed by composing the merged classes with the wrapper class. The `value` fields are grouped into a correspondence clause. The operations `g` and `S3_op`, realisation sets and call set clauses have no name based correspondences, so they get placed into identity correspondence clauses. Finally the call set attribute of the bracketed operation is prepended with a call to the wrapper operation. The order in the call set sequence denotes the call order for calling operations, ensuring that the wrapper is called before the wrappee.

order

Ordering of behaviours is significant and can change the overall effect of composed operations. Bracket relationships imply an order but the **unify** family of composition rules, which include **merge** and **mergeByName**, do not. The **order** directive performs pair wise modification of input sequences of realisation set clauses. It allows the order of execution to be set where the order is significant. Without **order**, the sequence of execution cannot be assumed. Compositions (3) to (8) in Figure 7-3 on page 136 use **order** to disambiguate the sequence in which merged operations are executed. This rule will fail if the clause universe does not contain the elements given by parameters.

7.2.7 Definitions

The following Sections will define groupers, type combinators, etc that rely on functions for manipulating the clause universe U in following ways:

- Finding all clauses matching some pattern.
- Replacing a label clause attribute with a new value or overwriting one clause by another.
- Extracting information from a compound name of a label clause.

A label clause has been defined as an attribute-value binding for a name. The label is identified by a compound name; it is a list of identifiers separated by dots:

$$\mathit{name}_0.\mathit{name}_1.\mathit{name}_2 \ldots \mathit{name}_n : \text{ Attribute description and values}$$

Figure 7-16 shows a list of functions. **match** and **extract** use underscores as non-null wildcards. Other names are matched exactly. For example:

- **match**($\mathit{S.cls.\_}$) is the set of class labels in subject S.

- **match**(n). Searches U for label clauses that match the compound name pattern n. Returns a set of such elements which may be empty.
- **replace**(x, y) or **replace**$(x\$t, x\$t')$. Replaces clause x by clause y in U, or replaces attribute t of clause x by attribute t' and clause y. x is discarded. Clause x must exist in U or there is an error condition.
- **extract**$_p(n)$. Filters compound name n based on pattern p. The pattern is a dotted expression of a form similar to a compound name which uses the question mark to denote the name to extract. Other names are used for pattern matching. It is an error for the compound name not to match the pattern.
- **dt**(t). Extracts the class name from type t when t has form $n_1.n_2.n_3\langle p_1 \ldots p_k\rangle$. $n_1.n_2.n_3$ is the compound name of a **class** label. **dt** is defined as:

  $$\mathbf{dt}(n_1.n_2.n_3\langle p_1 \ldots p_k\rangle) \overset{\text{def}}{=} n_1.n_2.n_3$$

- **forwards-to**(l). Searches U for correspondence clauses and returns the output label clause to which l forwards. In a well formed clause universe, each input **class** or **operation** label forwards to one output label or *null*:

  $$\textbf{forwards-to}(l) \overset{\text{def}}{=} n \textbf{ where } n : \textbf{composed-of}(q, F) \in U \land l \in F$$

Figure 7-16: Functions used in the definition of composition directives.

- **match**$(S.\mathit{cls}.\_.v)$ is the set of instance variable labels in subject S of classes that define instance variable v.
- **match**$(\_.\mathit{map}.\_.\_)$ is the set of all realisation sets in all subjects in U.

A well formed pattern for **extract** has exactly one question mark. Trailing underscores can be omitted.

- **extract**$_{?.\mathit{cls}}(S.\mathit{cls}.c.v) = S$. The *cls* in the pattern matches *cls* in the corresponding position in the compound name. This pattern can also be specified as $?.\mathit{cls}.\_.\_$.
- **extract**$_{\_.\mathit{ops}.?}(S.\mathit{ops}.o) = o$. The underscore indicates that we do not care about the name of the first element. The second element must be *ops*.
- **extract**$_{\_.\mathit{map}.\_.?}(S.\mathit{cls}.c.v)$ is an error. The *map* name does not match the name in the corresponding position of the compound name.

replace is commonly used with correspondence clauses to modify the input sequence and to change the attribute of an existing label clause, for example:

- **replace**$(c : \textbf{composed-of}(q, F),\ c : \textbf{composed-of}(q', F'))$ replaces q by q' and F by F' for the correspondence clause associated with label c.
- **replace**$(d : \textbf{instvar of type } t,\ d : \textbf{instvar of type } t')$ overwrites type t by t' for instance variable label d.

7.3 Grouper Definitions

In this Section we define groupers used in the specification of top level composition rules **mergeByName** and **overrideByName**. Groupers generate composition clauses either automatically based on the labels in the clause universe or based on the parameters. There are five groupers to present. Groupers **name-match**, **select-first** and **corresponding-rs** are the "automatic" groupers representing prior art [94]. **bracket-exec** and **bracket-call** accept parameters consisting of a pattern specifying points to bracket and the wrapper operations. Additionally, **bracket-call** takes a **from** parameter constraining the call points.

7.3.1 Name Matching

Grouper **name-match**(n, Q) generates correspondence clauses based on name equivalence. As input it takes label n denoting a prefix for the output label and sequence Q of sets containing the elements from which correspondences will be drawn. Name matching creates correspondences for sets of classes, instance variables and operations, etc. It works by checking for equivalence in the last name component of the elements' compound names. Elements with equivalent names in the last component are drawn into a correspondence clause. Elements that have no corresponding counterparts in other sets are put into identity correspondence clauses.

Definition: (Name Matching) For prefix n specifying the node type and sequence of sets $Q \equiv \langle S_1 \dots S_k \rangle$, the name matching grouper is defined as:

$$\begin{aligned}\textbf{name-match}(n, Q) \overset{\text{def}}{=} \; & \{n.x : \textbf{composed-of}(\langle n_1.x \dots n_k.x \rangle, \{n_1.x \dots n_k.x\}) \\ & \text{where } x \text{ is a distinct last name component of} \\ & \text{at least one set in } Q : \; \exists i \in [1, k], n_i.x \in S_i\end{aligned}$$

name-match also creates the forwarding set as the following example demonstrates. The forwarding set is used with operations and classes only:

$$\textbf{name-match}(S.ops, \langle \{S1.ops.fn1, S1.ops.fn2, S1.ops.fn3\}, \{S2.ops.fn1, S2.ops.fn2\} \rangle)$$

$$\begin{array}{ll} S.ops.fn1 : & \textbf{composed-of}(\langle S1.ops.fn1, S2.ops.fn1 \rangle, \{S1.ops.fn1, S2.ops.fn1\}) \\ S.ops.fn2 : & \textbf{composed-of}(\langle S1.ops.fn2, S2.ops.fn2 \rangle, \{S1.ops.fn2, S2.ops.fn2\}) \\ S.ops.fn3 : & \textbf{composed-of}(\langle S1.ops.fn3 \rangle, \{S1.ops.fn3\}) \end{array}$$

7.3.2 Selection

For **overrideByName** the output contains elements taken from the first set in Q. No pattern matching is required. Overridden elements are forwarded to the *null* subject which indicates to the Subject Composer that no code should be generated for this node. This is specified as:

Definition: (Select First) For prefix n specifying the node type and sequence of subjects $Q \equiv$

$\langle S_1 \ldots S_k \rangle$ the selection grouper is defined as:

$$\begin{aligned}\textbf{select-first}(n, Q) \overset{\text{def}}{=} \; & \{n.x : \ \textbf{composed-of}(\langle n_1.x \rangle, \{n_1.x\}) \\ & \mathit{null} : \ \textbf{composed-of}(\langle n_2.x \ldots n_k.x \rangle, \{n_2.x \ldots n_k.x\})\} \\ & \text{where } x \text{ is a distinct last name component of} \\ & \text{the first set of } Q : \ n_1 \in \textbf{head}(Q)\end{aligned}$$

7.3.3 Correspondence Matching

For grouping realisation sets, a special **corresponding-rs** grouper is specified. Realisation sets correspond when their constituent classes and operations correspond. That is, in order to establish realisation set correspondence, prior correspondence should exist between classes and operations.

Definition: (Corresponding Realisation Sets) For prefix n of the form $S.map$ and sequence of subjects $Q \equiv \langle S_1 \ldots S_k \rangle$, the grouper for realisation sets is defined as:

$$\begin{aligned}\textbf{corresponding-rs} \overset{\text{def}}{=} \; & \{n.o.c : \ \textbf{composed-of}(\langle n_1.o.c \ldots n_k.o.c \rangle, \{n_1.o.c \ldots n_k.o.c\})\} \\ & \text{where } o, c \text{ are distinct operation and class names} \\ & \text{and the clause universe contains:} \\ & \quad S.ops.o : \ \textbf{composed-of}(q_o, F_o) \\ & \quad S.cls.c : \ \textbf{composed-of}(q_c, F_c) \\ & \text{for some } o, c, q_o, F_o, q_c, F_c, \text{ such that} \\ & \quad S_i.ops.o_i \in q_o \ \wedge \ S_i.cls.c_i \in q_c \quad \forall i \in [1, k]\end{aligned}$$

7.3.4 Grouper For Execute Sites in Bracket Relationships

The **bracket-exec** grouper is a control clause. It sets up correspondences that realise the bracket relationship which wraps method execution sites in classes. One control clause is placed into the clause universe for each bracket relationship in the composition specification. It is envisaged that multiple grouper instances affecting overlapping sets of bracketed locations will be applied. The order in which the groupers are evaluated sets the order of wrapper method execution. The earlier ones are executed closer to the bracketed location. The activity of **bracket-exec** can be summarised as follows:

1. Identify the pertinent realisation set clauses representing the locations to bracket.
2. Add to them the realisation sets denoting the wrapper operations.
3. Compose into each class containing bracketed operations the elements of each class containing the wrappers, thereby "cloning" the wrapper classes.

This control clause has the following form:

$$\textbf{bracket-exec}(n, \langle p, \mathit{before}, \mathit{after} \rangle)$$

The first parameter n is the label of the output subject. The second parameter is a 3-tuple where p is the match pattern, *before* and *after* are the realisation set labels for the **before** and **after** wrappers. At most one of *before* and *after* can be `null`.

The pattern p matches realisation set clauses in the output subject n. For example, suppose we wish to bracket all operations matching `‘‘C.*’’`, i.e. execution of any operation in class `C`. The following set contains the realisation set labels in U described by this pattern:

$$\{c \mid c \in \mathbf{match}(n.map.\_.C)\}$$

Pattern are specified as regular expressions which **bracket-exec** expands into a set of matches in U. `‘‘*.*’’` matches all methods in all classes. Let $p_1 \ldots p_k$ be the compound names for realisation sets in subject n created from pattern p.

before and *after* are realisation set labels from an input subject. Using **forwards-to** and **extract** we construct output subject labels for these realisation sets. Let M be those realisation sets that should be bracketed:

$$\begin{array}{rcl} M & = & \{p_1 \ldots p_k\} \setminus \\ & & \{n.map.\mathbf{forwards\text{-}to}(\mathbf{extract}_{\_.map.?}(\mathit{before})).\mathbf{forwards\text{-}to}(\mathbf{extract}_{\_.map.\_.?}(\mathit{before})), \\ & & \mathbf{forwards\text{-}to}(n.map.\mathbf{extract}_{\_.map.?}(\mathit{after})).\mathbf{forwards\text{-}to}(\mathbf{extract}_{\_.map.\_.?}(\mathit{after}))\} \end{array}$$

For each entry in M, **bracket-exec** replaces the correspondence clause in U with a new correspondence clause containing the updated input sequence q. The forwarding set, describing the elements of q which contribute to return value computation, is unchanged.

$$\mathbf{replace}(m_i : \ \mathbf{composed\text{-}of}(q, F), m_i : \ \mathbf{composed\text{-}of}(\langle \mathit{before}, q, \mathit{after} \rangle, F)) \quad \forall m_i \in M$$

Next, the classes containing the brackets and their members are introduced into the classes containing bracketed locations. M_c, M_s and M_o respectively specify the **class**, **instvar** and **operation** labels that require changing:

$$\begin{array}{rcl} M_c & = & \{\mathbf{match}(n.cls.\mathbf{extract}_{n.map.\_.?}(m_i)) \mid \forall m_i \in M\} \\ M_s & = & \{\mathbf{match}(n.cls.\mathbf{extract}_{n.map.\_.?.\_}(m_i)) \mid \forall m_i \in M\} \\ M_o & = & \{\mathbf{match}(n.ops.\mathbf{extract}_{n.map.?}(m_i)) \mid \forall m_i \in M\} \end{array}$$

Figure 7-17 shows the clause universe being updated with new clauses containing the elements from the wrapper classes. **replace** is applied for each $m_c \in M_c$, $m_s \in M_s$ and $m_o \in M_o$. Note that the forwarding sets for classes and operations are unchanged. For classes, the wrapper class is instantiated only when the wrappee class is instantiated. For operations, *before* and *after* are called only when the bracketed operation is invoked. This property ensures that forwarding is done correctly. At this point the work of **bracket-exec** is complete.

7.3.5 Grouper for Call Sites in Bracket Relationships

The **bracket-call** grouper is a control clause that sets up correspondences that realise the bracket relationship which wraps methods at the call point. For each bracket relationship containing a `from`

$$\begin{array}{l}
\mathbf{replace}(m_c : \mathbf{composed\text{-}of}(q, F), \\
\qquad m_c : \mathbf{composed\text{-}of}(\langle \mathbf{extract}_{?.map}(\textit{before}).\textit{cls}.\mathbf{extract}_{\_.map.\_.?}(\textit{before}), \\
\qquad\qquad q, \\
\qquad\qquad \mathbf{extract}_{?.map}(\textit{after}).\textit{cls}.\mathbf{extract}_{\_.map.\_.?}(\textit{after})\rangle, \\
\qquad\qquad F))
\end{array}$$

$$\begin{array}{l}
\text{Let } v = \mathbf{extract}_{\_.cls.\_.?}(m_s) \text{ in} \\
\mathbf{replace}(m_s : \mathbf{composed\text{-}of}(q, F), \\
\qquad m_s : \mathbf{composed\text{-}of}(\langle \mathbf{extract}_{?.map}(\textit{before}).\textit{cls}.\mathbf{extract}_{\_.map.\_.?}(\textit{before}).v, \\
\qquad\qquad q, \\
\qquad\qquad \mathbf{extract}_{?.map}(\textit{after}).\textit{cls}.\mathbf{extract}_{\_.map.\_.?}(\textit{after}).v\rangle, \\
\qquad\qquad \emptyset))
\end{array}$$

$$\begin{array}{l}
\mathbf{replace}(m_o : \mathbf{composed\text{-}of}(q, F), \\
\qquad m_o : \mathbf{composed\text{-}of}(\langle \mathbf{extract}_{?.map}(\textit{before}).\textit{ops}.\mathbf{extract}_{\_.map.?}(\textit{before}), \\
\qquad\qquad q, \\
\qquad\qquad \mathbf{extract}_{?.map}(\textit{after}).\textit{ops}.\mathbf{extract}_{\_.map.?}(\textit{after})\rangle, \\
\qquad\qquad F))
\end{array}$$

Figure 7-17: Updating existing clauses with correspondences from the wrapper class.

clause in the composition specification, a **bracket-call** control clause is placed into the universe. Multiple groupers may be applied to a possibly overlapping set of call points. The order in which **bracket-call** groupers are processed sets the order of wrapper method invocation. The activity of this clause can be summarised as follows:

1. Use the pattern to identify the call set clauses whose attributes will be extended with the wrappers.
2. Add the realisation sets denoting the wrappers to each call set label attribute.
3. Compose into each class containing a bracketed call set the elements of each class containing the wrappers.

This control clause has the following form:

$$\mathbf{bracket\text{-}call}(n, \langle p, f, \textit{before}, \textit{after}\rangle)$$

The first parameter n is the name of the output subject. The second parameter is a 4-tuple where p is the pattern describing the methods which should be bracketed, f is a list of either class or operation labels describing the call points, *before* and *after* are the realisation sets of wrapper methods. The return value from the bracketed method call is passed back to the calling context and the return values from wrapper calls are discarded. At most one of *before* and *after* can be null.

Let $p_1 \ldots p_k$ be the realisation set labels matched by pattern p. In order to select the call points, we determine set M of realisation set labels matched by p. To prevent recursive bracketing, this set does not include the realisations denoting the wrappers:

$$\begin{aligned}
M \quad = \quad & \{p_1 \ldots p_k\} \setminus \\
& \{n.map.\mathbf{forwards\text{-}to}(\mathbf{extract}_{\_.map.?}(\textit{before})).\mathbf{forwards\text{-}to}(\mathbf{extract}_{\_.map.\_.?}(\textit{before})), \\
& \mathbf{forwards\text{-}to}(n.map.\mathbf{extract}_{\_.map.?}(\textit{after})).\mathbf{forwards\text{-}to}(\mathbf{extract}_{\_.map.\_.?}(\textit{after}))\}
\end{aligned}$$

Next, we search U for call set clauses that describe calls to nodes in M. Set D contains the call set labels which should be bracketed:

$$\begin{aligned} D \quad = \quad & \{\mathbf{match}(n.map.\_.\_.\mathbf{extract}_{\_.ops.?}(f').\_) \vee \\ & \mathbf{match}(n.map.\_.\_.\_.\mathbf{extract}_{\_.cls.?}(f')) \mid f' \in f\} \end{aligned}$$

The attribute of each $d \in D$ is extended with *before* and *after* realisation sets:

$$\begin{aligned} & \mathbf{replace}(d\$cs, d\$\langle \mathit{before}, d\$cs, \mathit{after}\rangle) \\ & \text{where } d\$cs \text{ is the attribute of a call set clause } d \end{aligned}$$

The final step is the same as for **bracket-exec**. Each class containing a wrapper method and its members is introduced into the classes containing bracketed call points. M_c, M_s and M_o respectively specify the **class**, **instvar** and **operation** labels that require changing:

$$\begin{aligned} M_c \quad &= \quad \{\mathbf{match}(n.cls.\mathbf{extract}_{n.map.\_.?}(d)) \mid \forall d \in D\} \\ M_s \quad &= \quad \{\mathbf{match}(n.cls.\mathbf{extract}_{n.map.\_.?.\_}(d)) \mid \forall d \in D\} \\ M_o \quad &= \quad \{\mathbf{match}(n.ops.\mathbf{extract}_{n.map.?}(d)) \mid \forall d \in D\} \end{aligned}$$

Figure 7-17 shows the clause universe being updated with new clauses containing the elements from the wrapper classes. **replace** is applied for each $m_c \in M_c$, $m_s \in M_s$ and $m_o \in M_o$. At this point the work of **bracket-call** is complete.

7.4 The Model of Type Composition

In this Section we introduce the model for composing subjects annotated with Subjective Ownership Types. We will argue that subject composition based on our notion of type equivalence leads to deep ownership in the output subject.

The aim of subject composition is to create a program that combines the functionality of its input subjects in a useful way. The role of SAPS is to clarify subject interaction by constraining aliasing in a multi subject environment. So SAPS should enable the ownership properties of the output subject to be determinable from the inputs and the composition specification.

The model we propose is one where composition preserves the ownership properties of its input subject. Subjects may be composed using the rules we have described so long as the ownership properties of each input subject continue to hold. We intend for all subjective ownership type declarations in each input subject to stay true after composition for all valid compositions. On the positive side, this model leads to a nice property that every object keeps its representation context; representation containment is preserved and no object is exposed outside its owner. On the negative side, this model requires subjects with inherently different ownership properties to compromise on a common ownership structure. The consequences of this composition model for subject-oriented programming are evaluated in Chapter 8.

Our model depends on two factors presented over the following two Subsections:

- A means of determining equivalent subjective ownership types across the input subjects.
- SOT-aware composition rules that preserve each subject's ownership properties.

7.4.1 Subjective Ownership Type Equivalence

In Subject-Oriented Programming the composable elements of subjects are brought together by defining a correspondence between them. Corresponding elements are composed into a single element in the output. The attributes of corresponding elements are combined to create the attribute of the output element. Types are attributes of both field variables and operations. SOP only permits composition of elements with equivalent types. In the case of value types equivalence is observable immediately, e.g. composition of `int` type variables is allowed but composition of variables of type `int` and `float` is not allowed. For abstract types, equivalence means either that corresponding elements have types common to all subjects or type equivalence results from class correspondence. Common types are generated from classes which are defined in class libraries and imported into each subject that uses them.

Subjective Ownership Types are an extension to existing type declarations of an object-oriented language. All elements with value types have global aliasing properties and require no additional type checking. Abstract types are derived from composable and uncomposable classes. The data component of a type is followed by a sequence consisting of the owner context and other identifiers which bind the ownership parameters in the uncomposable class declaration.

For types derived from composable classes, data type equivalence usually is inferred from the composition specification. By definition, uncomposable classes cannot be composed. Uncomposable classes are defined in libraries and imported into all subjects that need them. For types derived from uncomposable classes, data type equivalence is observable immediately. Context identifiers appear in a sequence after the data component. Suppose a number of such sequences are combined. Clearly, all sequences are of the same length: atomic types have no context identifiers; abstract data types derive from the same uncomposable class and thus require the same number of contexts in all subjects; all types derived from composable classes require exactly one context representing the owner. The equivalence in the context component of a type is checked one position at a time. There are three possible combinations of contexts:

1. Explicit context combined with an explicit context.
2. Explicit context combined with an unknown context.
3. Unknown context combined with another unknown context.

In Figure 7-18, context correspondences labelled (1), (2) and (3) relate to the points in the enumerated list above. For explicit contexts, equivalence is observable immediately from the representation. Case (1) in Figure 7-18 shows that subjects `S1` and `S2` both view `A.h` as being owned by its container, an object of class `A`. In case (2) an `exp` combines with an `unk` in class `A`. The `unk` assumes the value of the `exp` in `A`, producing a *resolution.* In the Figure, unk_k resolves to exp_1 in class `A`. In case (3), when all corresponding contexts are unknown, no resolutions occur but the composition specification must infer a correspondence between those `unks`. The composition specification must specify that unk_m in `S1.A` corresponds with unk_m in `S2.A`. This is indeed the case in Figure 7-18: the **mergeByName** composition strategy will create a correspondence for unk_m.

unk Resolution

In principle, a subject composition need not resolve all `unks` appearing in all composed subjects, and each composition can resolve more and more `unks`. Section 9.2 on page 215 will show that partial

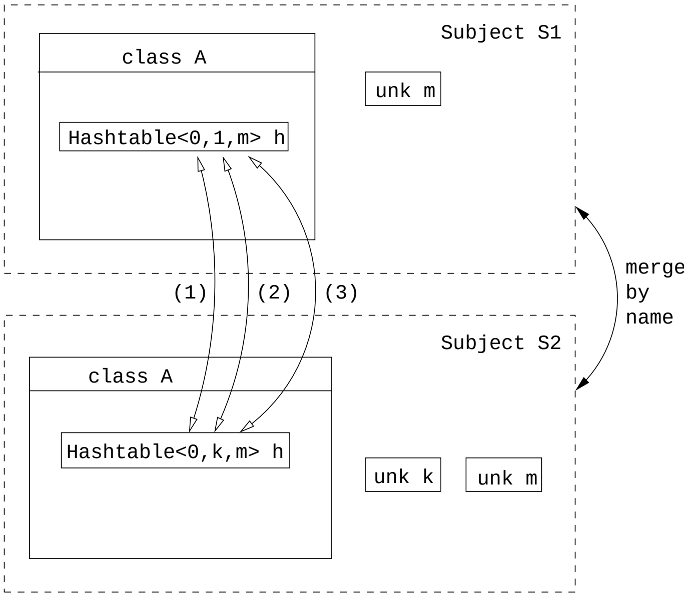

Figure 7-18: Context correspondences

resolution plays an important role in the future development of SAPS. However, presently we require that:

1. Every context combination yields a resolution.
2. Subject composition resolves all `unks` in all input subjects. An `unk` must resolve in every class where it may be observed in the type of a declaration, an expression or subexpression.

Resolutions can occur in two ways:

- Directly through combination of types of corresponding elements. When a resolution occurs, the Subject Composer creates a resolution mapping ρ which maps class name, `unk` pairs to `exp` values:
$$\rho \overset{\text{def}}{=} \{(C,k) \mapsto n\}^*$$
- Indirectly using association and inheritance relationships between classes. Associations are formalised in code by field accesses, updates and method calls. Associations and inheritance can propagate a resolution between classes. Section 7.6 on page 170 will present the resolution validation algorithm which consumes a resolution mapping and attempts to resolve `unks` in all classes.

Direct and indirect resolution is shown in Figure 7-19. Suppose that subjects `S1` and `S2` are merged by name. Classes `S1.A` and `S2.A` correspond, and so do instance variables `S1.A.c` and `S2.A.c`. Instance variable correspondence produces resolution $(\mathtt{A}, k) \mapsto 1$. However, class `S1.B` also has a construct whose type depends on $\mathtt{unk}_k$ which is not resolved by composition. Composition of `S1` and `S2` cannot resolve $\mathtt{unk}_k$ in `B` because `S2` has no concept of this kind. These sort of differences are totally consistent with subject-oriented development: each subject should only define concepts that serve to address its concern. Indirect resolution uses the association between classes `S1.A` and `S1.B` to propagate the resolution from `S1.A` to `S1.B`. `S1.A.foo` defines a field update expression

```
subject S1 {
  unk k;
  class A where 1 <= k {
    B<1> b;
    C<k> c;
    void foo() { b.c = c; }
  }
  class B where 1 <= k {
    C<k> c;
  }
  class C { }
}
subject S2 {
  class A {
    C<1> c;
    D<1> d;
    void bar() { c.d = d; }
  }
  class C {
    D<1> d;
  }
  class D { }
}
```

Figure 7-19: Example showing direct and indirect unk resolution

which associates classes A and B. Feeding the resolved values into Δ_1 (defined in Section 6.2.2 on page 107) yields resolution $(\mathtt{B}, k) \mapsto 1$.

Indirect resolution is meaningful because most subjects implement collaborations. Objects collaborate by sending each other messages containing references to other objects. Indirect resolution depends on the fact that at runtime field access, update and method call expressions create a link between two references to the same object. It then uses the principles of explicit context identifier arithmetic to calculate the correct type at the other end of the association. Thus, for subjects implementing a single collaboration, a single direct resolution may be sufficient to resolve the unknown context for all classes. Other subjects may require multiple direct resolutions to achieve subject-wide resolution.

Resolution Constraints

unks often have resolution constraints in the form of ucircs. Conceptually, resolution constraints ensure that in each class an unk denotes a range of exps and no resolution in the valid range causes representation exposure. Recall that at subject level, ucircs specify inter-unk ordering. Given declaration `ucirc k <= m` and some class A where the value of unk$_k$ and unk$_m$ can be observed, any resolution for unk$_k$ and unk$_m$ in A must satisfy `k <= m`. At class level, ucircs are stated in where clauses. These specify a range to which an unk must resolve in that class. Given declaration `class B where 1 <= k`, unk$_k$ must resolve to a value greater than exp$_0$ in B and its subclasses.

Resolution constraints are important during composition. After all, this is the time when unks are replaced by exps. However, the declared ucircs are not suitable for this purpose. Composition changes the make-up of a class, introducing new instance variables and changing operation behaviour. For instance, the **override** composition rule selects one method definition over a number of others. The resolution constraints required by the overriding expression are likely to be different

```
subject S1 {
  class TitleBar {
    Widget<O> w;
  }
  ...
}
subject S2 {
  class Window {
    Widget<O> w;
  }
  ...
}
```

Figure 7-20: Example for SOT-aware composition rules.

to the constraints of overridden operations. Consequently, to ensure valid resolution the resolution constraints pertaining to a class must be inferred from its declarations and definitions in the *output*.

7.4.2 SOT-Aware Composition Rules

Composition rules are defined in terms of element grouping and attribute combination. The preceding Subsection was concerned with combination of type attributes. Presently we are concerned with element grouping. There are two points which concern grouping: first, we argue that the composition rules in this Chapter do not cause representation exposure; secondly, we take a look at bracket relationships in the contexts of SAPS.

Composition Rules and Representation Exposure

We require subject composition to preserve the ownership properties of input subjects. We propose that in order to do so, *all class member grouping should take place in the scope of grouped classes.* Given type equivalence, this property ensures that for all objects, a representation object in one subject is treated as representation in all other subjects.

The SAPS model has two kinds of class member: instance variables and operations. The structural properties of the system of labels ensure that instance variables can correspond only within corresponding classes. To demonstrate this point, consider Figure 7-20. In order for `S1.TitleBar.w` and `S2.Window.w` to ever reference the same `Widget` object, `S1.TitleBar` and `S2.Window` must correspond. If both subjects type checked correctly before composition, then we can be certain that no subject exposes the `Widget` object outside its representation context.

Structural properties alone are not enough to ensure that newly introduced behaviour does not cause representation exposure – this is the domain of composition rules. In the system of labels, behaviour is abstracted by realisation labels. All is safe while classes execute realisations sourced from their own subject. Representation exposure can be caused by an *external* realisation that finds its way into the set by composition. So, this problem can be addressed if composition rules control what realisations get executed.

In the system of labels, realisations are nested inside realisation sets. By definition of realisation set, during method dispatch on a receiver all realisations in a realisation set will be executed. If that realisation set is itself composed of others, then all input realisation sets will contribute to the set of executed realisations. But, access to realisations is guarded by (**operation**, **class**) label pairs.

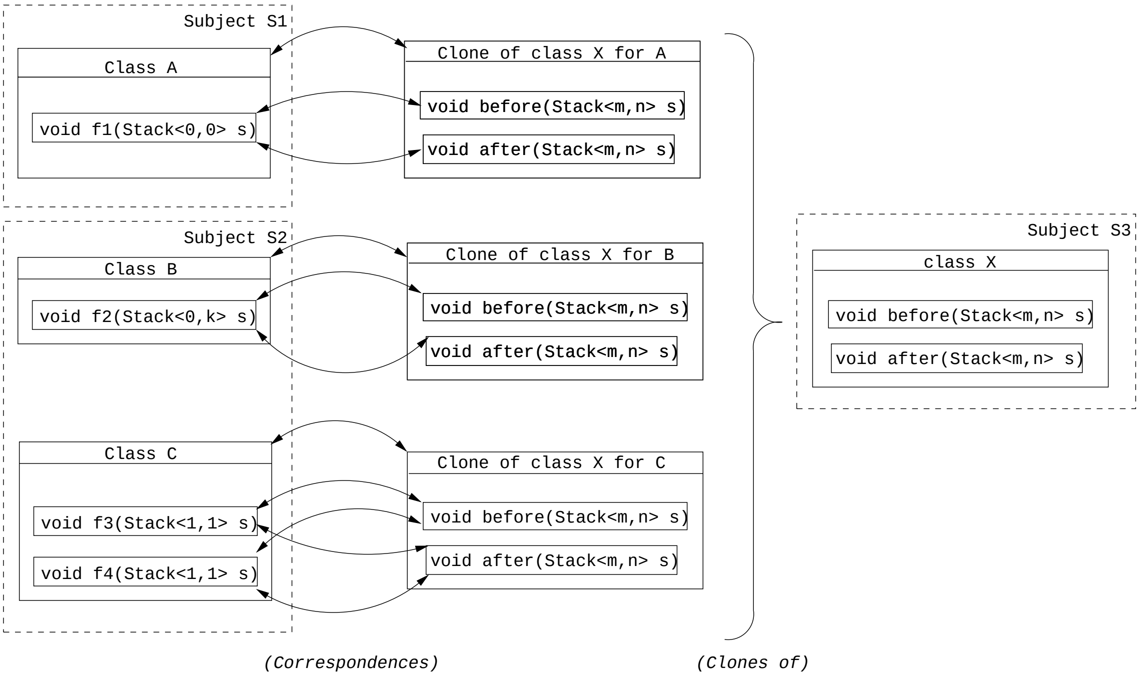

Figure 7-21: The effect of bracket relationships on unk resolution

It follows that the executed realisations will come from some input class. Consequently, in order to prevent representation exposure it is necessary for classes guarding that behaviour to be composed.

This property is true of all valid combinations of composition rules that we have specified. Top level composition rules use **name-match** to group classes. All finer grained composition may add but never delete classes from correspondence clauses.

Bracket Relationships and SAPS

Recall that bracket relationships are an asymmetric subject composition mechanism that enables the behaviour defined in one subject to extend the behaviour defined in another. Bracket relationships take a pattern parameter which expands into a list of classes containing the join points. The bracket relationship composes the classes containing the wrapper methods with each class in the list. The wrapper methods are set to execute around the bracketed points. Bracket relationships create the effect of cloning, with the bracketed locations and wrapper methods coming from corresponding classes. It follows that the type combination principles apply also to bracket relationships: the type in the interface of the wrappers must be equivalent to the type of the bracketed locations.

Operations that play the role of wrapper methods can use exps and unks in the types of their parameters. By using unks, the aspectual subject can adapt to the contexts appearing in the bracketed locations. Recall that an unk resolves to one exp per class. The resolution is well-defined if for multiple bracketed locations within one class, each join point resolves to equivalent exps. Figure 7-21 shows the effect of bracket relationships. Class X is “cloned” once for each class containing bracketed locations. The unks in the wrapper methods of “cloned” classes correspond with the contexts (both explicit and unknown) in the bracketed sites, producing the following resolutions and unk correspondences:

context\class	A	B	C
m	0	0	1
n	0	k	1

Section 7.2.3 on page 140 described the conceptual difference between its two forms. Both forms establish correspondences between the wrappee classes and the wrapper classes, in effect merging the classes of the wrappers with each class of the wrappee. **bracket-exec** creates correspondences at the receiver of the call whereas **bracket-call** creates correspondences at the class containing the call. Both forms of bracket relationship lead to type combination and, hence, `unk` resolution Technically, **bracket-exec** and **bracket-call** differ in terms of where resolutions occur. In **bracket-exec** resolutions occur in the class containing the bracketed realisation set(s). In **bracket-call** resolutions occur in the class containing the bracketed call set(s).

The SAPS notion of composable and uncomposable classes adds a new dimension to the way the two forms are understood. In SAPS only **bracket-call** can be used with bracket operations of uncomposable classes. There are two reasons for disallowing **bracket-exec** on operations of uncomposable classes:

- Uncomposable classes are black-boxes. **bracket-exec** implies that all calls to an object of an uncomposable class should be extended with additional behaviour. This is a "static extension" – an extension that affects all existing clients. Inheritance is black-box extension mechanism that should be used to specialise an uncomposable class, i.e. a black-box.

- **bracket-exec** implies "bracket calls to all objects of this kind". **bracket-call** allows for a more accurate specification of intent by implying "bracket calls to objects of this kind from this set of call points". SAPS prevents composers from making overly broad statements about interaction. Specifying interaction in the most semantically precise way will make programs more resilient to future changes [77]; which aids reusability.

7.4.3 Extensions to the System of Labels

SAPS introduces many new concepts into SOP. These concepts primarily affect type combination. Composition of elements whose attributes are subjective ownership types requires us to define new combinators. Type combination produces `unk` resolutions. To ensure that subject are composed correctly it is necessary to check that `unks` have resolved completely. Direct resolution through type combination creates some but not all resolutions. Therefore, in an additional final step to the composition process the Subject Composer uses resolution propagation to indirectly resolve all `unks` in all subjects.

We propose to extend to the system of labels to incorporate the following SAPS concepts:

- Composable and uncomposable classes defined or used by the subject.

- Instance variables and operations in composable classes only.

- Unknown contexts declared in the subject.

- Call sets appear only in composable classes but may call operations of composable classes.

- Clauses to collect `unk` resolutions from type combinations.

$*$	S :	**subject**
	$S.cls$:	**composable classes**
	$S.imp$:	**uncomposable classes**
$*$	$S.ops$:	**operations**
$*$	$S.map$:	**mappings**
	$S.own$:	**unknown contexts**
	$S.cls.c$:	**composable class**
	$S.cls.c$:	**resolution mapping** ρ
	$S.imp.c$:	**uncomposable class** $\langle k_1 \ldots k_n \rangle$
$*$	$S.cls.c.v$:	**instvar of type** t
$*$	$S.ops.o$:	**operation with signature** $\langle t_0, t_1 \ldots t_n \rangle$
	$S.ops.o$:	**partial resolution mapping** ρ^∂
	$S.imp.o.c$:	**uncomposable realisation set**
$*$	$S.map.o.c$:	**realisation set returning** u
$*$	$S.map.o.c.r$:	**realisation**
$*$	$S.map.o.c.m.r$:	**call set** $\langle \ldots, S.map.m.r, \ldots \rangle$
	$S.own.k$:	**unknown context**

Figure 7-22: Label clauses.

Figure 7-22 shows the additional clauses created to incorporate these. The * in the leftmost column indicates that the clause is unchanged from Figure 7-5 on page 138. Working from the top, the *cls* group now contains only composable classes. The *imp* group refers to uncomposable classes which are either defined in subject S or imported from an external library into S. The *own* group contains the subject's unks.

The *composable class* clause labels composable classes. The new **resolution mapping** attribute of composable classes collects the direct resolutions pertaining to each class. ρ is a set whose entries are of the form $k \mapsto n$ where k is an unk and n is an exp. Operation labels gain the **partial resolution mapping** attribute. ρ^∂ has the same form as the **resolution mapping** attribute of class labels. ρ^∂ temporarily store resolutions from operation signature combinations which take place independently of classes. An *uncomposable class* clause introduces class c which has sequence $\langle k_1 \ldots k_n \rangle$ of ownership parameters. The sequence does not include the implicit owner parameter. Uncomposable classes are included because they are used as types of composable elements. An *uncomposable realisation set* clause denotes operation o in uncomposable class c. This clause is included because the call set of a composable class may include calls to operations of uncomposable classes. An *unknown context* clause defines an unk appearing in a subject. Unknown context identifiers are also composable elements of subjects that may be grouped with unks from other subjects.

The creation of resolution mappings is closely related to the issue of attribute combination. Therefore, it makes sense to describe combination and resolution mappings together in Section 7.5. Checking resolution mappings for consistency can take place only once all direct resolution are collected. These checks are described in Section 7.6 on page 170.

Composition Construct	Attribute	Combinator	Resolution Mapping Fn
instance variable			R_{inst}
	type	C_t	
operation			R_{ops}
	signature	C_g	
realisation set			R_{rs}
call set			R_{cs}
	call set attrib.	C_g	

Figure 7-23: Composition elements used in the definition of the SAPS **mergeByName** composition rule

7.5 Type Combinators and Resolution Collection

This Section defines type combinators and resolution mapping functions used by composition rules. Figure 7-23 shows that there are two type combinators: C_t for combining a sequence of types and C_g for combining types in signatures. The latter simply calls the former to combine the types at each position. Signature combination is used to combine the attributes of **operation** clauses and to check the attributes of **call set** clauses.

Type combination produces `unk` resolutions for the class where resolution occurs. Resolution mapping functions are shown in the third column in Figure 7-23. Type combinators are applied in the context of some clause. Figure 7-4 on page 137 shows that in the case of **instvar** and **call set** labels, the class in which resolution occurs is known from the compound name; call sets and instance variables are nested inside a class. The same is not true of **operation** labels. Resolutions from compositions of **operation** labels cannot be immediately attributed to the a class. Instead, we postpone resolution mapping creation until **realisation set** labels are combined. Resolution mappings are stored as attributes of the output **class** labels. When operation signatures are combined we store partial resolution mappings as attributes of **operation** labels.

7.5.1 The Type Combinator

The type combinator $C_t(Q)$ determines the output type by composing types in Q. In the process, C_t checks that the types can be combined. What denotes a composable sequence of types? Ownership types have two parts: the data type and the ownership context. In order for types to be composable both parts should correspond. The data types must come either from the same uncomposable class or from composable classes that are specified as corresponding. For example, for composable classes `R` and `S` the following must hold:

$$\begin{array}{rcl} C_t(\langle \mathit{S1.cls.R}\langle 1\rangle, & \Longrightarrow & \mathit{S.cls.T} : \textbf{composed-of}(\langle \mathit{S1.cls.R}, \mathit{S2.cls.S}\rangle, \\ \mathit{S2.cls.S}\langle 1\rangle) = T\langle 1\rangle & & \{\mathit{S1.cls.R}, \mathit{S2.cls.S}\}) \in U \end{array}$$

Ownership contexts can be made up of explicit and unknown context identifiers. When `unk`s are used, we require every combination of contexts to yield a resolution. Hence, for any sequence Q there must be at least one type with an `exp` for each context parameter position.

Before C_t is presented, we must describe the functions used in its definition. **dt-forward**(Q) determines the name of the data component of the output type from a sequence of input types $Q = \langle t_1 \dots t_k\rangle$.

$$\begin{array}{lcll}
\textbf{dt-forward}(\langle t_1 \ldots t_k\rangle) & \overset{\text{def}}{=} & \textbf{dt}(t_1) & \textbf{if } \forall i \in [1,k]\ \textbf{extract}_{\_.?}(t_i) = \mathit{imp} \wedge \\
& & & \textbf{dt}(t_1) = \textbf{dt}(t_2) = \ldots = \textbf{dt}(t_k) \\
& & \textbf{forwards-to}(\textbf{dt}(t_1)) & \textbf{if } \forall i \in [1,k]\ \textbf{extract}_{\_.?}(t_i) = \mathit{cls} \wedge \\
& & & \textbf{forwards-to}(\textbf{dt}(t_1)) = \ldots = \textbf{forwards-to}(\textbf{dt}(t_k))
\end{array}$$

Let M_c be a two-dimensional matrix of contexts created from Q. The context identifiers of each type are placed into a row such that $M_c[i,j]$ refers to the jth context in the ith type in Q. We define two indices $\textbf{exp}_{index}(M_c, j)$ and $\textbf{unk}_{index}(M_c, j)$ to column j of matrix M_c as follows:

$$\begin{array}{rcl}
\textbf{exp}_{index}(M_c, j) & \overset{\text{def}}{=} & \{i \mid M_c[i,j] \in \mathcal{N} \cup \{\mathsf{world}\}\} \\
\textbf{unk}_{index}(M_c, j) & \overset{\text{def}}{=} & \{i \mid M_c[i,j] \notin \mathcal{N} \cup \{\mathsf{world}\}\}
\end{array}$$

Note that i is in range $[1, |Q|]$.

The **exp-exists** test holds if there is at least one exp for a corresponding set of contexts in column j of M_c. The **exp-equiv** test holds if **exp-exists** and all corresponding exps in column j are equivalent (or equal to world). **exp-value** returns the output context value for column j:

$$\begin{array}{rcl}
\textbf{exp-exists}(Q, j) & \overset{\text{def}}{=} & \textbf{exp}_{index}(M_c, j) \neq \emptyset \\
\textbf{exp-equiv}(Q, j) & \overset{\text{def}}{=} & \textbf{exp-exists}(Q, j) \wedge \forall i, i' \in \textbf{exp}_{index}(M_c, j), M_c[i,j] = M_c[i',j] \\
\textbf{exp-value}(Q, j) & \overset{\text{def}}{=} & M_c[i,j]\ \textbf{if exp-equiv}(Q, j) \wedge i \in \textbf{exp}_{index}(M_c, j) \\
& & \mathsf{null}\ \textbf{otherwise}
\end{array}$$

Now we can present the type combinator. C_t takes a sequence of types to combine Q and returns the output type.

Definition: (Type Combinator)

$$\begin{array}{rcl}
C_t(Q) & \overset{\text{def}}{=} & \textbf{dt-forward}(Q)\langle \textbf{exp-value}(Q,1), \ldots, \textbf{exp-value}(Q,m)\rangle \\
& & \text{where } Q = \langle t_1\langle c_{1,1} \ldots c_{1,m}\rangle, t_2\langle c_{2,1} \ldots c_{2,m}\rangle \ldots t_k\langle c_{k,1} \ldots c_{k,m}\rangle\rangle
\end{array}$$

The Main Resolution Mapping Function

The resolution mappings of unks are stored in the clause universe as attributes of class labels. Alongside combinators, the Resolution Mapping Function $R(Q)$ is called to create the resolution set for a sequence of corresponding types Q. The definition of R also uses the matrix representation of types.

The exps and the unks appearing in the same column of M_c generate resolutions. For some column j of M_c, **res-map** creates a set of resolutions:

$$\textbf{res-map}(Q, j) \overset{\text{def}}{=} \{M_c[i,j] \mapsto \textbf{exp-value}(Q, j) \mid i \in \textbf{unk}_{index}(M_c, j)\}$$

A union of resolutions produced by all columns gives the complete resolution mapping for a sequence of corresponding types in Q:

Definition: (Resolution Mapping Function)

$$R(Q) \overset{\text{def}}{=} \textstyle\bigcup_{j \in [1,m]} \mathbf{res\text{-}map}(Q, j)$$
$$\text{where } Q = \langle t_1\langle c_{1,1} \ldots c_{1,m}\rangle, t_2\langle c_{2,1} \ldots c_{2,m}\rangle \ldots t_k\langle c_{k,1} \ldots c_{k,m}\rangle\rangle$$

Resolutions on Instance Variables

When instance variables are composed, the resolution mapping attribute for the class containing the instance variable n is updated with resolutions from combinations of types in Q:

$$\begin{aligned} R_{inst}(n, Q) \quad \overset{\text{def}}{=} \quad & \mathbf{replace}(c : \mathbf{resolution\ mapping}\ \rho,\ c : \mathbf{resolution\ mapping}\ \rho \cup R(Q)) \\ & \text{where } c = \mathbf{extract}_{?.cls}(n).cls.\mathbf{extract}_{\_.cls.?}(n) \end{aligned}$$

7.5.2 Type Sequence Combinator

The type sequence combinator is best understood in terms of its application to operation signatures. Signatures are the attributes of operation clauses and this combinator is most commonly used to combine the signatures of corresponding operations. In order to be composable, operations must define the same number of parameters. The return values and parameters in corresponding positions must have equivalent types as defined by C_t.

The type sequence combinator $C_g(Q)$ takes a sequence Q of type subsequences to be combined, and produces the output subsequence. C_g is defined in terms of the type combinator C_t which is called once for each set of corresponding types.

Definition: (Type Sequence Combinator)

$$C_g(Q) \overset{\text{def}}{=} \langle C_t(\langle t_{0,0}, t_{1,0} \ldots t_{k,0}\rangle), \ldots C_t(\langle t_{0,m}, t_{1,m} \ldots t_{k,m}\rangle)\rangle$$
$$\text{where } Q = \langle\langle t_{0,0}, t_{0,1} \ldots t_{0,m}\rangle, \langle t_{1,0}, t_{1,1} \ldots t_{1,m}\rangle, \ldots, \langle t_{k,0}, t_{k,1} \ldots t_{k,m}\rangle\rangle$$

Resolutions in Operation Signatures

Signature combination produces `unk` resolutions which should be associated with output classes. But operation signature combination is defined separately from classes. The missing information becomes available only when realisation sets are combined. In the meantime, we associate resolutions from signature combination with the *partial resolution mapping* attribute of the output operation label n.

$$\begin{aligned} R_{ops}(n, Q) \quad \overset{\text{def}}{=} \quad & \mathbf{replace}(n : \mathbf{partial\ resolution\ mapping}\ \rho^{\partial}, \\ & n : \mathbf{partial\ resolution\ mapping} \bigcup_{i \in [1,m]} R(\langle t_{0,i}, t_{1,i} \ldots t_{k,i}\rangle)) \\ & \text{where } Q = \langle\langle t_{0,0}, t_{0,1} \ldots t_{0,m}\rangle, \langle t_{1,0}, t_{1,1} \ldots t_{1,m}\rangle, \ldots, \langle t_{k,0}, t_{k,1} \ldots t_{k,m}\rangle\rangle \end{aligned}$$

Realisation set combination provides an opportunity to fill in the missing information for partial resolutions created by signature combination. Partial resolution mappings are stored as attributes of output operation labels such as the following:

$$n.ops.foo : \textbf{partial resolution mapping } \{k \mapsto v\}$$

The name of the class is taken from the name of the output realisation set label r, and the resolution mapping attribute of this class is updated. Notation $m\$prm$ denotes the partial resolution mapping attribute of label m:

$$\begin{aligned} R_{rs}(r) \overset{\text{def}}{=} \;& \textbf{replace}(c : \textbf{resolution mapping } \rho,\ c : \textbf{resolution mapping } \rho \cup \rho^{\partial}) \\ & \text{where } c = \textbf{extract}_{?.map}(r).cls.\textbf{extract}_{\_.map.\_.?}(r) \\ & \text{and } \rho^{\partial} = (\textbf{extract}_{?.map}(r).ops.\textbf{extract}_{\_.map.?}(r))\$prm \end{aligned}$$

7.5.3 Checking Call Sets

A call set label has as its attribute a sequence of realisation sets. Any method calls to the realisation set denoted by the label generate calls to the elements in the attribute. This property is used with bracket relationships on call sites. During composition it is necessary to type check that the signatures of operations described by the call sets match because the same arguments are bound to the parameters of all operations. However, there is an exception. The wrapper operations used in bracket relationships also accept either no parameters or special meta-parameters describing the bracketed operation, e.g. the bracketed operation name. These do not concern us, for wrapper operations with no parameters or meta-parameters do not combine types. Methods used as wrappers always have `void` return type, so only parameter types are checked.

Call set checking applies C_g to sequences of signature types not including the return type. For attribute M of a call set label CS this sequence is given by:

$$\begin{aligned} Br \;=\; & \langle [\textbf{match}(\textbf{extract}_{?.map}(CS).ops.\textbf{extract}_{\_.map.?}(M[i]))]\$params \mid i \in [1, |M|] \rangle \\ & \text{where } params \text{ denotes the parameter types of an operation label} \end{aligned}$$

Resolutions from Call Sets

Allied with the above checks is resolution mapping collection from call sets. The `unks` in the interface of wrapper methods resolve the `exps` in the types of the wrapper operation.

Function R_{cs} updates the resolution mapping of the class containing the call set label. It has two parameters where the first parameter n is the compound name of the call set label, and the second parameter Q is a sequence of signature types not including the return type given by Br above:

$$\begin{aligned} R_{cs}(n, Q) \overset{\text{def}}{=} \;& \textbf{replace}(c : \textbf{resolution mapping } \rho, \\ & c : \textbf{resolution mapping } \rho \cup \bigcup_{t \in [1,s]} R(\langle p_{0,t}, p_{1,t} \dots p_{k,t} \rangle) \\ & \text{where } c = \textbf{extract}_{?.map}(n).cls.\textbf{extract}_{\_.map.\_.?}(n), \text{ and} \\ & Q = \langle \langle p_{0,0}, p_{0,1} \dots p_{0,s} \rangle, \langle p_{1,0}, p_{1,1} \dots p_{1,s} \rangle \dots \langle p_{k,0}, p_{k,1} \dots p_{k,s} \rangle \rangle \end{aligned}$$

```
subject FireController {
  unk sc_owner, arr_owner, pr_owner;
  ucirc arr_owner <= pr_owner;
  abstract class Stage {
    SafetyCurtain<sc_owner> sc;
    Vector<0,pr_owner> props;
    abstract void arrangeProps(Prop<arr_owner,pr_owner>[] pr);
    void makeSafe() {
      arrangeProps(null);
      sc.lower();
    }
  }
  ...
}
subject Performance {
  class Stage {
    SafetyCurtain<1> sc;
    Vector<0,2> props;
    void arrangeProps(Prop<1,2>[] pr) { .. }
  }
}

compose Performance, FireController into SafePerformance;
mergeByName;
```

Figure 7-24: Composition of `Performance` and `FireController` subjects

7.5.4 Example

Type combination and resolution mapping collection is demonstrated in terms of composition of subjects `Performance` and `FireController` shown in Figure 7-24. The output subject, `SafePerformance`, is created using **mergeByName** semantics.

The type combinator is applied to determine the types of corresponding instance variables `sc` and `props`. It is activated after grouping activity completes, creating the following clauses in U: C_t is applied to integrate the types of corresponding instance variables:

$$\begin{array}{ll}
\mathit{SafePerformance.cls.Stage.sc}: & \textbf{instvar of type } C_t(\\
 & \langle \mathit{Performance.cls.SafetyCurtain}\langle 1\rangle, \\
 & \mathit{FireController.cls.SafetyCurtain}\langle \mathit{sc\_owner}\rangle\rangle) \\
\mathit{SafePerformance.cls.Stage.props}: & \textbf{instvar of type } C_t(\\
 & \langle \mathit{Performance.imp.Vector}\langle 0,2\rangle, \\
 & \mathit{FireController.imp.Vector}\langle 0,\mathit{pr\_owner}\rangle\rangle)
\end{array}$$

Next R_{inst} is invoked as follows:

$$\begin{array}{ll}
R_{inst}(\mathit{SafePerformance.cls.Stage.sc}, & \langle \mathit{Performance.cls.SafetyCurtain}\langle 1\rangle, \\
 & \mathit{FireController.cls.SafetyCurtain}\langle \mathit{sc\_owner}\rangle\rangle) \\
R_{inst}(\mathit{SafePerformance.cls.Stage.props}, & \langle \mathit{Performance.imp.Vector}\langle 0,2\rangle, \\
 & \mathit{FireController.imp.Vector}\langle 0,\mathit{pr\_owner}\rangle\rangle)
\end{array}$$

R_{inst} creates resolution which are added to clause universe as the attribute of the pertinent class label. The clause universe will contain the following clauses:

$$\begin{array}{ll} \mathit{SafePerformance.cls.Stage.sc} : & \textbf{instvar of type } \mathit{SafePerformance.cls.SafetyCurtain}\langle 1\rangle \\ \mathit{SafePerformance.cls.Stage.props} : & \textbf{instvar of type } \mathit{SafePerformance.imp.Vector}\langle 0,2\rangle \\ \mathit{SafePerformance.cls.Stage} : & \textbf{resolution mapping } \{\mathit{sc\_owner} \mapsto 1, \mathit{pr\_owner} \mapsto 2\} \end{array}$$

To resolve unk_{arr_owner}, operations `arrangeProps` must be composed using the type sequence combinator C_g. C_g invokes C_t once in a benign way to combine the corresponding `void` types of return values. The second invocation combines the parameter types. At this point R_{ops} is invoked in order obtain the resolutions from this signature combination:

$$\begin{array}{ll} R_{ops}(\mathit{SafePerformance.ops.arrangeProps}, & \langle\langle \mathit{void}(), \mathit{Performance.cls.Prop}\langle 1,2\rangle\rangle, \\ & \langle \mathit{void}(), \mathit{FireController.cls.Prop}\langle \mathit{arr\_owner}, \mathit{pr\_owner}\rangle\rangle\rangle) \end{array}$$

After these activities the clause universe contains the following clauses:

$$\begin{array}{ll} \mathit{SafePerformance.ops.arrangeProps} : & \textbf{operation with signature } \langle \mathit{void}\langle\rangle, \\ & \mathit{SafePerformance.cls.Prop}\langle 1,2\rangle\rangle \\ \mathit{SafePerformance.ops.arrangeProps} : & \textbf{partial resolution mapping } \{\mathit{arr\_owner} \mapsto 1, \mathit{pr\_owner} \mapsto 2\} \end{array}$$

mergeByName creates a correspondence clause for the realisation set representing `Stage.arrangeProps(..)`. Combination of realisation sets provides an opportunity to complete the partial resolution. Resolutions in operation `arrangeProps` occur inside class `Stage`. The resolution mapping of class `Stage` is updated with the partial resolutions from operation `arrangeProps`, giving:

$$\begin{array}{ll} \mathit{SafePerformance.cls.Stage} : & \textbf{resolution mapping} \\ & \{\mathit{sc\_owner} \mapsto 1, \mathit{arr\_owner} \mapsto 1, \mathit{pr\_owner} \mapsto 2\} \end{array}$$

7.6 Resolution Validation

Resolution validation checks that `unks` resolve to `exps` in all classes where they appear and that resolutions satisfy resolution constraints. For each `unk` in the output subject we construct a graph. Its vertices are classes and edges are established by the definitions and behaviour in the output classes.

We will present a resolution propagation algorithm which attempts to determine an `exp` value for each vertex of each graph. Resolution propagation starts when the graphs are seeded by direct resolutions from type combinations. In the present work, failure to determine the `exp` for all vertices indicates an invalid subject composition. The value of partial resolution is discussed in Future Work on page 215. Before presenting the algorithm we explain the preparation stages which include `unk`, resolution constraint and inter-class relationship collection.

7.6.1 Preparation

Resolution validation requires access to the following:

- The set of `unks` that appear in the types of composition components. The `unks` pertaining to each class are a union of `unks` in the composition subelement of the class.
- The set of resolution constraints that apply to the `unk` set. These are used to check that `unks` resolve correctly in each class.

$S.cls.c.v$:	**instvar of type** t
$S.ops.o$:	**operation with signature** $\langle t_0, t_1 \ldots t_n \rangle$
$S.map.o.c.r$:	**realisation types** $\{t_1 \ldots t_m\}$
$S.cls.c$:	**unks to resolve** $\{u_1 \ldots u_p\}$
$S.cls.c.v$:	**resolution constraints** $\langle$e.g. `1 <= k, m <= n, p <= 2`$\rangle$
$S.ops.o$:	**resolution constraints** $\langle \ldots \rangle$
$S.map.o.c.r$:	**resolution constraints** $\langle \ldots \rangle$
$S.cls.c$:	**resolution constraints** $\langle \ldots \rangle$
$S.map.o.c.r$:	**association set** $\{\langle d, n, k \rangle\}^*$
$S.cls.A_1.v$:	**with classes** $\{A_2 \ldots A_n\}$

Figure 7-25: Labels used for resolution validation.

- The resolutions mappings from type combinations for seeding the resolution propagation algorithm.
- Relationships between classes by which a resolved value can be propagated from class to class.

All this information is collected during subject type checking. The system of labels conveys the resolution data to the resolution propagation algorithm.

unk Collection

The set of all unks in a class is determined from the types of composition components defined or used in that class. The top three labels in Figure 7-25 show that this information is obtained from:

- The declared types in operation signatures, instance and local variables.
- The types of expressions in realisations.

An unk resolves to one exp per class, so the next task is to collect all unks pertaining to each output class. The "**unks to resolve**" attribute of class labels holds this set. Suppose that an output class $S.C$ is created by composing $S_1.C_1 \ldots S_n.C_n$, then the attribute value is given by **unksToResolve**:

$$\begin{aligned}\textbf{unksToResolve}(S.cls.C) \overset{\text{def}}{=} \; & \{\textbf{get-unks}(v\$\mathit{type}) \mid v \in \textbf{match}(S_i.cls.C_i.\_) \wedge i \in [1,n]\} \; \cup \\ & \{\textbf{get-unks}(o\$\mathit{sig}) \mid o = \textbf{match}(S_i.ops.x) \wedge \\ & x \in \textbf{match}(S_i.map.\_.C_i) \wedge i \in [1,n]\} \; \cup \\ & \{\textbf{get-unks}(r\$\mathit{unks}) \mid r \in \textbf{match}(S.map.\_.C.\_)\}\end{aligned}$$

where, for some type t or set of types T, **get-unks** is defined as:

$$\begin{aligned}\textbf{get-unks}(T) & \overset{\text{def}}{=} \bigcup_{t \in T} \textbf{get-unks}(t) \\ \textbf{get-unks}(t) & \overset{\text{def}}{=} \{c \mid c \text{ is an } \mathsf{unk} \text{ in } t\}\end{aligned}$$

Resolution Constraint Collection

Composition changes the make-up of a class, introducing new members and overriding operation implementations. Consequently, the resolution constraints of an output class must be gathered from its subcomponents. In the system of labels only instance variables, operation signatures and realisations are associated with types. Figure 7-25 shows that in the system of labels, this information is available as attributes of **instvar**, **operation**, and **realisation** clauses. The constraints pertaining to each output class $S.C$ composed from $S_1.C_1 \dots S_n.C_n$ are collected together using **collectUcircs** and set as an attribute of $S.C$:

$$\begin{aligned}\mathbf{collectUcircs}(S.cls.C) \overset{\text{def}}{=}\ & \{v\$rc \mid v \in \mathbf{match}(S_i.cls.C_i.\_) \wedge i \in [1,n]\} \ \cup \\ & \{o\$rc \mid o = \mathbf{match}(S_i.ops.x) \wedge \\ & x \in \mathbf{match}(S_i.map.\_.C_i) \wedge i \in [1,n]\} \ \cup \\ & \{r\$rc \mid r \in \mathbf{match}(S.map.\_.C.\_)\}\end{aligned}$$

The penultimate step is to reduce the resolution constraints to a canonical form using the following rewrite rules. n, n' denote exps and u, v, w denote unks.

$$\begin{aligned}\{(u \leq n), (u \leq n')\} \subseteq RC \ & \overset{\text{redex}}{\Longrightarrow} \ \{u \leq \mathbf{min}(n, n')\} \cup [RC - \{(u \leq n), (u \leq n')\}] \\ \{(n \leq u), (n' \leq u)\} \subseteq RC \ & \overset{\text{redex}}{\Longrightarrow} \ \{u \leq \mathbf{max}(n, n')\} \cup [RC - \{(n \leq u), (n' \leq u)\} \\ \{(u \leq v), (v \leq w)\} \subseteq RC \wedge (u \leq w) \notin RC \ & \overset{\text{redex}}{\Longrightarrow} \ \{u \leq w\} \cup RC\end{aligned}$$

The resolution sets of unks may not be empty. No input subject's resolution set was empty before composition, so an empty resolution set at this point indicates an invalid composition. unk cycles lead to singleton resolution sets. Singleton resolution sets are acceptable during composition because they still make valid resolution possible.

Example

To demonstrate resolution constraint collection consider the example in Figure 7-26. The example shows a **mergeByName** composition of subjects S1, S2 and S3 into the output subject S. For output classes A and T, two unks are defined: unk_k and unk_m. Resolution constraints are collected from all instance variables types, operation signatures and method implementations that contribute to classes A and T. **mergeByName** semantics ensure that all input elements contribute to the output, so we can equally well observe the constraints collection from Figure 7-26:

- For class A:

  line 4: $k \leq 1$

  line 6: $k \leq 1$

  line 14: $k \leq m$

  line 16: $1 \leq k, 1 \leq m$

- For class T:

  line 20: $k \leq m$

  line 26: $1 \leq m$

```
subject S1 {
  unk k;
  class A where k <= 1 {
    Vector<k,1> v;
    void foo(S<1> s) {
      v.add(s);
    }
  }
}
subject S2 {
  unk k, m;
  ucirc k <= m;
  class A where 1 <= k, 1 <= m {
    Vector<k,m> v;
    void bar(T<1> t) {
      t.v = v;
    }
  }
  class T where 1 <= k, 1 <= m {
    Vector<k,m> v;
  }
}
subject S3 {
  unk m;
  class T where 1 <= m {
    Vector<1,m> v;
  }
}
compose S1, S2, S3 into S;
mergeByName;
```

Figure 7-26: Resolution validation example

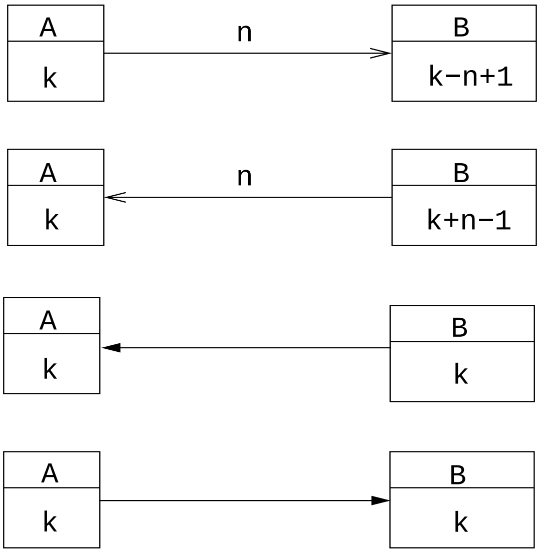

Figure 7-27: unk resolution propagation rules

This composition creates the following labels in the clause universe:

$$\begin{aligned} S.cls.A &: \textbf{unks to resolve } \{k, m\} \\ S.cls.T &: \textbf{unks to resolve } \{k, m\} \\ S.cls.A &: \textbf{resolution constraints } \{(k \leq 1), (k \leq m), (1 \leq k), (1 \leq m)\} \\ S.cls.T &: \textbf{resolution constraints } \{(k \leq m), (1 \leq m)\} \end{aligned}$$

The composition in Figure 7-26 has created resolutions that are represented in U as attributes of class nodes:

$$\begin{aligned} \textbf{Correspondence of lines 4 and 14} \quad S.cls.A &: \textbf{resolution mapping } \{m \mapsto 1\} \\ \textbf{Correspondence of lines 20 and 26} \quad S.cls.T &: \textbf{resolution mapping } \{k \mapsto 1\} \end{aligned}$$

Resolution Propagation Between Classes

Indirect resolution uses association and inheritance for propagation. The principles of context identifier arithmetic presented in Section 6.2.2 on page 107 create the association links. Propagations can pass both up and down the inheritance hierarchy using the types of class members to create links.

We can represent both kinds of propagation pictorially in class graphs. Figure 7-27 shows resolution propagation for unk_k. In each diagram, unk_k is already resolved in class A, and the aim is to resolve it in B. There are two kinds of edges between classes:

- **Association** is represented by open ended edges. Associations denote resolution propagation due to behaviour, such as due to method calls, field access and update expressions. The identifier on the edge denotes the owner context of the receiver expression.

- **Inheritance** is represented by solid triangular ended edges. Here, inheritance denotes resolution propagation due to the declared inheritance relationships.

From the top down in Figure 7-27, an association edge from `A` to `B` indicates that `A` contains an expression that passes an object whose type contains `unk`$_k$ to an n-owned instance of class `B`. The expression in `B` denotes the value of `unk`$_k$, calculated from the value of `unk`$_k$ in `A` and the value on the edge. Similarly, for the second picture from the top, the value of `unk`$_k$ is known in `A` and not known in `B`. The arrow direction denotes that the expression connecting these classes is defined in `B`. n can be an `unk` or an `exp`. When n is an `unk`, it must be resolved at the class from where the arrow emanates before proceeding to resolve `unk`$_k$ in `B`.

For inheritance, the directed end points to the superclass. If `A` is a subclass of `B`, `unk`$_k$ should resolve to the same value in `B` only if `B` also has types that utilise `unk`$_k$. If `A` is `B`'s superclass and `B` has types that utilise `unk`$_k$, then `unk`$_k$ should resolve to the same value in `B` as in its superclass.

7.6.2 Clausal Representation of Association and Inheritance

In the clause universe the association relationships in subjects are represented as attributes of realisations. These labels are constructed during subject typechecking. Figure 7-25 shows that the **association set** attribute of realisations is a set of tuples where d is the datatype of the receiver expression, n is the owner context (explicit or unknown) in the type of the receiver expression and k is an `unk` in the type of the actual parameter or the field update expression. For example, consider the following code and the label it generates:

```
1    class C {
2      D<0> d;
3      E<k> e;
4      F<p, q> f;
5      void foo() {
6        d.bar(e);
7        e.f = f;
8      }
9    }
```

Suppose the body of method `foo()` is represented by label $S.map.foo.C.r$. The expressions in the realisations produce the following association tuples:

Line 6: $\langle D, 0, k\rangle$

Line 7: $\langle E, k, p\rangle$; $\langle E, k, q\rangle$

Note that line 7 produces two tuples: one for each `unk` in the type of the expression on the right hand side of the assignment. The complete label is:

$$S.map.foo.C.r : \textbf{ association set } \{\langle D, 0, k\rangle, \langle E, k, p\rangle, \langle E, k, q\rangle\}$$

Inheritance between classes is not represented explicitly in the system of labels but is inferred from realisation set, realisation and instance variable labels:

- For realisation set-based propagation, suppose that `unk`$_k$ resolves in class C of subject S and there exist the following labels in the clause universe:

$$\begin{array}{ll} S.map.foo.C : & \textbf{realisation set returning } \ldots \\ S.ops.foo : & \textbf{operation with signature } \langle \ldots, t\langle \ldots, k, \ldots\rangle, \ldots\rangle \end{array}$$

Then all other classes in S that also define operation `foo(..)` have the same resolution for unk_k. This notion is formalised by **RSBP** which takes subject S and operation *foo*, and returns the labels of classes that share this operation. The set of classes affected by this resolution is given by:

$$\mathbf{RSBP}(S, \mathit{foo}) \stackrel{\text{def}}{=} \{c \mid c \in \mathbf{match}(S.\mathit{map}.\mathit{foo}.\_)\}$$

- For realisation-based propagation, suppose that unk_k resolves in class C, realisation r has unk_k in its **realisation types** attribute and the clause universe has the following labels:

  $$\begin{array}{rl} S.\mathit{map}.\mathit{foo}.C.r: & \textbf{realisation} \\ S.\mathit{map}.\mathit{bar}.D.r: & \textbf{realisation} \\ S.\mathit{map}.\mathit{foo}.E.r: & \textbf{realisation} \end{array}$$

  That is, three realisation sets share the same realisation r. Classes D and E must also have the same resolution. Hence, when unk_k resolves in class C, it resolves in all classes in this set. In the general case, for realisation r', the set of classes in subjects S that have r' in one or more realisation sets is given by $\mathbf{RBP}(S, r')$:

  $$\mathbf{RBP}(S, r') \stackrel{\text{def}}{=} \{\mathbf{match}(S.\mathit{cls}.x) \mid \; x \in \{\mathbf{extract}_{\_.\mathit{map}.\_.?}(\mathit{rs}) \mid \mathit{rs} \in \mathbf{match}(S.\mathit{map}.\_.\_.r')\}$$

- Finally, in instance variable-based propagation instance variables propagate resolutions up and down the class hierarchy. Suppose unk_k resolves in class C and there exist the following labels in the clause universe:

  $$S.\mathit{cls}.C.v: \;\textbf{instvar of type}\; t\langle \ldots, k, \ldots \rangle$$

  Then, all classes that define the same instance variable will also resolve unk_k. The following *does not* specify all affected classes because unrelated classes may define an instance variable with the same name:

  $$\{c \mid c \in \mathbf{match}(S.\mathit{cls}.\_.v)\}$$

  Instead, all classes that define or inherit the same instance variable are given as values of the **with classes** attribute of an instance variable label (also shown in Figure 7-25 on page 171:

  $$S.\mathit{cls}.A_1.v: \;\textbf{with classes}\; \{A_2 \ldots A_n\}$$

  Upon seeing this label we can conclude that there exist $n - 1$ other labels:

  $$S.\mathit{cls}.A_i.v: \;\textbf{with classes}\; \{A_1 \ldots A_n\} \setminus \{A_i\} \quad i \in [2, n]$$

  Supposing unk_k resolves in class C of subject S, for each instance variable v whose contexts include unk_k, the set of classes affected by this resolution is given by **IVBP**:

  $$\mathbf{IVBP}(S, C, v) \stackrel{\text{def}}{=} \{\mathbf{match}(S.\mathit{cls}.x) \mid x \in S.\mathit{cls}.C.v\$\mathit{wc}\}$$

  where notation $p\$\mathit{wc}$ denotes the values associated with the **with classes** attribute of label p.

7.6.3 Graph Representation for Resolution

Resolution propagation is most easily understood in terms of a class graph representation. For each unk_k in the output subject S, we construct graph G_k by obtaining values from the clause universe. Each $G_k = \langle V, A, H\rangle$ is made up of three elements: V is the set of vertices; A is the set of association edges connecting the the elements of V such that $v \xrightarrow{n} v' \in G_k$ if $\langle v, n, v'\rangle \in A$; H is the set of (bi-directed) inheritance edges connecting the elements of V such that $v \leftrightarrow v' \in G_k$ if $\langle v, v'\rangle \vee \langle v', v\rangle \in H$. The edges are bi-directional to indicate that the same rule is used to propagate resolutions both ways. The set of all graphs is $G = \{G_{k_1} \dots G_{k_m}\}$. For each $G_k \in G$ the sets V, A, H are constructed as follows:

- The set of vertices in G_k is given by:

  $$\{c \mid c \in \mathbf{match}(S.cls.\_) \wedge k \in c\$utr\}$$

  where $c\$utr$ is the value associated with the **unks to resolve** attribute of class c.

- The association edges are drawn based on **association set** attributes of realisation labels. Each such attribute has class C as a component of its compound name. The attribute value is a set of tuples of the form $\langle C', n, u\rangle$. G_k has an edge labelled n from C to C' if and only if $k = u$.

- The inheritance edges are drawn based on the propagation rules defined in the preceding Section. In G_k, we draw an edge between C and its superclass C' if and only if C inherits from C' a method or an instance variable, either directly or transitively, that has unk_k in its type or signature, or if any methods of C (realisation sets) share code (a realisation) with class C' that contains an expression or sub-expression whose type features unk_k.

With each vertex v in G_k we associate its resolved value. In addition, the following functions are defined:

- $\mathbf{lookup}_k(v)$ is the resolved value of unk_k at vertex v in G_k (class v), or **error** if unk_k is not resolved.

- $\mathbf{update}_k(v, n)$ sets the resolution for unk_k in class v to n, i.e. the value at vertex v in G_k is set to n.

Class labels' **resolution constraints** attributes are not added to graphs. For each class C they are placed verbatim into resolution constraints environments $\mathcal{RC}^C$. Let notation $\mathcal{RC}^C[k/n]$ denote the environment created by substituting n for unk_k in $\mathcal{RC}^C$. When unk_k resolves to n in class C, we perform the substitution in the environment. If the constraints are satisfied, $\mathcal{RC}^C$ is reduced by eliminating tautologies, i.e. inter-exp expressions. Otherwise the resolution is invalid and composition aborts.

Propagation starts by applying the resolution mapping on to the graph set. For each vertex of each graph we apply the resolution if one exists and also reduce the appropriate resolution constraints environment.

$$
\begin{array}{l}
\forall G_k = \langle \{v_1 \ldots v_m\}, \_, \_ \rangle \in G \\
\quad \forall v_i, i \in [1, m] \\
\quad\quad \textbf{let } F_{rm} = [\textbf{match}(S.cls.v_i)]\$rm \textbf{ in} \\
\quad\quad\quad \textbf{update}_k(v_i, F_{rm}(k)) \wedge \mathcal{RC}^{v_i}[k/F_{rm}(k)] \textbf{ if } k \in dom(F_{rm})
\end{array}
$$

where $c\$rm$ is the value of the **resolution mapping** attribute of class c

At this stage we can test for termination and, if not finished, apply the resolution propagation algorithm (presented in Section 7.6.4 on page 179). The composition is valid when all vertices in all graphs in G have a value within the specified constraints. In order to exemplify the resolution theory presented thus far and to motivate the propagation algorithm we present an example.

Example

This example turns attention towards the way resolutions are propagated in the program in Figure 7-26 on page 173. From the clause universe we observe that the output subject has two unks: unk$_k$ and unk$_m$. Two graphs are constructed, $G = \{G_k, G_m\}$:

$$
\begin{array}{rcl}
G_k & = & \langle \{A, T\}, \{\langle A, 1, T \rangle\}, \emptyset \rangle \\
G_m & = & \langle \{A, T\}, \{\langle A, 1, T \rangle\}, \emptyset \rangle
\end{array}
$$

The graphs are identical because a single expression (line 16 in Figure 7-26) propagates the resolutions for unk$_k$ and unk$_m$. The ucircs are given by:

$$
\begin{array}{rcl}
\mathcal{RC}^A & = & \{(k \leq 1), (k \leq m), (1 \leq k), (1 \leq m)\} \\
\mathcal{RC}^T & = & \{(k \leq m), (1 \leq k), (1 \leq m)\}
\end{array}
$$

The resolution mapping is collected from the clause universe and applied to the graphs:

- $m \mapsto 1$ in A:

    1. $\textbf{update}_m(A, 1)$ sets the value at vertex A in G_m to 1.
    2. Substituting 1 for m in $\mathcal{RC}^A$ leads to $\mathcal{RC}^A = \{(k \leq 1), (1 \leq k)\}$.

- $k \mapsto 1$ in T:

    1. $\textbf{update}_k(T, 1)$ sets the value at vertex T in G_k to 1.
    2. Substituting 1 for k in $\mathcal{RC}^T$ leads to $\mathcal{RC}^T = \{(1 \leq m)\}$.

At this point, the value of unk$_k$ is known in T but not in A. Likewise, the value of unk$_m$ is known in A but not in T. Propagation of resolutions occurs as follows:

1. By $\textbf{lookup}_k(T) = 1$ and association edge $\langle A, 1, T \rangle$ in G_k, we conclude that unk$_k$ resolves to 1 in vertex A. The association edge was followed in the direction opposite to its arrow.
2. Substituting 1 for k in $\mathcal{RC}^A$ leads to $\mathcal{RC}^A = \emptyset$.

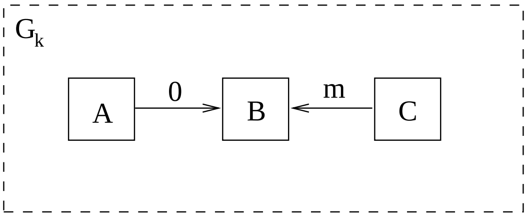

Figure 7-28: Resolution propagation example

3. By $\mathbf{lookup}_m(A) = 1$ and association edge $\langle A, 1, T\rangle$ in G_m, we conclude that unk_m resolves to 1 in vertex T. This time, the association edge was followed in the same direction as the arrow.
4. Substituting 1 for m in $\mathcal{RC}^T$ leads to $\mathcal{RC}^T = \emptyset$.

Now all unks have resolved correctly in all nodes in G. The composition in Figure 7-26 is valid.

7.6.4 Propagation Algorithm

The algorithm for resolution propagation, based on the graph representation described in preceding Section and used in the above example, consists of two parts. *Per-graph propagation* resolves one unk at a time, using resolutions on other unks where possible. *Top-level propagation* applies per-graph propagation until either all vertices in all graphs are resolved or there remain unresolved vertices with no further resolutions possible. This indicates an invalid composition.

The terminating condition is resolution of all vertices in all graphs. It is defined as:

$$\begin{aligned}\mathbf{Term}(G) \overset{\text{def}}{=} \quad & \forall G_{k_i} = \langle \{v_1 \dots v_m\}, \_, \_\}\rangle \in G \\ & \bigwedge_{j\in[1,m]} \mathbf{lookup}_{k_i}(v_j) \in \mathcal{N}\end{aligned}$$

Resolution may also abort prematurely if ucircs are not satisfied, or if a collection of direct resolutions and propagations resolve an unk to different values for one class.

Per-graph propagation differentiates between two kinds of resolution due to association. Consider the graph G_k given in Figure 7-28. Suppose unk_k resolves to exp_0 directly in A. We can immediately resolve unk_k to exp_1 in B. However, resolving unk_k in C is predicated on the resolution of unk_m in C. Suppose unk_m resolves to exp_2 in C. Now we can resolve unk_k, giving exp_2 in C. These two forms of resolution propagation give rise to the following definitions:

- A **simple association** is one where the context on the edge is an exp. **Simple propagation** occurs immediately following the resolution on one end of the edge.
- An **unk-predicated association** is one where the context on the edge is an unk. An **unk-predicated propagation** occurs only after the value at the edge is resolved.

All inheritance based propagations occur immediately because there is no condition on resolution.

Simple and unk-predicated propagations depend on two functions which were described graphically in the top two diagrams of Figure 7-27 on page 174. The context at the vertex is given by m and on the edge by n. Function Δ_1 describes propagation along the arrow and Δ_2 describes propagation in the opposite direction. These were presented in Section 6.2.2 on page 107.

Per-Graph Propagation Algorithm

The **Per-Graph Propagation Algorithm** (**PGPA**), shown in Figure 7-29, is applied on to each graph in G. It proceeds by doing simple, unk-predicated, and inheritance-based propagations. When no more propagation can be done, either because all vertices are resolved or an unk-predicated propagation requires the unk to be resolved, per-graph propagation stalls, and we move on to the next graph. **PGPA** returns the number of successful propagations or aborts. Abortions indicate invalid compositions and can occur due to the following reasons:

- The resolution does not satisfy the resolution constraints.
- At each resolved vertex, **PGPA** rechecks the values at the other end of each edge. There are often multiple propagation paths and all should produce the same resolutions. We abort if the value calculated for a node is different from a value previously calculated via a different path.

PGPA is called recursively if the unk resolved at least at one vertex in the outer call.

Top-Level Propagation Algorithm

The **Top-Level Propagation Algorithm** (**TLPA**), shown in Figure 7-30, proceeds by doing per-graph propagations. These return a count of succesful new resolutions. If, after visiting all graphs, the termination condition is satisfied, the composition is successful and we halt. Otherwise, if any per-graph propagation has a non-zero count, the top-level propagation is restarted. When the count is zero from all per-graph propagations and the termination condition is not satisfied, we conclude that composition failed to resolve all unks subject-wide and the composition is invalid.

To create the output subject, all unks in all classes are replaced by their resolved values in G.

7.7 Conclusion

This Chapter has presented extensions to Subject-Oriented Programming necessary for composing subjects annotated with Subjective Ownership Types. The extensions integrate seamlessly with the subject composition model. Composition of elements is described in terms of the system of labels which represent each subject's composable elements. Composition is based on the concept of correspondence: corresponding labels from different subjects are unified into a single result label. The code for the output subject is created by linking based on the result label.

In order for corresponding elements to be combined, they must define equivalent types. Type equivalence is based on both datatype and context equivalence. The type must derive either from the same uncomposable class or from corresponding composable classes. Context equivalence allows for explicit-explicit and explicit-unknown context combinations. Explicit and unknown context combinations produce resolution mappings which describe the value to which an unk resolves in a particular class.

We require all unks used in all input subjects to be resolved by composition. Thus, subjects featuring unks have the missing information filled in through application to other subjects where contextual information is explicit. Composition rules used in the composition specification are defined in terms of groupers, combinators and resolution mapping functions. Groupers define the elements which should correspond, combinators perform the integration, and resolution mappings are used to eliminate all unks in the output subject. Composition alone is often insufficient to

Definition: (Per-Graph Propagation Algorithm) The following conventions are used in the definitions:

$G = \{G_{k_1} \ldots G_{k_m}\}$
$G_{k_i} = \langle V, A, H \rangle \quad i \in [1, m]$
$\quad V = \{v_j \quad j \in [1, n]\}$
$\quad A = \{\langle v_p, E, v_q \rangle \quad p, q \in [1, n]\}^*$
$\quad H = \{\langle v_p, v_q \rangle \quad p, q \in [1, n]\}^*$

PGPA uses two global variables: *total* is a count of propagations for each iteration of simple/unk-predicated/inheritance-based propagations; *propagations* is a count of propagations for each call to **PGPA**.

```
var $propagations = 0$, $total = 0$

PGPA$(G, i) =$
    $propagations = 0$
    foreach $v_j \quad j \in [1, n] \wedge \mathbf{lookup}_{k_i}(v_j) \in \mathcal{N}$
        let $r_j = \mathbf{lookup}_{k_i}(v_j)$ in
            foreach $\langle v_j, E, v_q \rangle \in A$ where $E \in \mathcal{N}$
                let $r_q = \Delta_1(r_j, E)$ in target-update$(G, i, v_q, r_q)$
            foreach $\langle v_q, E, v_j \rangle \in A$ where $E \in \mathcal{N}$
                let $r_q = \Delta_2(r_j, E)$ in target-update$(G, i, v_q, r_q)$
            foreach $\langle v_j, E, v_q \rangle \in A$ where $E \notin \mathcal{N}$
                let $E_{val} = \mathbf{lookup}_E(v_j)$ in
                    continue if $E_{val} =$ error
                    let $r_q = \Delta_1(r_j, E_{val})$ in target-update$(G, i, v_q, r_q)$
            foreach $\langle v_q, E, v_j \rangle \in A$ where $E \notin \mathcal{N}$
                let $E_{val} = \mathbf{lookup}_E(v_q)$ in
                    continue if $E_{val} =$ error
                    let $r_q = \Delta_2(r_j, E_{val})$ in target-update$(G, i, v_q, r_q)$
            foreach $\langle v_j, v_q \rangle \in H$
                let $r_q = r_j$ in target-update$(G, i, v_q, r_q)$
            foreach $\langle v_q, v_j \rangle \in H$
                let $r_q = r_j$ in target-update$(G, i, v_q, r_q)$
    $total = total + resolutions$
    PGPA$(G, i)$ if $resolutions > 0$
    $total$
```

Function **target-update** modifies the graph with resolutions, reducing the resolution constraints set, or aborts **PGPA** if the value at target is not as expected.

```
target-update$(G, i, v_q, r_q) =$
    update$_{k_i}(v_q, r_q) \wedge \mathcal{RC}^{v_q}[k_i/r_q]$
        $\wedge propagations = propagations + 1$ if lookup$_{k_i}(v_q) =$ error
    abort if lookup$_{k_i}(v_q) \neq r_q$
```

Figure 7-29: Per-Graph Propagation Algorithm

Definition: (Top-Level Propagation Algorithm) The following convention is used in the definition:

$G = \{G_{k_1} \dots G_{k_m}\}$

```
$\mathbf{TLPA}(G) =$
    let $count = 0$ in
        foreach $i \in [1, m]$    $count = count + \mathbf{PGPA}(G, i)$
        halt if $\mathbf{Term}(G)$
        $\mathbf{TLPA}(G)$ if $count > 0$
        abort
```

Figure 7-30: Top-Level Propagation Algorithm

resolve unks in all classes where they are used. We presented a resolution validation algorithm for propagating resolutions subject-wide based on direct resolutions from correspondences.

Chapter 8

Evaluation

In this Chapter we evaluate the Subjective Alias Protection System in order to show that SAPS has addressed the problems that have motivated it. SAPS was motivated first by reuse and secondly by interaction problems. We will show how our proposal improves on Subject-Oriented Programming in both of those areas.

Subject-Oriented Programming is more than an enhancement to Object-Oriented Programming. It represents a a new way of addressing design challenges. SAPS was designed to work in the context of SOP; so it is important to show the range of ways in which SAPS is useful to the subject-oriented developer. We will demonstrate the utility of SAPS to the subject-oriented programmer through a presentation of design cases where reuse and interaction problems play a part.

Evaluation takes place through a range of examples. The examples have been carefully chosen based on a range of applications of SOP, an application of Alias Protection Systems, and to demonstrate a SAPS strongpoint. Limitation of SAPS are discussed also.

Section 8.1 evaluates the contribution of SAPS with respect to the motivation factors for this thesis. SOP enables decomposition of systems by feature. Decomposition by feature can be applied to the development of applications and large grained black-box components. Section 8.2 shows the way to construct components by combining SOT-annotated feature subjects. With SAPS, for all feature combinations, it can be shown that the component, i.e. the output subject, hides its representation from component clients. Section 8.3 shows the modularisation of a cross-cutting concern using SAPS. This example evaluates the flexibility of SAPS when adapting to the different ways the cross-cutting concern may be implemented. In Section 8.4 we show the modularisation of a security concern with SAPS. Uncomposable classes may be used to hide an algorithm implementation that would otherwise be accessible to another subject through join point interaction. SOP has no concept of composable or uncomposable classes, thus there is no way of specifying the places that subjects should not interact. This example shows that SAPS addresses a concern that could not be addressed in SOP without SAPS. In Section 8.5 we show the way explicit contexts may be used to restrict composition in order to steer clear of anomalous interactions. Finally, Section 8.6 demonstrates the known weaknesses of our approach.

8.1 Interaction Problems and Reuse

Our position on reuse means that SAPS is required to play two different roles. In the first role, SAPS has to be useful to the subject's original developer. As part of a design process a system is decomposed into subjects with the intention of developing subjects modularly. In its second role, SAPS has to be useful to the subject composer, the reuser. SAPS annotates the way subjects use objects. During composition, Subjective Ownership Types help the reuser to understand the subject and to gain insight into the consequences of interaction and detect anomalies.

The theme that ties these two roles of SAPS is modularity: the issue of modular construction of subjects and the reuse of subjects as modules. Before descussing interaction problems and reuse, this Section evaluates how SAPS impacts modular software construction with SOP. But first, we present the Library Management System as a running example.

8.1.1 The Library Management System

The Library Management System (LMS) was first introduced by Clarke in her work on Subject-Oriented Design [24]. The LMS manages the resources within a library, and the activities relating to those resources. The full set of features of this system is beyond our scope, but the subset in which we are interested concerns the management of books and periodicals, their ordering and physical location within the library.

A library's resources are multiple copies of books and, optionally, periodicals. Librarians and borrowers are library users but only librarians interact with the system. There are a number of requirements on the system, including:

- **Add library resource.** The librarian may add to the catalogue new books, in some instances new periodicals, or new copies of existing titles. The librarian supplies information on book related details such as author or title. Location information is generated by the system.

- **Remove library resource.** All copies of a given resource may be removed from the LMS once they have been returned to the library by the borrowers.

- **Order library resource.** Order information may be kept in the LMS. Once the order arrives, the system is updated with new resources.

- **Search for library resource.** All users may search for physical locations of copies of a particular title.

- **Borrow library resource.** The borrowing of resources depends on the library where the LMS will be used. In some libraries only books may be borrowed, while in others periodicals may also be borrowed. The number of books that can be borrowed depends on who is doing the borrowing and the application. For example, in a university application, postgraduates may be allowed to borrow 10 books compared to 6 books for undergraduate students.

- **Return library resource.** When a resource is returned late, a fine is issued to the borrower which he must pay before borrowing any more books. The length of time a resource can be borrowed depends on the library and the type of borrower. For instance, librarians may be allowed to borrow books for longer than members of the public.

The LMS is a multi-user application. While searches can be performed concurrently, exclusive access is required in order to add or remove resources.

To enable the traceability of requirements in code, each requirement above can be considered a feature of the LMS. Subjects can be used either to implement these features directly or each feature can be decomposed further into subconcerns with one subject implementing each subconcern. In the experience of Lai and Murphy [71] two people working independently may identify different features of importance in the same piece of software. As our starting point, a system is already decomposed into subjects based on the features the development manager has identified as important.

8.1.2 Modular Development of Subjects

Software is decomposed into modules because it is believed that tackling one module at a time is easier than tackling the whole problem at once. However, the scattering of object representation across subjects in certain subject-oriented decompositions inhibits the modular development of concerns. One of the factors that inspired decomposition by feature was the productivity improvement which may be gained through concurrent development of features by separate teams (see Section 3.1.3 on page 27). The purpose of this Section is to explain what improvements SAPS has made in this respect.

In her thesis on Subject-Oriented Design, Clarke [24] writes that designers can work on subjects representing different parts of the system with little need for communication. It is true that subject-oriented decomposition allows partially overlapping views of a domain to be specified modularly. Two designers can work on the design of one class simultaneously. However, as in any system where modules interact, communication between design teams is required in order to establish the details of the interaction. SOP requires advance planning in order that subjects may be composed together.

For example, consider the subconcern of **Add library resource** for adding a new book to the library catalogue. This subconcern is realised in terms of a subject called `AddNewBook`. The librarian supplies the author, the title and the number of copies. The system adds the new resource to the catalogue and determines a suitable location for the resource from author details. Based on this informal description, the subject designer can (modularly) identify the main objects as viewed from the perspective of this feature:

- The system used by the librarian is represented by a `ResourceManager` object.
- A `Book` is a kind of `Resource`.
- One or more `Copies` of a `Book` are created.
- The librarian supplies `Author`, `Title` and `NumberOfCopies` to the `ResourceManager`.
- The `ResourceManager` object assigns a `Location` to each `Resource`.

Furthermore, the librarian is the actor who interfaces with a `ResourceManager`. The external properties supplied by the actor include the book details and number of copies being introduced. All other objects including `Book`, `Copy` and `Location` are part of this concern's implementation.

However, as the following demonstrates, no further meaningful modular activity is possible at this stage. In `AddNewBook`, a unique integer identifier is associated with each copy of a new book. No other information needs to be recorded. For this subject it is sufficient to use an `int` array to store the identifiers. However, from the **Borrow library resource** requirement we are aware

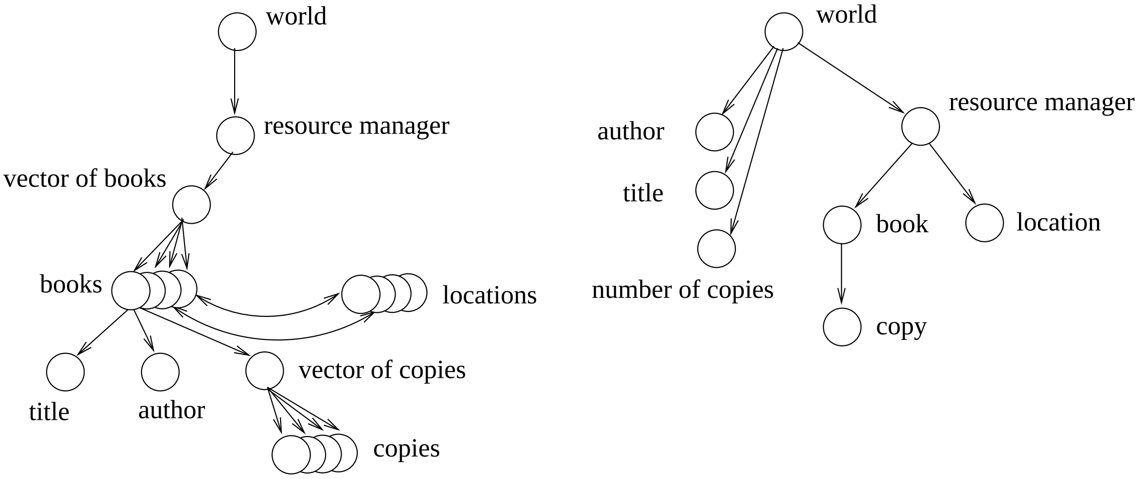

Figure 8-1: Sketches of an object graph (left) and ownership tree (right) for the AddNewBook concern

that borrower related information will be associated with copies. Thus in view of composition, the subject author instead should use objects of class Copy to store the integer identifier. Thus, modular development of subjects is possible, but subject developers must collaborate in order to ensure that their subject can be composed without further changes.

In SOP the subject developers must collaborate on many aspects of interaction, not least of which is the issue of desirable interaction. In exchange for greater certainty in interaction correctness, SAPS formally delays the point at which modular development commences until a mutually compatible ownership structure is established. When subjects are developed in concert, the establishment of ownership structures should be done before independent work on subjects begins. Ownership trees, first presented in Section 6.2 in page 103, are a useful way of communicating the ownership structure.

To create an ownership tree, it is necessary to understand the intended ownership structure, which in turn is understood from a sketch of the object graph for a subject. For example, a sketch of the collaboration implemented by subject AddNewBook is represented by an object graph shown in the left diagram of Figure 8-1. To create an ownership tree, one separates the external objects from the internal ones: title and author are properties of books that are supplied by the librarian; the number of copies is also an externally determined property. For the internal objects, each book is associated with a location for storage and a location may store many different books; books and locations must have the same owner in order to enable them to reference each other. A book is responsible for keeping track of all copies of that book, making each book the conceptual owner of its copies.

The key elements of the ownership structure are represented by an ownership tree sketch in the right diagram of Figure 8-1. The sketch elides the details of data structures used in the implementation. The purpose of the diagram is to convey the main ownership properties.

This ownership tree can be used to aid subject implementation. Figure 8-2 shows an implementation for subject AddNewBook. The ResourceManager.addNewBook(..) collaboration takes two String type parameters denoting the new book, and a single int type parameter representing the number of copies to be added. Note that the String class is immutable. Objects of type String are treated as elements of value type and require no context identifiers. Immutable objects

```
subject AddNewBook {
  class ResourceManager {
    Vector<0, 0> resource;
    void addNewBook(String title, String author, int noCopies) {
      Book<0> book = new Book<0>(title, author, noCopies);
      book.location = new Location<0>(author);
      resource.add(book);
    }
  }
  abstract class Resource {
    Vector<0, 0> copies;
    String title;
    Location<1> location;
    Resource(int noCopies) {
      while((noCopies--) > 0) {
        int id = ID.newID();
        Copy<0> copy = new Copy<0>(id);
        copies.add(copy);
      }
    }
  }
  class Book extends Resource {
    String author;
    Book(String title, String author, int noCopies) {
      super(noCopies);
      this.title = title;
      this.author = author;
    }
  }
  class Copy {
    int id;
    Copy(int id) {
      this.id = id;
    }
  }
  class Location {
    Location(String author) { /* determines location based on bibliographic details */ }
  }
}
```

Figure 8-2: The `AddNewBook` subject in the Library Management System

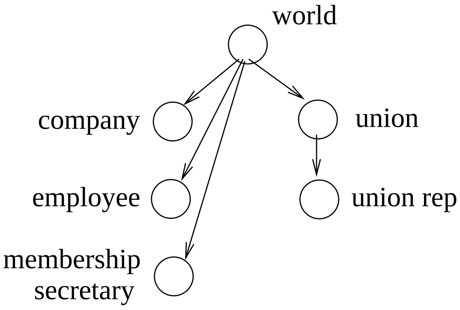

Figure 8-3: A sketch of the ownership tree for the Union set of concerns

have global ownership; the owner is implicitly `world`. The books and locations are owned by the `ResourceManager` object as indicated by exp_0. Copies are created in the scope of a `Book` instance which also owns them.

Sketches of ownership trees for different subjects should be similar. For instance, other subjects in the LMS that manipulate books, locations and copies should have the same ownership structure for those objects.

We do not prove that the length of time spent on development is less with SAPS than without SAPS. To do so requires timing different teams of similar experience and with similar familiarity with the application domain. However, by making ownership information explicit in types, we believe that SAPS leads to productivity improvements. First, by eliminating certain interaction problems, and secondly, by helping to pinpoint the cause of other interaction problems as explained in the following.

8.1.3 Interaction Problems

Section 4.4 on page 59 has shown that the scattering of data concerns can lead to interaction problems. SAPS was partly motivated by interaction problems that required invasive subject modifications. With SAPS, in order to be composable, corresponding elements must define compatible types including compatible context identifiers. Now, interaction problems are anomalies that occur despite corresponding elements having compatible types. Even then, SAPS remains useful because the owner represents a boundary within which object state may be changed. This is an improvement over SOP programs without SAPS which do not define a boundary on aliasing.

Revisiting the Union example in Section 4.4 on page 59, recall that the composition of subjects `JoinUnion`, `Dismiss` and `Retire` manifested an interaction problem. The problem was caused by uncontrolled aliasing of union representative objects, such that a link between union representatives and members which was previously assumed to be invariant became broken when the `Retire` subject was introduced.

With SAPS, one approach is to develop these subjects independently from each other, using SOT to do conceptual modelling as described in Section 5.2.4 on page 87. These subjects have different ownership structures which translate to incompatible Subjective Ownership Types at join points. Consequently, subjects cannot be composed using the composition rules we defined in Chapter 7.

A better approach is to use the strongest mode in view of composition. These subjects are intended to be composed together, so it makes sense to identify the common ownership structure and then define the subjects in relation to it. A sketch of such a structure is shown in Figure 8-3. All objects in the diagram except the union representatives are `world` owned. Figure 8-4 shows the main details of these subjects. SAPS has eliminated the original interaction problem by making aliasing an explicit concern and by creating a well structured subject-oriented program based on the SOT model of alias protection.

Interaction problems in SOP can occur in spite of compatible Subjective Ownership Types at the join points. In such cases SAPS helps to detect interaction problems because it constrains object aliasing. For example, consider the ownership tree in Figure 8-5 which depicts the main objects in a Lift system created using SAPS. Suppose that during testing a problem is discovered with the software controlling the opening and closing of lift `doors`. From the ownership structure it is clear that only objects inside the ownership context of `doors` can directly affect the state of the `doors` object. SOT direct the maintainer to analysing code in all subjects which can affect `doors`. Specifically, this includes code which contributes to the state and behaviour of `lift`, `floor selection button`, `stop button`, `door open button`, `motor` and `doors`. Any code which contributes exclusively to the state and behaviour of `building`, `floors` and `buttons` cannot change the state of the `doors` object.

8.1.4 Reuse and Reusability

Our position on reuse stated that improving opportunities on reuse depends on ideas that are of value to the original developer as well as the reuser. In the conclusion to Chapter 4 on page 68 we stated our belief that Alias Protection Systems will be useful to subject developers. APSs already help object-oriented programmers to create well structured object-oriented programs that control alias exposure. The construction of subjects is essentially an object-oriented activity, so there is also a benefit the subject developer.

Reuse in SOP is most commonly associated with composition; however, it is also possible to subclass individual classes from an existing subject when creating a new subject. For instance, to create the `AddNewPeriodical` subject, it is necessary to introduce a new operation into `ResourceManager` and to define class `Periodical`. In LMS, periodicals differ from books in having only a single copy and an additional field denoting the category. Instead of composition, the subject author may choose to use inheritance or delegation to define `Periodical`.

Intentional construction of reusable abstractions is supported in SAPS in two ways. The subject author may define uncomposable classes. For the most part, the decision to create uncomposable classes is made using the heuristic specified in Section 6.4.1 on page 122. There is a notable exception: in Section 8.4 on page 199 uncomposable classes are used for security. In the LMS, no classes identified during requirements analysis require parameterisation with respect to their ownership properties. Consequently all are composable.

Reusability is also supported through unknown context identifiers. An `unk` represents a choice of `exps` so a subject that employs `unks` can adapt to a number of different ownership structures that can be represented using `exps`. For example, Section 5.4 on page 92 described the reusability requirement on the Composite design pattern. `unks` can be used to create a reusable definition for this pattern. Figure 8-6 shows the pattern implementation annotated with Subjective Ownership Types. Two `unks` are used: unk_k denotes the owner of the children objects with respect to the composite object; unk_m denotes the owner of object returned by the collaboration realised by the composite structure. With slight modifications, subjects `CADdraw` (Figure 5-14 on page 94) and `FileSystemSize` (Figure 5-15 on page 95) can be composed with subject `Composite`.

```
subject JoinUnion {
  class MembershipSecretary {
    Union<world> theUnion;
    Employee<world> employee;
    void joinUnion() { theUnion.join(employee); }
  }
  class Employee {
  }
  class UnionRep {
    String repName;
  }
  class Union {
    Hashtable<0, world, 0> member2rep;
    Vector<0, 0> reps;
    join(Employee<world> e) { /* assign a rep to an employee */ }
  }
}

subject Dismiss {
  class Company {
    Vector<0, world> employees;
    void dismiss() { ... e.appeal(); ... }
  }
  class Employee {
    Union<world> theUnion;
    String appeal() { return theUnion.getRepName(this); }
  }
  class UnionRep {
  }
}

subject Retire {
  class UnionRep {
    Union<2> theUnion;
    void retire() { theUnion.retire(this); }
  }
  class Union {
    Hashtable<0, world, 0> member2rep;
    Vector<0, 0> reps;
    void retire(Member<0> m) { ... }
  }
  class Member { }
}
```

Figure 8-4: `JoinUnion`, `Dismiss` and `Retire` subjects annotated with SOT

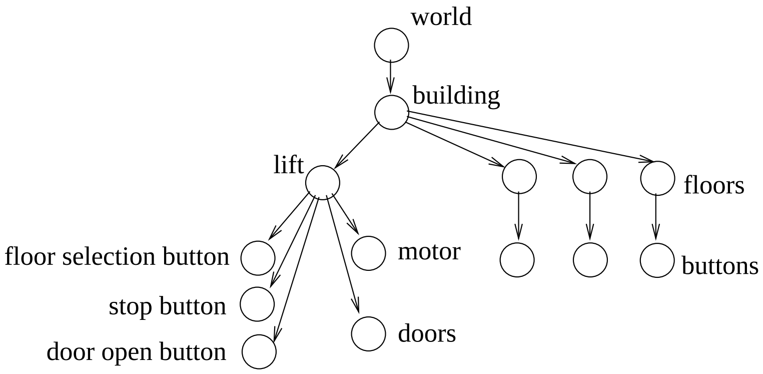

Figure 8-5: A sketch of the ownership tree for a Lift system

```
subject Composite {
  unk k, m;
  abstract class Component {
    abstract Object<m> doAction();
  }

  abstract class Composite extends Component {
    Vector<0, k> children;

    Object<m> doAction() {
      Iterator<0, k> it = children.iterator();
      while(it.hasNext()) {
        Component<k> c = (Component<k>)it.next();
        perChild(c);
      }
      return null;
    }
    abstract void perChild(Component<k> c);
  }
}
```

Figure 8-6: Composite design pattern as a subject annotated with SOT

8.2 Feature-Oriented Development

One strength of Subject-Oriented Programming is the ability to mix-and-match features for on-demand remodularisation. The subject composer, in the role of a component vendor, can supply software containing precisely those features that are required by the client. The supplied software takes the form of a traditional black-box component that will be used by the client in his application development.

The exact environment in which the component will be used is not known by the vendor, but the component developed with SOP may inter-operate with client software that maliciously or accidentally subverts its state through representation exposure. It is important that the vendor has complete confidence in the encapsulation of the component's mutable state *for all combinations of features*, such that the only way the state can be changed is through interface operations. Furthermore, the component may be used in an environment where the client is possibly unaware of either SOP or SAPS.

SAPS extends the benefits of alias protection to component development with SOP. Each feature is developed as a subject. In order to compose features successfully the subject designers must agree on the way corresponding classes use objects: representation object in one subject is also a representation object in all other subjects. Any features introduced as enhancements at a later date must also conform to this model of encapsulation. SAPS is downwardly restrictive; that is, clients using a component developed with SAPS need not be aware of Subjective Ownership Types used in its development.

The composition rules defined in the preceding Chapter have a monotonic effect: composition can introduce new behaviour to objects but composition does not change the object owner. With respect to feature F, composing F with other features does not change the ownership context of any objects created or referenced within the behaviour specified by F. This is precisely the property required to safely mix-and-match features.

Before a set of collaborating subjects can be implemented, the development teams must agree on the ownership properties of the common objects manipulated by features. For the LMS requirements in Section 8.1.1 on page 184 these properties can be summarised as follows:

- Resources (book and periodicals) are owned by the resource manager. Any external referencing to these resources should be done using value identifiers.
- The copies of a resource are owned by the resource.
- Borrowers are external to the resource manager. Consequently, external references to copies is done through value identifiers.
- Fines are owned by borrowers that collect them.

Figure 8-7 depicts a sketch of the ownership structure common to these features. Having presented the `AddNewBook` subject in Figure 8-2, attention now turns to the ownership details of the other features that make up the LMS.

The `RemoveResource` subject deletes a resource from the library catalogue. The item to be removed can be any valid subtype of `Resource`. The concrete type of the item is not relevant to the present concern, so only class `Resource` is declared. This subject manipulates the same objects and has the same SOT declarations as `AddNewBook`:

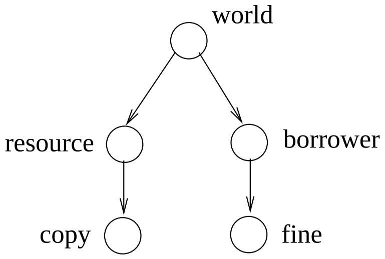

Figure 8-7: A sketch of the ownership tree common to the subjects making up the Library Management System

```
subject RemoveResource {
  class ResourceManager {
    Vector<O, O> resource;
    void removeResource(String title, String author) { ... }
    ...
  }
  class Resource {
    Vector<O, O> copies;
    ...
  }
  ...
}
```

The ordering of resources is the responsibility of the `OrderBook` and `OrderPeriodical` subjects. For books, this subject is similar in design to `AddNewBook` except that no shelf location is associated with the resource until the order arrives. The resource is still added to the catalogue but a special flag indicates that the item is not yet available for browsing or borrowing. The search for library resources is implemented by the `SearchByTitle` subject. Given the title of a resource, the collaboration implemented by this subject returns the location of this resource.

Subject `BorrowBook` associates a book copy with a borrower. The copy can reference the borrower because the borrower's owner context is external to that of the `Copy` object. The borrower stores the `int` identifier denoting the copy.

```
subject BorrowBook {
  class Book {
    Vector<O, O> copies;
    void borrow(Borrower<world> b, int copyId) { ... }
    ...
  }
  class Copy {
    Borrower<world> b;
    ...
  }
  abstract class Borrower { ... }
  class UndergraduateBorrower extends Borrower { ... }
  class PostgraduateBorrower extends Borrower { ... }
}
```

When returning a book late, a fine is issued. The `ReturnBook` subject creates `Fine` objects inside `Borrower` objects. In SOT, only an object inside `Borrower` can create objects whose owner is this `Borrower`. It is expected that methods `Book.return(..)` and `Book.borrow(..)` will be activated at runtime using a barcode scanner object that is owned by the `ResourceManager`:

```
subject ReturnBook {
  class Book {
    void return(int copyId) { ... }
    ...
  }
  class Copy {
    Borrower<world> b;
    ...
  }
  class Borrower {
    Vector<0, 0> fine;
    ...
  }
  class LibrarianBorrower extends Borrower { ... }
  class PublicBorrower extends Borrower { ... }
  class Fine { ... }
}

subject Scanner {
  abstract class ResourceManager {
    Scanner<0> s;
    abstract void borrow(int copyId);
    abstract void return(int copyId);
    ...
  }
  class Scanner implements Runnable { ... }
}
```

The requirement for concurrency has been identified as necessary to facilitate multi-user access to the Library Management System. Updating of the library records requires a write lock to exclude all readers. Concerns such as AddNewBook and RemoveResource require a write lock. Each activation of searching behaviour increments the count of readers. This behaviour is implemented by subject Synch:

```
subject Synch {
  abstract class SynchClass {
    int activeReaders, activeWriters;
    synchronized void waitWriterReaders() { ... }
    synchronized void waitReaders() { ... }
    void decrementWriters() { activeWriters--; }
    void decrementReaders() { activeReaders--; }
    ...
  }
}
```

The mixing and matching of features occurs within the composition specification. For example to supply a component that contains features AddNewBook, RemoveResource, OrderBook, SearchByTitle, BorrowBook, ReturnBook and Synch, the composition specification is given by:

```
compose AddNewBook, RemoveResource, OrderBook, SearchByTitle, BorrowBook, ReturnBook, Synch
  into LMS;
mergeByName;
bracket ResourceManager.addNewBook with before Synch.SynchClass.waitWriterReaders
                                        after Synch.SynchClass.decrementWriters;
bracket ResourceManager.removeResource with before Synch.SynchClass.waitWriterReaders;
                                            after Synch.SynchClass.decrementWriters;
bracket ResourceManager.orderBook with before Synch.SynchClass.waitWriterReaders;
                                       after Synch.SynchClass.decrementWriters;
bracket Book.borrow with before Synch.SynchClass.waitWriterReaders;
```

```
                         after Synch.SynchClass.decrementWriters;
bracket Book.return with before Synch.SynchClass.waitWriterReaders;
                         after Synch.SynchClass.decrementWriters;
bracket ResourceManager.search with before Synch.SynchClass.waitReaders;
                                    after Synch.SynchClass.decrementReaders;
```

The output subject contains the ResourceManager class. This class is the interface to the LMS component. This component can be used in applications requiring LMS functionality. The interface does not to expose any representation objects used in the implementation while still making it possible for the LMS component developers to reap the benefits of feature-based decomposition. To a client who is unaware of Subjective Ownership Types, the functional interface is given by:

```
class ResourceManager {
  void addNewBook(String title, String author, int noCopies) { ... }
  void removeResource(String title, String author) { ... }
  void orderBook(String title, String author, int noCopies) { ... }
  String searchByTitle(String title) { ... }
}
```

Borrowing and returning of resources is not a part of the LMS functional interface. This functionality is part of the implementation of the ResourceManager.

8.3 System Integration: A Cross-Cutting Concern

Subject-Oriented Programming is a technology that enables Multi-Dimensional Separation of Concerns. In addition to the modularisation of concerns in the feature dimension, SOP can also modularise cross-cutting concerns in other dimensions. The preceding Section showed the utility of SAPS in feature-oriented development. In order for SAPS to be useful to the subject-oriented developer, SAPS must be able to express the different representation containment requirements demanded by a range of SOP applications. In this Section, we demonstrate the utility of SAPS with respect to a cross-cutting concern; it has been shown that system integration is a cross-cutting concern in object-oriented software [117, 118].

Suppose one constructs a system that integrates the behaviour of several binary digits. Each Bit is defined as:

```
subject JustABit {
  class Bit {
    boolean value;
    void set() { value = true; }
    void clear() { value = false; }
    boolean get() { return value; }
  }
}
```

The integration concern is to synchronise the states of particular Bit pairs. Associations (relations) between pairs of bits are created dynamically. There are two kinds of association: Equality and Trigger. Figure 8-8 shows three Bit objects connected by Equality relations. Equality propagates set() and clear() calls from the left to the right side and vice versa. Trigger propagates them in one direction only.

When trying to map these design structures into object-oriented programs, one finds that integration issues become tangled in the implementation of class Bit. In a purely object-oriented solution the Bit class stores references to ends of relations; the code for set() and clear() also implements

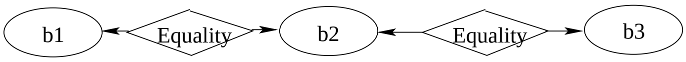

Figure 8-8: Integration of Bits

the propagation of effects to related Bits. Looking at the Equality association, we see that the lack of an abstract implementation for Equality makes the concern hard to understand. The Bit and the Equality have been hardwired together and are difficult to separate. Design patterns improve the designs a little but fail to achieve clear separation because the pattern code is still tangled with the Bit class definition [116].

The SOP designs for the Equality association improve on object-oriented solutions. Subject Equality cleanly disentangles this concern from subject JustABit. However, code created through system integration seldom runs in isolation. More likely, it is a subsystem that inter-operates with other subsystems, possibly also created by integration. For example, a collection of Bits may be aggregated into a binary instruction to perform a bit shift operation or Bits may be organised into temporary groups based state patterns. Suppose that the Bit integration scenarios for the Equality relation translate into design variants for the Equality subject. The differences can be distilled into different ownership structures created by combinations of bits and Equality associations. We observe three variants:

1. The Association object is co-owned by the Bits that are connected by it.
2. The Association is split into two parts, with each Bit owning an Association object that refers to the other Bit.
3. The Association is owned either by the same object as the Bits or another object outside the Bit owner.

The first solution is the best case for Equality associations that the integrator wishes to hide inside the Bits. Once created, only the Bit objects in the association would be able to modify associations directly. SAPS cannot implement this model because every object must have exactly one owner.

The second solution is well-suited to Trigger associations. Trigger propagates one way, so only the object at the source needs to maintain the association. For Equality, two Association objects are created with each referencing the Bit at the end opposite. Like the first proposal, this approach hides the Association object in the representation of the Bit. There may be many associations between pairs of Bits and associations may be added or removed dynamically. In order to prevent representation exposure, the corresponding associations have an int id field. The ids of the two ends of an association have the same value. This solution is shown in Figure 8-9.

The third model allows the integrator to decide on the owner of the Associations and the Bits. In Figure 8-10, example client code creates Bits and Associations whose owner is given by $\texttt{exp}_0$. Associations reference Bits and vice versa. Consequently, Association and Bit should always have the same ownership context. In this model, the Association can be directly accessed and modified by the Bit client.

These designs demonstrate the flexibility and the limitations of SAPS for adapting to different ownership structures demanded by a cross-cutting concern. SAPS can model two out of three

```
subject Equality {
  class Bit {
    Vector<0,0> assoc = new Vector<0,0>();
    boolean busy;  // used to prevent infinite loops
    void associate(Bit<1> b) {
      Association<0> a = new Association<0>(b);
      assoc.add(a);
      b.assoc_back(this, a.id);
    }
    void assoc_back(Bit<1> b, int id) {
      Association<0> a = new Association<0>(b, id);
      assoc.add(a);
    }
    void set() {
      for(Iterator<0,0> it = assoc.iterator(); it.hasNext(); ) {
        Association<0> a = (Association<0>)it.next();
        if(!busy) {
          busy = true;
          a.b.set();
          busy = false;
        }
      }
    }
    // code for clear() elided
  }

  class Association {
    int id;     // unique key identifying this association
    Bit<2> b;
    Association(Bit<2> b) { ... }
    Association(Bit<2> b, int id) { ... }
  }
}


// composition specification used
compose JustABit, Equality into Integration;
mergeByName;
order Equality.Bit.set after JustABit.Bit.set;

// example client code
Bit<0> b1 = new Bit<0>();
Bit<0> b2 = new Bit<0>();
Bit<0> b3 = new Bit<0>();
b1.associate(b2);
b2.associate(b3);
```

Figure 8-9: `Equality` subject with encapsulated associations

```
subject Equality {
  class Bit {
    boolean busy; // used to prevent infinite loops
    Relation<1> rel;
    void clear() {
      for(Iterator<1,1> it = rel.r.iterator(); it.hasNext(); ) {
        Association<1> a = (Association<1>)it.next();
        if(!busy) {
          busy = true;
          if(a.l == this) a.r.clear();
          if(a.r == this) a.l.clear();
          busy = false;
        }
      }
    }
    // code for set() elided
  }

  class Association {
    Bit<1> l, r;
    Association (Bit<1> l, Bit<1> r, Relation<1> rel) {
      this.l = l;
      this.r = r;
      l.rel = rel;
      r.rel = rel;
      rel.add(this);
    }
  }

  class Relation {
    Vector<1,1> r = new Vector<1,1>();
    void add(Association<1> a) { r.add(a);  }
  }
}

// composition specification
compose JustABit, Equality into Integration;
mergeByName;
order Equality.Bit.clear after JustABit.Bit.clear;

// example client code;
Relation<0> r = new Relation<0>();
Bit<0> b1 = new Bit<0>();
Bit<0> b2 = new Bit<0>();
Bit<0> b3 = new Bit<0>();
Association<0> a1 = new Association<0>(b1, b2, r);
Association<0> a2 = new Association<0>(b2, b3, r);
```

Figure 8-10: `Equality` subject with exposed associations

ownership structures presented here. However, SOT are not intended for co-ownership of the kind described in the first case. Co-ownership is discussed as part of future work in Section 9.2 on page 215.

8.4 Using Uncomposable Classes for Security

Public key cryptography is one of a number of security tools in distributed systems. The implementation of public key cryptography should therefore be secure. Specifically, the random numbers used in the generation of public and private keys should not be accessible outside the RSA algorithm [107].

The implementation of RSA should be reusable in a number of different settings. In any application of RSA, the objects that represent private keys should be confined to their application and under no circumstances should an external client gain access to the private key.

Subject-oriented programming achieves the goal of making the public key cryptography algorithm modular by defining all pertinent functionality within the RSA subject (shown in Figure 8-11). Within SOP, making random numbers secure is due to Subjective Ownership Types. The functionality associated with generating random numbers used in keys is associated with method `KeyFactory.genKeyPair(..)`. We have implemented `KeyFactory` as an uncomposable class. Had we made `KeyFactory` composable it would have been open to attack through advanced SOP composition rules. For instance, bracket relationships on method call sites could be used to intercept method calls to private random number generators (line 17). Uncomposable classes never have internal join points, making this method of implementation secure for future extensions.

Class `Key` contains the algorithm for RSA encryption. Class `Key` is composable, making it possible to introduce a more efficient encryption algorithm by subject composition at any time in the future. In order to make the encryption facilities of RSA reusable in different settings, the owner of the message to encrypt is given by unk_k. The encrypted message owner is given by unk_m. The encrypted message can be seen where its decrypted counterpart cannot be: giving rise to ucirc $k \leq m$. Fields `Key.mod` and `Key.exp` are owned by the owner of their `Key` object. The components making up a key should be accessible to the key's owner but never outside.

Besides being uncomposable, class `KeyFactory` has priv_key_owner as the ownership parameter binding to the owner of the private key it receives as argument. Field `KeyFactory.rnd` is owned by this instance of `KeyFactory`. No object outside any `KeyFactory` instance can change the state of the object referenced by `KeyFactory.rnd`. This privilege is granted only to this `KeyFactory` and other objects inside this `KeyFactory` that have been given a permission to do so.

Subject `SecureTerminal` (Figure 8-12) implements secure transmission of messages. As in all subject-oriented programs, there is no explicit connection to other subjects other than in the composition specification. In principle, `SecureTerminal` need not be composed with RSA but with any subject(s) implementing the undefined functionality. The final application would be expected to involve other subjects, such as for doing I/O and so on. However, the presented subjects are composed on the basis of the following composition specification:

```
compose RSA, SecureTerminal into RSATerminal;
mergeByName;
```

In class `SecureTerminal.Key`, messages to encrypt have this `Key` as owner, given by exp_1. The encrypted messages can be aliased globally, given by world. During composition, unk_k – the owner of decrypted messages – resolves to exp_1 in class `Key`; unk_m – the owner of encrypted messages – resolves

```
subject RSA {
  unk k, m;
  ucirc k <= m;
  class Key {
    BigDecimal<1> mod;
    BigDecimal<1> exp;
    String<m> encrypt(String<k> msg) {
      // encrypt using mod and exp.
    }
    String<k> decrypt(String<m> msg) {
      // decrypt using mod and exp.
    }
  }
  class KeyFactory<priv_key_owner> {
    Random<0> rnd = new Random(System.currentTimeMillis());
    void genKeyPair(Key<world> pub, Key<priv_key_owner> priv) {
      double d = rnd.nextDouble();
      // use the random value to compute and set the key components.
    }
  }
}
```

Figure 8-11: Subject containing RSA algorithm

to world. The resolution satisfies the subject-level ucirc in subject RSA. SecureTerminal.KeyFactory is defined as an uncomposable class. It is included for declarative completeness: in order to enable each subject to typecheck correctly.

The Terminal class owns the decrypted messages. privateKey is confined to the Terminal object as required, but not to the SecureTerminal subject. Other subjects' classes composed with Terminal can see and change privateKey. This is precisely the effect we require: any additional functionality introduced explicitly through composition should be able to manipulate privateKey; other subjects must specify the same owner for privateKey, making the effect of composition predictable.

In line 12, keyFactory is also owned by this Terminal instance. The second parameter binds the priv_key_owner parameter. This must be exp$_0$ in order for the method call in line 14 to typecheck correctly. Operations send(..) and receive(..) respectively dispatch the outgoing message and accept incoming messages.

The SAPS solution is superior both to a pure subject-oriented solution and to an object-oriented solution created with the aid of Confined Types [127]. Compared to SOP, by making KeyFactory uncomposable, SAPS ensures that secrecy is afforded to the algorithm for generating keys. In the case of subject RSA, SAPS restricts the set of objects that can observe the private keys and, in the case of the SecureTerminal application, guarantees that no object other than the Terminal object (and objects owned by the Terminal) can view or modify the private key.

Compared to Confined Types, the RSA subject is much more compact than the RSA package in Java with Confined Types. Confined Types require the programmer to declare and use anonymous methods (see Section 5.1.3). Anonymous methods may require additional classes to be introduced which would not be there if Confined Types were not used [127]. The SecureTerminal subject has no syntactic dependencies on the RSA subject. SOT also allow a number of different ownership structures to be defined for use in conjunction with RSA instead of the binary confined/unconfined

```
subject SecureTerminal {
  abstract class Key {
    abstract String<world> encrypt(String<1> msg);
    abstract String<1> decrypt(String<world> msg);
  }
  abstract class KeyFactory<priv_key_owner> {
    abstract void genKeyPair(Key<world> pub, Key<priv_key_owner> priv);
  }
  class Terminal {
    String<0> msg_in, msg_out;
    Key<0> privateKey;
    KeyFactory<0,0> keyFactory;
    Terminal(Key<world> publicKey) {
      keyFactory.genKeyPair(publicKey, privateKey);
    }
    String<world> send(Key<world> publicKey) {
      return publicKey.encrypt(msg_out);
    }
    void receive(String<world> msg) {
      msg_in = privateKey.decrypt(msg);
    }
  }
}
```

Figure 8-12: Subject implementing a secure terminal application

modes of Confined Types.

An intriguing solution to object containment has been demonstrated within Object Teams [55]. Extension of `RSA` with `SecureTerminal` functionality is achieved with family polymorphism, or team inheritance. This solution carries all the benefits of subtyping which are presently lacking in subject composition. The Object Teams solution is based on the Confined Types model: confined roles are encapsulated within their enclosing team instance.

8.5 Using exps for Composition Restriction

Examples in Chapter 6 and in this Chapter have shown that unknown context identifiers can be used to delegate design decisions on contexts to another subject. Explicit contexts can do the opposite: they can constrain subjects to particular compositions in order to ensure that only functionally valid compositions are specified.

Consider a strategy game where one or more human players compete against one or more computer opponents. Each player (human or computer) controls an army of droids that can be arbitrarily organised into squads. The game objective is to capture the oppositions' flags. To achieve the aim, players split armies into squads and deploy some *strategy.* Each squad then plays a role in the strategy. The role involves reaching some destination waypoint as defined by the strategy. For example the `Surround` strategy involves positioning squads at points on the circle circumference defined by the target at the centre of the circle. Within each squad the droids are put into a *formation.* Each formation has different fighting characteristics. For example, the `Square` formation is good for defending a position from multi-directional attacks. The artificial intelligence engine is able to select both the strategy and the formation at each stage in the game but a strong (human) player should be able to win by making better strategic and, occasionally, formation decisions.

```
subject Game {
  class Player {
    Vector<0,0> droids;
    Vector<0,0> squads;
  }
  class Squad {
    Droid<1> commander;
    Vector<0,1> droids;
  }
  class Droid { }
}
```

Figure 8-13: Strategy game composition interface

What makes this game different from its competitors is the facility for specifying new strategies and formations. These can be uploaded by players and added to the set of control options. Strategies and formations are defined as SOT-annotated subjects. These subjects are integrated into the game using subject-oriented composition rules.

The game architects require that no user specified strategy or formation lets one player take control of droids in another player's army. This is enforced by restricting the composition interface to the design given in Figure 8-13.

The players would like to ensure that subjects for new strategies and formations are deployed correctly. That is, the composition is restricted to particular correspondences such that the composed subject functions as intended. The aim is to restrict composition in order to eliminate compositions that are known to lead to anomalies. For example, one anomalous interaction is identified in the combination of subjects for the *flanking maneuvre* strategy and the *keep distance* formation:

- The flanking maneuvre (subject `FM` in Figure 8-14) is a well-known military strategy. It involves splitting one's army into two squads. A smaller squad is left to resist the attacking force and a bigger squad goes around and attacks the opponent from behind.

- The keep distance formation unbunches droids, putting each droid the same distance from its neighbours. Two subjects can be created here: in Figure 8-14, `KD1` uses `unks` to denote `Droid` owner and `KD2` uses `exps`. The `unks` in the definition of `KD1` make this subject more reusable.

Composition of `FM` with either `KD1` or `KD2` is desirable because the keep distance formation gives the smaller squad an appearance of being bigger than it is in reality in order to mislead the enemy. Two composition specifications can be created for integrating subjects `Game`, `FM` and `KD1` based on two resolutions of `unk`$_k$:

1. `unk`$_k$ resolves to `exp`$_0$ in `KD1.Aggregation`: classes `Game.Player.droids<0,0>`, `FM.FlankingManeuvre.droid<0,0>` and `KD1.Aggregation.droids<0,k>` correspond.

2. `unk`$_k$ resolves to `exp`$_1$ in `KD1.Aggregation`: classes `Game.Squad.droids<0,1>` and `KD1.Aggregation.droids<0,k>` correspond. `unk`$_k$ resolves to `exp`$_2$ in `KD1.Droid` by resolution propagation.

The first composition contains an anomaly that causes droids of the smaller squad to keep distance with droids of the bigger squad, creating one long chain instead of cleanly splitting into two squads. The second composition produces the intended result. To ensure correct deployment of

```
subject FM {
  class FlankingManeuvre {
    Position<1> target2attack;
    Vector<0,0> droids;
    Vector<0,0> squads;
    void do_FM() {
      Squad<0> front = ...
      Squad<0> flank = ...
      squads = new Vector<0,0>(front, flank);
      front.attackDirect(target2attack);
      flank.round(target2attack);
    }
  }
  ...
}

subject KD1 {  // using unks
  unk k;
  class Aggregation {
    Vector<0,k> droids;
    void do_KD() {
      for(Iterator<0,k> it = droids.iterator(); it.hasNext(); ) {
        Droid<k> d = (Droid<k>)it.next();
        d.neighbour_left = findNeighbour();
        d.neighbour_right = findNeighbour();
        ...
      }
    }
    ...
  }

  class Droid where 1 <= k {
    Droid<k> neighbour_left;
    Droid<k> neighbour_right;
  }
  ...
}

subject KD2 {  // using exps
  class Aggregation {
    Vector<0,1> droids;
    void do_KD() {
      for(Iterator<0,1> it = droids.iterator(); it.hasNext(); ) {
        Droid<1> d = (Droid<1>)it.next();
        d.neighbour_left = findNeighbour();
        d.neighbour_right = findNeighbour();
        ...
      }
    }
    ...
  }

  class Droid {
    Droid<2> neighbour_left;
    Droid<2> neighbour_right;
  }
  ...
}
```

Figure 8-14: Subject `FM` and 2 versions of subjects `KD` using unks and exps

the keep distance concern, we replace `KD1` by `KD2` which uses `exps` instead of $\mathtt{unk}_k$. Now the second composition is the only valid option.

The alternative of using `ucircs` with `KD1` does not work. For example, one may try to specify:

```
class Aggregation where 1 <= k {...}
```

All definitions inside class `Aggregation` satisfy this definition. But the resolution constraints of the output class are not based on the declared `ucircs` but on the behaviour defined within the composed class. Thus $k = 0$ will still be in the resolution set of $\mathtt{unk}_k$ in the class to which `Aggregation` forwards. `ucircs` are intended to prevent representation exposure in the input subject. For composition constraints, `exps` should be used to convey a particular ownership structure.

8.6 Limitations

This Section describes the known limitations of SAPS. It is important to isolate the issues which are specific to the decisions taken in the creation of SAPS from the limitations of SOP as a paradigm. The latter was reviewed in Chapter 3 on page 22 where SOP was compared to other technology for advanced separation of concerns. SOT as an APS has limitations: for instance, it does not support dynamic aliases which are necessary to support subject design with respect to certain object-oriented idioms. The challenges in providing support for dynamic aliases and other aliasing modes are discussed in future work on page 215.

The following two Subsections deal with two fundamental limitations of SAPS. When concerns to be composed have incompatible views of a domain, the differences may translate to incompatible ownership structures. Incompatible domain views may force changes to the subject structure in order to accommodate the SAPS model of composition. The second problem concerns the selection between composable and uncomposable classes. The system of explicit contexts is more rigid than ownership parameterisation. The rigidity enables desirable restrictions on subject composition as seen in Section 8.5 on page 201 but may prove too restrictive during evolution.

8.6.1 Incompatible Domain Views

This limitation of SAPS concerns domain modelling. To enable clean separation of concerns a subject defines only those abstractions which pertain to addressing its concern. A problem can occur if domain views with inherently incompatible ownership structures need to be composed. For example, consider the development of a graphics suite. The system is decomposed into subjects such that one subject designer can implement each algorithm. The following two algorithms are identified:

- A blurring algorithm recalculates the colour at each pixel from the values of its immediate neighbouring pixels.
- A magnification algorithm computes the colour at the current pixel based on the values in its region. A region is an array of neighbouring pixels.

These two algorithms are implemented as subjects `Blur` and `Magnify` shown in Figure 8-15. Subject `Blur` defines classes `Picture` and `Pixel` only. Blurring is performed per pixel. The pixels are owned by the picture that they represent. The neighbouring pixels are obtained dynamically by

```
subject Blur {
  class Picture {
    Pixel<0,0,0>[][] p;
    Pixel<0> getLeftNeighbour(Pixel<0> px) { ... }
    Pixel<0> getRightNeighbour(Pixel<0> px) { ... }
    ...
  }
  class Pixel {
    Picture<2> inPic;
    void blur() {
      Pixel<1> leftP = inPic.getLeftNeighbour(this);
      Pixel<1> rightP = inPic.getRightNeighbour(this);
      ...
    }
  }
}

subject Magnify {
  class Picture {
    float magFactor;
    Region<0,0,0>[][] r;
    ...
  }
  class Region {
    Pixel<0,0,0>[][] p;
    Region<1> magnify() { /* magnify this region */ }
    ...
  }
  class Pixel {
    int blue, red, green;
    Region<2> inRegion;
    Picture<3> inPic;
    void calcValue() {
      /* calculate new blue, red, green for this pixel based on
         values in inRegion and the magnification factor in inPic */
    }
}
```

Figure 8-15: Subjects `Blur` and `Magnify`

sending a message to `inPic`. For efficiency, subject `Magnify` performs magnification one region at a time. A picture owns the region and the region owns the pixels. A call to `Region.magnify()` creates a new region whose pixel values are determined from the current region based on the magnification factor stored in `Picture.magFactor`.

Across these two subjects classes `Picture` and `Pixel` represent the same concept. Composition of subjects `Blur` and `Magnify` creates an efficient implementation where magnification and blurring can be applied on the same picture. However, the additional concept of region in subject `Magnify` introduces an additional layer of abstraction which affects the `exps` used in the definition. Observe that in `Blur.Pixel` variable `inPic` has type `Picture<2>` whereas in `Magnify.Pixel` this variable has type `Picture<3>`. Consequently, these two subjects cannot be composed.

We may attempt to use `unks` in subject `Blur` in order to enable variability between the contexts of a picture and its pixels. However, this is futile because in `Magnify` the pixels are in the representation context of `Region` and in `Blur` they are in the representation context of `Picture`. To make it possible to compose these subjects it is necessary to harmonise the context identifiers of corresponding elements either by adding the region concept to `Blur` or by flattening the ownership structure in `Magnify`. Either way, separation of concerns is affected: one subject has to be modified in order to ensure composability with another subject. This is an endemic problem of the subject composition model we have adopted. More flexible alias annotation systems may be better able to cope with incompatible domain views expressed by subjects.

8.6.2 Defining Composable and Uncomposable Classes

This limitation of SAPS concerns the definition of new classes. The choice is between composable and uncomposable classes. If for some reason it becomes necessary to modify a definition from composable to uncomposable or vice versa, there will be expensive repercussions. Section 6.4.1 on page 122 defined a heuristic for helping developers select what kind of class to define. In our experience the heuristic serves well and drastic changes are rare. However, an exception to the heuristic may occur when a new concern is added to an existing concern set.

Continuing with the example of the graphics suite, the drawing of a picture is performed one region at a time in relation to a colour map. A colour map is a function from the pixel value to the real colour of that pixel. The colour map for a picture is a property owned by the picture library. New picture creation and the drawing functionality is associated with subject `Base` shown in Figure 8-16. In an alternative implementation, these concerns may be developed as separate subjects but the current decomposition is sufficient to illustrate our point.

Later, copy and paste features are added. The proposed solution uses a `Region` object as a buffer for storing the copied fragment. The copy operation uses `PictureLibrary.buffer` to alias a clone of the marked-up region. The paste operation applies the buffer to the target image. Figure `CopyPaste` shows the subject that we would like to create.

A problem becomes apparent when we try to compose instance variables `Region.cm`. These fields correspond because they clearly represent the same object. The types are `ColourMap<2>` and `ColourMap<1>` in subjects `Base` and `CopyPaste` respectively. In `Base` the colour map is owned by the owner of the current region which gives rise to $\texttt{exp}_2$. In `CopyPaste` the colour map is owned by the picture library which also owns the current region, giving rise to $\texttt{exp}_1$. The system of `exps` cannot cope with dynamic hierarchy changes; it requires all objects referred to by context identifiers (both explicit and unknown) to have the same relative positions for all instances of a class.

```
subject Base {
  class PictureLibrary {
    ColourMap<0> cm;
    Picture<0> p1, p2;
    void newPic() {
      p1 = new Picture<0>(cm);
    }
  }
  class Picture {
    ColourMap<1> cm;
    Region<0,0,0>[][] r;
    Picture(ColourMap<1> cm) { this.cm = cm; }
    void draw() /* draw each region */ }
  }
  class Region {
    ColourMap<2> cm;
    void draw() { /* draw this region in relation to cm */ }
  }
  class ColourMap { ... }
}
```

Figure 8-16: Subject `Base` in the graphics suite

```
subject CopyPaste {
  class PictureLibrary {
    ColourMap<0> cm;
    Picture<0> p1, p2;
    Region<0> buffer;
    void copy() { buffer = p1.copy(); }
    void paste() { p2.paste(buffer); }
  }
  class Picture {
    Region<0,0,0>[][] r;
    Dims<0> markup;
    Region<1> copy() { /* create region based on markup */ }
    void paste(Region<1> buffer) { ... }
  }
  class Region {
    ColourMap<1> cm;
  }
  class ColourMap { ... }
  class Dims { /* specifies the dimensions of the area of interest */ }
}
```

Figure 8-17: Subject `CopyPaste` in the graphics suite

```
1    class PictureLibrary {
2      ColourMap<0> cm;
3      Picture<0> p1, p2;
4      Region<0,0> buffer;
5      void newPic() { p1 = new Picture<0>(cm); }
6    }
7    class Picture {
8      Region<0,0,<0,0>>[][] r;
9      ColourMap<1> cm;
10     Picture(ColourMap<1> cm) {
11       this.cm = cm;
12       /* for each region with indices i,j */
13         r[i][j].setCM(cm);
14     }
15   class Region<cm_owner> {
16     void setCM(ColourMap<cm_owner> cm) { ... }
17   }
```

Figure 8-18: Code fragment showing `Region` as an uncomposable class

To provide the required flexibility, class `Region` should be made uncomposable with the owner of the colour map passed to the region as an ownership parameter. Figure 8-18 shows a fragment of code where `Region` has been declared as an uncomposable class. The type of `PictureLibrary.buffer` in line 4 is `Region<0,0>` and of `r[i][j]` in line 13 is `Region<0,1>`. The ownership parameter `cm_owner` of uncomposable class `Region` binds differently in each case.

This example has shown that additional requirements can put a strain on the usability of a composable class, necessitating a change to an uncomposable class. The system of explicit context naming is more rigid (or less flexible) than the system of ownership parameterisation. Having made `Region` uncomposable it is no longer possible to extend this class by subject composition which tends to limit future adaptability. Earlier examples have shown that the rigidity of explicit contexts is also a strength of SAPS. In our experience rigidity is a compromise that works well in most cases.

8.7 Conclusion

Through a range of examples we have presented an evaluation of SAPS. The evaluation has assessed SAPS with respect to the factors that have motivated it. We have presented a range of subject-oriented development scenarios in which SAPS is an aid to subject design or reuse and demonstrated the limitations of our approach.

SAPS satisfies our reuse position by being of value to the subject developer and the reuser. We have shown that SAPS addresses the interaction problems which motivated it in Chapter 4 on page 44. SAPS is a useful tool in subject-oriented software development. This Chapter has demonstrated the following:

- Black-box components can be internally decomposed by feature while keeping the representation of the black-box hidden from its external clients.

- SOT can adapt to a variety of ownership structures when subjects are used to modularise cross-cutting concerns.

- Uncomposable classes may be used to hide implementation details from interception at join points.

- Unknown context identifiers support the *a priori* construction of reusable subjects.
- Deliberate use of explicit context identifiers can constrain composition to achieve the intended interaction.

Chapter 9

Conclusions and Future Work

This thesis was motivated by our interest in the factors affecting reuse. The conventional approach to reuse involves the construction of reusable software. The original developer needs to invest upfront in order to reap the benefits later. The initial investment can be recouped by marketing the software. Component frameworks support the customisation and integration of prebuilt components for the purpose of constructing new systems. However, most software built today is not intended for reuse but is constructed to meet some functional requirements. Reuse and evolution issues are secondary to functionality. Upfront investment in reusability will waste time that should be spent on meeting the deadline for the current iteration. Consequently, a question arose concerning how to build more reusable software in cases where future reusability is not in the initial requirements. We believe that to successfully tackle this issue, a technological solution must be of value to the original developers as well as the reuser. The original developer will be more interested in a reuse technology if that technology addresses certain problems during software construction.

In search of a solution, we looked at the way software is engineered. It is generally acknowledged that separation of concerns is an important reuse factor. Developers require technology that helps to separate all concerns that they believe to be important. For functional concerns, modularity is the key. Faced with a design problem, developers should be able to modularise the functional concerns. Traceability between the artifacts of importance in the requirements and code better supports the reuse, maintenance and evolution of those artifacts.

Object-oriented programming technology has failed to provide all the reuse benefits it was supposed to offer. Many functional concerns can be represented as object collaborations. In mainstream object-oriented programming languages object collaborations are not modular. Also, there exist aspects of systems – the concerns that cross-cut multiple object – which are scattered and tangled with the main functionality defined by abstractions. The scattering and the tangling makes both the abstractions and the aspects less reusable. Design patterns either cannot cleanly separate the concerns or the flexibility they provide becomes required after the program is written. Applying patterns during evolution is invasive, often requiring significant changes to program structure. Even concerns which are cleanly modularised by classes often cannot be extended in the way the reuser wants. Interfaces intended for defining the boundary between the client and abstraction implementation impede non-invasive evolution. In mainstream object-oriented languages there is a fuzzy line between subclassing for implementation reuse and subtyping for type substitutability. A subclass that is intended to be used in another application is required to conform to the interface(s) of its superclass(es) even when substitutability is not required.

9.1 The Subjective Alias Protection System

Of all the recent proposals for addressing separation of concerns we have argued that Subject-Oriented Programming (SOP) [49] best meets our reuse objectives. SOP introduces subjects as a new kind of module. Subjects are packages of classes and each subject defines a concern from its own perspective using a familiar object-oriented language. In many cases, subjects are suitable vehicles for modularising object collaborations and aspects. Subject interaction is specified in the composition specification using a special composition language. Unlike traditional paradigms, where interaction takes the form of procedure calls of one kind or another, subjects interact at join points. Join points are defined by language constructs such as classes and their members. Whereas functional interfaces are defined explicitly, join point interfaces 'simply exist'; there are no predefined extension or evolution points. One subject can be created as an extension to another and applied without changes to the extended subject. There is no substitutability defined between subjects, but internally each subject retains the benefits of inheritance for creating families of type substitutable abstractions.

9.1.1 An Understanding of Interaction Problems

In moving from classes to subjects as the main unit of reuse we encountered interaction problems. Understanding the interaction inevitably becomes more difficult as the number of subjects increases. Interaction problems are caused in part by the absence of composition interfaces. In our investigation we categorised interaction problems by their severity, that is, by effort required to overcome or eliminate the anomaly. In increasing order of magnitude they are:

- Change the composition specification.
- Extend the composition language with a new composition rule.
- Either modify the input subjects or create a patch subject.

The need for a powerful composition language was understood by SOP's creators from the start. The composition language is defined on top of an extensible framework that allows many rules to be specified. However, invasive changes to subjects or patching are a significant drawback to SOP as both a design and a reuse medium.

Independent development of modules is an important part of any paradigm. After a system is decomposed into subjects, it should be possible to assign each team the task of implementing each subject. In order for the composition of independently implemented or reused subjects to satisfy the requirements, the interface of subject interaction must be identified in advance. The problem is that the reuser must have a deep understanding of the subject. For many compositions, it is not enough to know what the subject does from the behaviour observed at the functional interfaces of its classes. It is necessary to know how those classes are implemented. The types of elements at join points provide little insight into the effects of subject interaction. Interaction problems requiring invasive changes or patching can be due to unanticipated state changes, e.g. when behaviour specified in one subject breaks an implicit condition in another subject, but the types of elements at join points provide little insight into the effects of subject interaction.

In an early attempt at a solution we considered introducing formal composition interfaces. These are attractive because they enable modular reasoning and delineate `public` join points from the `private` implementation. However, a formal composition interface is incompatible with our view on reuse: a subject should be reusable and extensible in ways not anticipated by its developers.

9.1.2 SAPS

The alternative that we propose, SAPS, improves the understandability of subject interaction without introducing formal composition interfaces. Like SOP, SAPS is not tied to any one language but is intended to be used in conjunction with today's mainstream programming technology. The SAPS proposal was inspired by a number of important observations:

- SOP cannot improve the basic design of abstract data types.
- SOP may be used to create new components.
- Object aliasing is a cross-cutting concern in SOP.

For SAPS we split classes into two hierarchies called composable and uncomposable. Uncomposable classes are common abstract data types and other container abstractions. Uncomposable classes have no internal join points. Method calls to their instances may be bracketed using SOP composition rules and they can be extended using conventional means but subject composition cannot be used to modify class definitions. The full range of composition rules may be applied on composable classes.

Composable and uncomposable classes are annotated using Subjective Ownership Types (SOT). These extend the familiar data types with ownership contexts. For every object in a subject-oriented program SOT define an owner. The owner forms a boundary. The object may have multiple references (aliases) inside the boundary, any of which may mutate its state, so long as no references are exposed outside the boundary. The ownership contexts themselves are objects and from the perspective of each object there are objects that it owns, known as the representation context. SOT ensure that objects never expose objects in their representation context.

The system is very flexible: an object is not required to reference objects in its representation context, yet it can also reference objects it does not own. Two totally different mechanisms make flexibility possible. For uncomposable classes we adapted the system used by Ownership Types [23]. At instantiation, an object is parameterised by the contexts it needs to reference. An ownership capability is passed in the form of an ownership parameter. Parameterisation is required for uncomposable classes because the subject developer may require two instances of the same container class with different ownership properties. For composable classes a totally new system of context identification was invented. The nesting between ownership contexts inspired explicit context identifiers or exps. Instead of passing ownership capabilities using parameters, each context is numbered in relation to the current representation context. exps and ownership parameters enforce very similar representation containment properties. The main difference is that ownership parameters grant permission and exps do not require permission to be granted.

Ownership parameters were unsuitable for composable classes because each subject may assign responsibilities to a class and subjects have partially overlapping views of abstractions. Thus, in each subject a class defines the ownership parameters it requires. In a subject-oriented program any subject may create objects resulting in other subjects' ownership parameters not binding. The system of exps avoids this problem by requiring only a single ownership parameter for composable classes – the object owner. This is bound no matter which subject performs the instantiation.

During requirements capture for SAPS we identified the need for a special kind of ownership parameterisation. Subjects may be used to describe concerns which apply in different contexts. The system of exps is too rigid to describe the variety. So, we introduced unknown context identifiers or unks for context identifiers which get bound by subject composition (*resolved* in our terminology). unks can refer to contexts that are external to the collaboration implemented by a subject or simply to those contexts which are part of another subject's design. To ensure correct resolution, unks are accompanied by resolution constraints or ucircs.

In order to compose subjects, the corresponding elements must define compatible types. Compatibility is based on type equivalence. In addition to subject-oriented composition requirements, SAPS requires that elements define equivalent context identifiers. In the case of exp and unk combination, resolutions are produced which must satisfy the resolution constraints. We have argued that, under certain composition rules, representation objects as defined in each input subject remain protected and the types in the output subject are well-formed.

9.1.3 Contributions

Together, Subjective Ownership Types and extensions to subject composition rules form the Subjective Alias Protection System. SAPS contributes to solving interaction problems, to making SOP a viable paradigm of software construction, and to improving opportunities for reuse.

SAPS for Addressing Interaction Problems

Subject interaction problems lie on the critical path that leads to SAPS. Therefore, it is important to explain how SAPS helps to solve interaction problems.

The success of SOP as a paradigm for software development and, in parallel, as a concern reuse technology depends on the developers' ability to implement subjects independently and reuse subjects off-the-shelf. Interaction problems in part stem from composers' inability to foresee all consequences of interaction on state. We have shown that for stateful (de)compositions the effect of subjects on state can be understood by studying the details of implementation but not from join point interfaces alone. We believe that the level of granularity for understanding subject interaction is too low, which makes independent development impractical and reuse of subjects uneconomical.

As an Alias Protection System, SAPS directly helps to solve interaction problems caused by unconstrained aliasing in SOP. Modular development commences after a mutually compatible ownership structure has been agreed by subject developers, otherwise the subjects may not be composable. We have shown that by having to agree on an ownership structure some interaction problems can be eliminated entirely. SAPS partially addresses the problem of granularity. It constrains object aliasing in subject-oriented programs, making it easier to understand the effect of subjects on scattered state. The Subjective Ownership Types at join points help to determine those objects that can directly affect state, although it remains necessary to study method implementations to understand state mutation in detail. The annotational properties of SAPS make it easier for the subject composers/reusers to understand the interaction and detect anomalies.

SAPS for Subject-Oriented Design

Subject composition is associated both with design and reuse. This dissertation has demonstrated the strengths and the limitations of SAPS as a tool for supporting subject-oriented software construction.

Uncomposable classes can be used to hide class implementations. Defining a class as uncomposable is equivalent to 'do not compose here'. Thus it is possible, where necessary, to restrict composition to particular classes. However, we believe that this should be done with care; uncomposable classes should not in general be used to define the composition interface, but, rather, to hide the implementations of those abstractions which should not be accessible.

By making common abstract data types uncomposable we acknowledge that open class development with SOP cannot improve the core design of these abstractions. However, it should be possible to apply aspects to abstract data types with SOP. For example, a client may require persistent `Queue` objects. SAPS allows instances of uncomposable classes to participate in open class compositions but never their classes. SAPS bracket relationship can create a persistent `Queue` object. Inheritance or delegation should be used if a persistent `Queue` class is required. To create new container abstractions with parameterisable ownership properties developers must use uncomposable classes. Whether this is a significant design impediment in practice remains to be seen.

Many examples of SOP demonstrate a single composition that ties together input subjects to produce the output application. No further compositions are considered. We have looked into the next dimension of composition, at the ownership properties associated with output subjects. By making ownership properties explicit it has become possible to communicate and enforce the ownership properties required for the output subject. In this way SAPS supports the construction of systems using black-box components created using SOP.

SAPS on Reuse

This thesis is motivated by software reuse. SOP provides the essential platform for reuse and most reuse benefits derive from using subjects as reuse artifacts. Compatibility with existing platforms makes SOP useful to reusers. SOP can synthesise new programs from existing software created without awareness of SOP. We believe that there are four ways in which SAPS supports or improves reuse opportunities:

- **Compatibility with existing practices.** With SAPS we have improved SOP while attempting to minimise the impact on software which exists already. The checking of Subjective Ownership Types is static both for input subjects and for compositions. For instance, the SAPS implementation for Java should run on a standard virtual machine. SOT are downwardly restrictive which means that SOT-annotated subjects (created either by programmer or by composition) place no constraints on the way the client code is implemented.

- **Encapsulation of representation.** It can be said that component reusability is improved if the developer is certain of its correctness. Representation encapsulation impacts correctness. SOP can be used to create components by mixing and matching features. SAPS extends the benefits of an Alias Protection System to SOP. It helps to ensure that all combinations of features keep representation objects hidden inside the component.

- **Correct subject deployment.** Compared to ownership parameterisation, the system of explicit context naming does a good job of conveying the ownership structure. This property is useful during subject composition. We have shown that, when alternative and seemingly valid compositions exist, by using `exps` we can ensure that subjects are composed correctly.

- **Unknown context identifiers.** Through `unks`, SAPS supports the realisation of concerns where precise ownership structure is determined by composition. For example, the Composite design pattern [43] should be reusable with a number of different ownership structures. `unks` support the development of subjects where reuse is in the requirements.

Finally, SAPS satisfies our reuse position by being of value to the original subject developers. Subjects are object-oriented and aliasing is a concern in object-oriented programming. Some investment is required by subject developers in order to apply Subjective Ownership Types to subjects. However, SOT are a fully-fledged Alias Protection System that helps the developer to create well structured subjects.

9.2 Future Work

There are three avenues of research that are relevant to the work in this thesis:

- Software reuse is and, we believe, will remain a research topic so long as software engineers pursue ways to drive down software development costs. The success of a reuse technology depends on many factors, one of which is the motivation for the original developer whose efforts may or may not be reused in the future. All proposals that seek to improve reuse should address this issue.

- Interaction problems in the presence of shared object state were presented and tackled in this thesis. However, interaction problems in Aspect-Oriented Software Development is still an open research issue.

- In this thesis we have not discussed the subject-oriented design process: how concerns that become subjects are identified and analysed. A process of subject-oriented design is an open research issue. Consequently, it is too early to assess the impact of SAPS on the analysis, design and testing stages.

The above are important issues in the long term. We dedicate this Section to topics which are of more immediate concern. We discuss implementation, formalisation, and extensions to SAPS in order to improve its concern modelling potential.

9.2.1 Implementation Issues

SOP concepts are realised within the programming language Hyper/J. In this language the subjects are implemented in Java. At the time of writing, not all functionality specified in the documentation [121] is implemented in the language. Also, the relationship between composition rules and access modifiers is not fully developed in either SOP or Hyper/J.

We have constructed a simple SOT compiler and a Subject Composer for a toy language based on Java. The development of a SOT compiler for all of Java and the integration of SAPS concepts into Hyper/J is future work. Although the major theoretical issues are specified in this dissertation, before a compiler for a production language can be specified, the relationship between SAPS and advanced language features should be considered, e.g. inner classes and exception handling.

9.2.2 Formalisation

The approach used in this work has been predominantly an informal one. Firstly, our aim was to improve opportunities for reuse and, secondly, to help solve interaction problems. We have not only made progress on both issues but also demonstrated SAPS as an invaluable tool for subject-oriented design. A specification of static and dynamic semantics of Subjective Ownership Types, and a proof of soundness is future work. The ownership concepts at subject level may be modelled using Clarke's extensions [22] to the imperative variant of Abadi and Cardelli's object calculi [1]. We believe that the containment properties of subjective ownership concepts can be shown to map on to the core model.

In [128], the authors define the semantics of MinAML, an idealised aspect-oriented programming language that distills the essence of AspectJ and the bracketing functionality of Hyper/J. The core aspect calculus on which it is based features explicitly labelled join points and a single piece of advice that applies at the label. The MinAML language is inspired more by AspectJ than Hyper/J in that it is based on an assymmetric model [50]. The base programs cannot manipulate advice in any significant way. In the core aspect language the labelled join points are defined independently of other constructs and hence can be reused in other computational settings with little change. Walker et al [128] show how constructs from Abadi and Cardelli's first order object calculus inter-operate with the aspect calculus. A possible avenue of investigation would be the inter-relation between subjective ownership concepts expressed using object calculi and the aspect calculus.

9.2.3 More Powerful Aliasing Systems

Dynamic Aliases

The original work on Ownership Types [23] that inspired Subjective Ownership Types lacked support for dynamic aliases. Dynamic aliases were added to Ownership Types in [21]. In Section 5.1.1 on page 72 we showed the utility of dynamic aliases for supporting object-oriented idioms such as iterators. Dynamic aliases are still required for uncomposable classes for the same reason as in object-oriented programs: to enable efficient access to data stored in containers. In subject design their other uses include the definition of friendly functions and initialisation of object representation with externally created objects to which there are no external aliases [32]. Friendly functions are permitted access into another object's private representation. A number of important issues remain outstanding at this point:

- How is a mode describing dynamic aliases useful to the subject composer? A dynamic mode describes an additional alias usage policy which may prove useful for constraining composition to ensure correct subject deployment.

- How to incorporate a dynamic model into SAPS? Clearly, a mode describing dynamic access may never appear in the type of an instance variable. But such a mode may appear in the types of operation signatures, local variables and expressions.

- Composing two elements when both have a dynamic mode yields an output element with the

```
subject Base {
  class Byte {
    Bit<0,0>[8] bit;
    void set(int index) {
      bit[index] = true;
    }
    int value() { /* return int value of byte */ }
    ...
  }
  class Bit {
    boolean value;
    void set() { value = true; }
  }
}
```

Figure 9-1: Subject `Base`

same mode. What is the meaning of composing an `exp` annotated element with an element whose owner is specified by a dynamic mode?

Co-ownership

During evaluation we identified a concern that would benefit from a different form of object ownership to that offered by SAPS: it was necessary to associate state with more than one representation context. The need for shared ownership of this kind has not been considered by APS researchers whose work is reviewed in this thesis. This is probably because in many traditional applications of object-oriented technology, the flexibility provided by single owner systems proves sufficient. However, advanced separation of concerns advocated by MDSOC and supported by SOP may benefit from a more powerful system.

We believe that there are two forms of co-ownership that would improve the modelling potential of SAPS:

- Systems where co-ownership is required by a fixed number of objects identified in advance.
- Systems where co-ownership is a concern that emerges during subject composition.

When discussing co-ownership of an object we presume that the owners are not ordered and that existing ownership structures are insufficient to express the required relationship. For example, consider the program in Figure 9-1. At present SAPS disallows a `Bit` to be owned by two or more different `Byte` objects.

But suppose that such an ownership structure was necessary. How would the owners be specified and how can we ensure that only the owners and other trusted object access the co-owned object? The problem is that in order to get into the representation of its owners, an object has to pass through an untrusted context. Thus we require some form of representation exposure.

Ways of constraining external references include uniqueness, dynamic aliases, read-only interfaces and reference-only access. We prefer to steer clear of uniqueness for it requires either programming language support for linear types [89] or the programmer to adopt an unconventional programming style [5]. Read-only references are upwardly restrictive and operations which are read-only may become read-write operations after composition. Dynamic aliases still allow objects other than the owners to change object state.

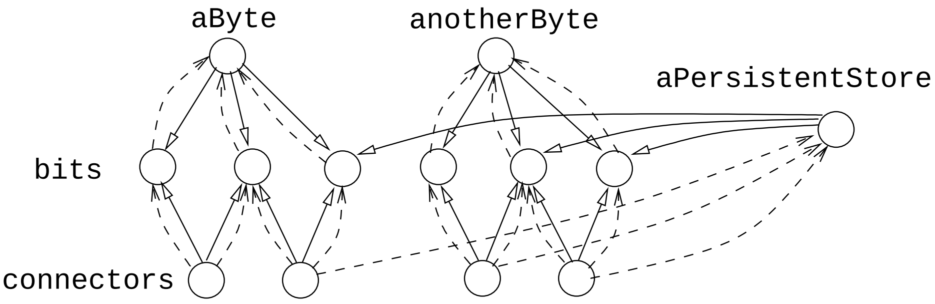

Figure 9-2: Co-ownership Tree

We shall discuss two candidate schemes for co-ownership. Co-ownership with a fixed number of pre-specified owners is represented by a sequence of context identifiers that follow the class name. Co-ownership as an emergent concern requires a form of context polymorphism that is not easily represented using either `unks` or ownership parameters. Figure 9-2 shows an object graph and ownership relations for two concerns that would benefit from these schemes. Solid edges indicate inter-object references and dashed edges relate objects to their owners. `Connector` objects associate exactly two `Bits` that also own the `Connector`. Only the owner `Bits` (and objects inside) should be allowed access to the shared `Connector`. A `PersistentStore` object saves the state of `Bits` belonging to potentially different `Bytes`. The `PersistentStore` object should be allowed access in addition to the explicitly specified `Bit` owner.

Co-ownership: Fixed Owner List

For a fixed number of pre-identified owners the code in Figure 9-3 shows a subject with annotations we propose. It is composed with subject `Base` specified in Figure 9-1. This example shows the component integration concern from Section 8.3 on page 195 where the associations are co-owned by the pertinent `Bits`. An array of `Connector` objects is declared in line 4 and initialised in lines 6–7. The array is owned by the encapsulating `Byte`. The `Connectors` are co-owned by the `Bits` they connect. We use $ instead of angle brackets to indicate a family of owners.

Explicit contexts and variable identifiers in scope can be specified as owners. The co-owned object can be passed based on existing SAPS rules so long as all co-owners are in scope. That is, each owner can be referred to using either an `exp` or a variable. When the object type contains only variable names, a e.g. line 4 in Figure 9-3, the client may only initialise and pass the reference; it is not allowed to access the object's inteface. This system ensures, first, that external clients have no state dependencies on objects exposed outside their owners and, secondly, that state changes to exposed objects are avoided. Variables in the type are indicative of external owners that cannot be specified with an `exp`. In fact, the type of a co-owned object may contain at most one `exp`, otherwise there exists a redundant co-owner in the definition.

The following code fragment shows a reference being passed to a `U` instance. Object `myT` is co-owned by the current representation context and by `myV`. `myT`'s reference can be passed to `myU` only if its co-owners remain accessible either using `exps` or using a variable name:

```
U<O> myU;
V<1> myV;
T$myV, O$ myT;
```

```
subject Equality {
  class Byte {
    Bit<0,0>[8] bit;
    Connector<0, $bit[i],bit[i+1]$>[7] c;
    void makeEquality() {
      for(int i = 0; i < 7; i++)
        c[i] = new Connector$bit[i],bit[i+1]$(bit[i],bit[i+1]);
    }
  }
  abstract class Bit {
    abstract void set();
  }

  class Connector {
    boolean busy;
    Bit<2> left, right;
    Connector(Bit<2> left, Bit<2> right) {
      this.left = left;
      this.right = right;
    }
    void after_set(Bit<2> target) {
      if(target == left) {
        if(!busy) {
          busy = true;
          right.set();
          busy = false;
        }
      }
      // same for right
    }
  }
}

// composition specification
compose Base, Equality into BE;
mergeByName;
bracket ‘‘Bit’’.‘‘set’’ with after Equality.Connector<$Receiver>.after_set($Receiver);
```

Figure 9-3: Subject Equality and composition specification for integration with subject Base

```
myU.setMyT(myT, myV);

class U {
  void setMyT(T$someV, 1$ someT, V<2> someV) { ... }
}
```

The composition specification connecting subjects `Base` and `Equality` (lines 35–37 in Figure 9-3) is conventional except for the last statement. The normal effect of the bracket relationship on execute sites is to introduce the body of class `Connector` into class `Bit`, so the state of a single `Connector` object is associated with a single `Bit` object. However, this is not the effect we require. We need to associate the state of two `Bits` with the right `Connector`. This is indicated in the composition specification by `Connector<$Receiver>` where `$Receiver` is a meta-parameter binding to the identity of the receiver object matched by the pattern. This notation means exactly "the `Connector` with a co-owner given by `$Receiver`". Finally, notation `after_set($Receiver)` passes the receiver object to the wrapper method as the sole argument.

To implement this system, changes are required both to the SOT compiler and the Subject Composer. The dynamic association between two `Bit` objects to one `Connector` object should be hidden from the composition author by the implementation of the Subject Composer. In related work, Sakurai et al [109] showed the way stateful aspects like those implemented by the `Equality` subject can be implemented in AspectJ. The authors define `Association` aspects which associate the state of one aspect instance with a number of objects, selected dynamically using AspectJ's join point mechanism. While our proposal predefines the co-owners, it also aims to enforce object containment in a multi-owner environment.

Co-ownership: Emergent Owners

When the set of co-owners is not known in advance, it should be possible to parameterise an object by its other owners. SAPS has two forms of parameterisation: ownership parameters are bound during object instantiation and unks are resolved by subject composition. In addition we propose a form of context polymorphism that we will call ω-contexts. The purpose of ω-contexts is to refer to multiple owners dynamically.

To motivate ω-contexts consider the Persistence concern. The Persistence concern is difficult to implement with SAPS presently because state is associated with just one owner. The `PersistentStore` is a separate object with a separate representation context. In order to save the objects in the representation contexts of a `Byte` object, it is necessary to expose the `Bit` objects from the representation of `Byte` and pass them to the `PersistentStore` object. The above system of pre-specified co-ownership is not suitable here: `Bits` from many `Bytes` may be associated with the `PersistentStore`. The `PersistentStore` needs to know about the elements it stores but not about `Bytes` or about the way `Byte` objects are organised. Thus co-ownership is a cross-cutting concern that emerges when subjects `Base` and `Persistence` are composed.

Figure 9-4 shows the Persistence subject annotated with ω-contexts. The $-punctuated identifier in line 2 refers to the ω-context bound dynamically during inter-subject interaction. According to the composition specification (line 15), the world-owned PersistentStore object's `save_set(..)` method is called with the object of the bracketed method as argument. The exp_0 in `save_set(..)` shows that the PersistentStore co-owns the Bit parameter with the object bound to `bit_owner`. The `bit_owner` ω-context binds to multiple `Byte` objects, thus allowing one `PersistentStore` object to store `Bits` from the representation of a limitless number of `Bytes`.

```
subject Persistence {
  class PersistentStore with $bit_owner$ {
    Hashtable<0,$0,bit_owner$,world> h;
    void save_set(Bit$0,bit_owner$ b) { h.put(b,true); }
    boolean retrieve(Bit$0,bit_owner$ b) { h.get(b); }
    ...
  }
  class Bit { }
}

// composition specification
compose Base, Equality, Persistence into BP;
mergeByName;
bracket ‘‘Bit’’.‘‘set’’ with after Equality.Connector<$Receiver>.after_set($Receiver);
bracket ‘‘Bit’’.‘‘set’’ with after Persistence.PersistentStore<world>.save_set($Receiver);
```

Figure 9-4: Persistence subject demonstrating co-ownership as a concern that emerges during composition

In conclusion, in some cases, in order to support separation of concerns we require a more advanced alias protection system than SAPS. The two schemes that we have presented allow multiple objects to co-own an object while still protecting it from external access. We believe that the schemes we propose are feasible but further work is required in order to fully develop the two forms of co-ownership and to assess their impact on Subject-Oriented Programming.

9.2.4 Support for Layered Designs

In Chapter 5 on page 70 we discussed the construction of new components by subject composition. Chapter 8 on page 183 has shown that SAPS helps to ensure that the representation of such components stays hidden behind the functional interface. One extension concerns the use of components constructed by subject composition with SAPS-aware clients.

Imagine a Spreadsheet component which has been created by subject composition. Spreadsheet is created without reference to any one particular application. Consequently, contexts other than exp_1 and world in the functional interface are not meaningful. Furthermore, as proposed in Chapter 5, the client should be able to parameterise Spreadsheet instances with respect to their ownership properties.

To meet the demand for new components which are constructed by subject composition, we propose an extension for transforming the output subject into an uncomposable class with ownership parameters. The uncomposable class can then be reused as a black-box in the design of larger-grained subject-oriented programs. The transformation is a mapping from a subject with unresolved unks to a class or classes where unks become ownership parameters.

Up to now we required that composition resolves all unks. In order to have ownership parameters some unks should not be resolved by composition. Subjects contain class and subject-level ucircs. Class level ucircs for the unresolved unks remain. Subject-level ucircs where only one of two unks resolves become class-level ucircs that get appended to each class. Subject-level ucircs where neither unk resolves are put into the uncomposable classes' where clauses. For each class nominated as a component interface, the resolution set should not be constrained although it may exclude the representation context exp_0.

To illustrate the transformation we use a small example. The Queue concern may be decomposed

```
subject Put {
  unk k;
  class Queue {
    Link<O> head = null;
    Link<O> tail = null;

    void put(Object<k> o) {
      Link<O> l = new Link<O>(o);
      if(head == null) {
        head = tail = l;
      } else {
        tail.next = l;
        tail = l;
      }
    }
  }

  class Link {
    Object<k> o;
    Link<1> next;
    Link(Object<k> o) { this.o = o; }
  }
}
```

Figure 9-5: Subject `Put` implementing the Put feature in the Queue concern

into two 'features' for putting elements into the `Queue` and getting elements out of the `Queue`[1]. We may use SOP to implement each feature as a subject: subjects `Put` and `Get` are shown in Figures 9-5 and 9-6 respectively.

The subjects are merged in the usual way but $\texttt{unk}_k$, denoting the data owner, is unresolved. Nominating class `Queue` as the interface, $\texttt{unk}_k$ is turned into an ownership parameter, creating an uncomposable class with the following externally accessible interface:

```
class Queue<k> {
  Object<k> get() { ... }
  void put(Object<k> o) { ... }
}
```

This example shows that the proposed extension helps to hide feature concerns of a component while, at the same time, making the component reusable within a variety of contexts. With this extension SAPS will be better suited to support software development where components are organised in layers. To realise this extension, further work is required in the area of unknown context resolution.

9.3 A Final Word

The Subjective Alias Protection System developed in this dissertation has brought together subjectivity and ownership in response to our perspective on reuse. We believe that to construct reusable software when reuse is not in the requirements demands technology that is of value to the original developer. Our approach is characterised by feature-based decomposition, using subjects to modularise concerns identified in the requirements. Where state was involved, anomalies in subject

[1] Note that we do not advocate that `Queue` be implemented this way.

```
subject Get {
  unk k;
  class Queue {
    Link<0> head = null;
    Link<0> tail = null;

    Object<k> get() {
      if(head == null) return null;
      Object<k> o = head.o;
      if(head == tail) {
        head = tail = null;
      } else {
        head = head.next;
      }
      return o;
    }
  }

  class Link {
    Object<k> o;
    Link<1> next;
  }
}
```

Figure 9-6: Subject `Get` implementing the Get feature in the Queue concern

interaction were deemed to make independent development unlikely and subject reuse improbable. The ownership concepts we introduce into subject-oriented development raise the level of abstraction, improving the developers' and reusers' ability to understand subject interaction, while at the same time adding value to the subject creator for whom future reuse is generally not a prominent concern.

Recently, Jacobson [60] wrote about the important difference that Aspect-Oriented Software Development will make to the way software is constructed. He drew the link between Use Cases in UML and aspects (in the general sense that includes subjects). The design process that Jacobson envisages for the future involves the following stages:

1. Find and specify the use case to describe the system requirements.

2. Design and code each use case.

3. Compose the use case slices (e.g. code in the form of subjects implementing each use case).

4. Test each use case.

For the third step, Jacobson writes "I expect that this activity will be reduced through tooling and through collaboration between the concerned use case designers". We believe that SAPS has made a contribution in this respect.

Bibliography

[1] M. Abadi and L. Cardelli. *A Theory of Objects.* Monographs in Computer Science. Springler-Verlag, 1996.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers. Priciples, Techniques, and Tools.* Addison-Wesley series in Computer Science. Addison-Wesley Publishing Company, 1986.

[3] M. Aksit, L. Bergmans, and S. Vural. An object-oriented language-database integration model: The composition-filters approach. In Ole Lehrmann Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, volume 615, pages 372–395, Berlin, Heidelberg, New York, Tokyo, 1992. Springer-Verlag.

[4] M. Aksit, B. Tekinerdogan, and L. Bergmans. The six concerns for separation of concerns. In *ECOOP 2001 Workshop on Advanced Separation of Concerns*, Budapest, 2001.

[5] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, November 2002.

[6] P. S. Almeida. Balloon types: Controlling sharing of state in data types. *Lecture Notes in Computer Science*, 1241:32–59, 1997.

[7] A. Batenin. Integration of independently-developed object-oriented designs. In *OOPSLA'01 Companion.* ACM, October 2001.

[8] D. Batory, J. Liu, and J. N. Sarvela. Refinements and multi-dimensional separation of concerns. In *ESEC/FSE'03*, 2003.

[9] D. Batory, J.N. Sarvela, and A.Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.

[10] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin. The GenVoca model of software-system generation. *IEEE Software*, 11(5):89–94, September 1994.

[11] K. Beck. Embracing change with Extreme Programming. *Computer*, 32:70–77, October 1999.

[12] K. Beck and W. Cunningham. A laboratory for teaching object oriented thinking. *ACM SIGPLAN Notices*, 24(10):1–6, October 1989.

[13] R. Ben-Natan. *CORBA: A guide to CORBA.* McGraw-Hill, 1995.

[14] R. Biddle, A. Martin, and J. Noble. No name: just notes on software reuse. *ACM SIGPLAN Notices*, 38(12):76–96, December 2003.

[15] R. Biddle and E. Tempero. Explaining inheritance: A code reusability perspective. *SIGCSEB: SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education)*, 28, 1996.

[16] T. Biggerstaff and C. Richter. Reusability framework, assessment, and directions. *IEEE Software*, 4(2):41–59, March 1987.

[17] B. W. Boehm, J. R. Brown, H. Kasper, M. Lipow, G. Macleod, and R. Merritt. *Characterstics of Software Quality*. TRW Series on Software Technology, 1978.

[18] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[19] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. *ACM SIGPLAN Notices*, 38(1):213–223, January 2003.

[20] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A system of patterns - pattern oriented software architecture*. Wiley, 1996.

[21] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, November 2002.

[22] D. Clarke, J. Noble, and J. M. Potter. Simple ownership types for object containment. *Lecture Notes in Computer Science*, 2072:53–76, 2001.

[23] D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. *ACM SIGPLAN Notices*, 33(10):48–64, October 1998.

[24] S. Clarke. *Composition of Object-Oriented Software Design Models*. PhD thesis, School of Computer Applications, Dublin City University, 2001.

[25] S. Clarke, W. Harrison, H. Ossher, and P. Tarr. Subject-oriented design: towards improved alignment of requirements, design, and code. *ACM SIGPLAN Notices*, 34(10):325–339, October 1999.

[26] S. Clarke and R. J. Walker. Composition patterns: an approach to designing reusable aspects. In *Proceedings of the 23rd international conference on Software engineering*, pages 5–14. IEEE Computer Society, 2001.

[27] S. Clarke and R. J. Walker. Separating crosscutting concerns across the lifecycle: From composition patterns to AspectJ and Hyper/J. Technical Report TR-2001-05, Department of Computer Science, University of British Columbia, August 08 2001. Wed, 08 Aug 2001 21:00:41 GMT.

[28] C. Clifton and G. T. Leavens. Spectators and Assistants: Enabling modular aspect-oriented reasoning. Technical Report 02-10, Iowa State University, Department of Computer Science, 2002.

[29] C. Constantinides and T. Skotiniotis. Providing multidimensional decomposition in object-oriented analysis and design. In *IASTED International Conference on Sofware Engineering (ICSE 2004)*, Innsbruck, Austria, 2004.

[30] M. Day, R. Gruber, B. Liskov, and A. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. *ACM SIGPLAN Notices*, 30(10):156–168, October 1995.

[31] A. DeSoto. *Using the Beans Development Kit 1.0.* JavaSoft, Sun Microsystems Inc., September 1997.

[32] D. L. Detlef, K. R. M. Leino, and G. Nelson. Wrestling with rep exposure. Technical report, Compaq Systems Research Center, 1998.

[33] R. Douence, P. Fradet, and M. Sudholt. Composition, reuse and interaction analysis of stateful aspects. In *Conference on Aspect-Oriented Software Development*, pages 141–150, 2004.

[34] E. Ernst. Family polymorphism. In *ECOOP'01 – Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 303–326. Springer Verlag, 2001.

[35] M. Fayad and D. C. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, October 1997.

[36] M. E. Fayad. Introduction to the Computing Surveys' electronic symposium on object-oriented application frameworks. *ACM Computing Surveys*, 32(1), March 2000. Article No. 4.

[37] A. J. Field and P. G. Harrison. *Functional Programming.* Addison-Wesley, Wokingham, 1988.

[38] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. Technical report, January 2000.

[39] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: A framework for integrating multiple perspectives in systems development. *International Journal of Software Engineering and Knowledge Engineering*, 1(2):31–58, 1992.

[40] G. Fischer. Cognitive view of reuse and redesign. *IEEE Software*, 4(4):60–72, July 1987.

[41] B. Foote and J. W. Yoder. Big Ball of Mud. In *Proceedings of PLop '97*, 1997. Fourth Conference on Patterns Languages of Programs (PLoP '97/EuroPLoP '97).

[42] Martin Fowler. *Extreme Programming Examined*, pages 3–18. Addison-Wesley, 2001.

[43] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1994. ISBN 0-201-63361-2.

[44] T. Gardner. *Inheritance Relationships for Disciplined Software Construction.* PhD thesis, Univerity of Bath, 1999.

[45] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison Wesley, 1997.

[46] A. Greenhouse and J. Boyland. An object-oriented effects system. *Lecture Notes in Computer Science*, 1628:205–229, 1999.

[47] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. *ACM SIGPLAN Notices*, 37(11):161–173, 2002.

[48] E. R. Harold. *JavaBeans.* I D G Books Worldwide, Indianapolis, IN, USA, 1998.

[49] W. Harrison and H. Ossher. Subject-oriented programming (A critique of pure objects). In *Proceedings of the OOPSLA '93 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 411–428. IEEE Computer Society, Los Alamitos, CA, USA, October 1993.

[50] William H. Harrison, Harold L. Ossher, and Peri L. Tarr. Asymmetrically vs. symmetrically organized paradigms for software composition. Technical report, IBM Research, 2002.

[51] Y. Hassoun and C. A. Constantinides. The development of generic definitions of hyperslice packages in Hyper/J. In Uwe Assmann, Elke Pulvermueller, Isabelle Borne, Noury Bouraqadi, and Pierre Cointe, editors, *Electronic Notes in Theoretical Computer Science*, volume 82. Elsevier, 2003.

[52] J. D. Hay and J. M. Atlee. Composing features and resolving interactions. In David S. Rosenblum, editor, *Proceedings of the ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering (FSE-00)*, volume 25, 6 of *ACM Software Engineering Notes*, pages 110–119, NY, November 8–10 2000. ACM Press.

[53] R. Helm, I. Holland, and D. Ganghopadhyay. Contracts: Specifying behavioural compositions in object-oriented systems. In *OOPSLA'90*, pages 169–180, 1990.

[54] S. Herrmann. Object teams: Improving modularity for crosscutting collaborations. *Objects, Components, Architectures, Services, and Applications for a Networked World, LNCS 2591*, 2002.

[55] S. Herrmann. Confinement and representation encapsulation in object teams. Technical Report ISSN 14369915, University of Berlin, June 2004.

[56] J. Hogg. Islands: Aliasing protection in object-oriented languages. In Andreas Paepcke, editor, *OOPSLA '91 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*, pages 271–285. ACM Press, 1991.

[57] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The Geneva Convention on the treatment of object aliasing. *OOPS Messenger*, 3(2):11–16, April 1992.

[58] I. M. Holland. Specifying reusable components using contracts. In *ECOOP'92 LNCS 615*, pages 287–308, 1992.

[59] U. Hölzle. Integrating independently-developed components in object-oriented languages. *Lecture Notes in Computer Science*, 707:36–56, 1993.

[60] I. Jacobson. The case for aspects. *Software Development Magazine*, October–November 2003.

[61] I. Jacobson, M. Christerson, P. Jonsson, and G. G. Overgaard. *Object-Oriented Software Engineering*. Addison-Wesley, 1992.

[62] S. Katz. Diagnosing harmful aspects using regression verification. In *Foundations of Aspect Languages Workshop at AOSD'04*, 2004.

[63] S. E. Keene. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison Wesley Professional, 1989.

[64] G. Kiczales. Beyond the black box: open implementation — Soapbox. *IEEE Software*, 13(1), January 1996.

[65] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Meta-Object Protocol.* MIT Press, Cambridge (MA), USA, 1991.

[66] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[67] K. Kimbler. Comprehensive approach to service interaction handling. *Computer Networks and ISDN Systems*, 1998.

[68] K. Kimbler and H. Velthuijsen. Feature interaction benchmark. In *Discussion paper for the panel on Benchmarking at Feature Interaction Workshop*, 1995.

[69] H. Klaeren, E. Pulvermüller, A. Rashid, and A. Speck. Aspect composition applying the design by contract principle. *Lecture Notes in Computer Science*, 2177:57–69, 2001.

[70] C. W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

[71] A. Lai and G. C. Murphy. The structure of features in java code: An exploratory investigation. In *OOPSLA Companion'99*, 1999.

[72] W. LaLonde and J. Pugh. Subclassing $\neq$ subtyping $\neq$ is-a. *Journal of Object-Oriented Programming*, pages 57–60, January 1991.

[73] K. J. Lieberherr, I. Silva-Lepe, and C. Xaio. Adaptive object-oriented programming using graph-based customizations. *Communications of the ACM*, 37(5):94–101, May 1994.

[74] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[75] C. Lopes and W. Hursch. Separation of concerns. Technical report, College of Computer Science, Northeastern University, February 1995.

[76] C. V. Lopes. *Aspect-Oriented Software Development*, chapter AOP: An Historical Perspective. Addison-Wesley, 2004.

[77] C. V. Lopes, P. Dourish, D. H. Lorenz, and K. Lieberherr. Beyond AOP: toward naturalistic programming. *SIGPLAN Notices*, 38(12):34–43, 2003.

[78] C. V. Lopes and G. Kiczales. D: A language framework for distributed programming. Technical Report SPL97-010, P9710047, Xerox Palo Alto Research Center, Palo Alto , CA , USA, February 1997.

[79] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–57. ACM Press, 1988.

[80] A. Lynex and P. J. Layzell. Organisational considerations for software reuse. *Annals of Software Engineering*, 5:105–124, 1998.

[81] J. D. McGregor and T. Corson. Supporting dimensions of classification in object-oriented design. *Journal of Object-Oriented Programming*, 5(9):25–30, 1993.

[82] M. D. McIlroy. Mass produced software components. In P. Naur and B. Randell, editors, *Report on a conference by the NATO Science Committee*, pages 138–150. NATO Scientific Affairs Division, 1968.

[83] B. Meyer. Genericity versus inheritance. In *OOPSLA '86*, pages 391–405, September 1986.

[84] B. Meyer. *Eiffel: The Language.* Prentice Hall, 1991.

[85] B. Meyer. Applying design by contract. *IEEE Computer*, 1992.

[86] B. Meyer. *Object-Oriented Software Construction, 2nd Ed.* Prentice-Hall, Englewood Cliffs, NJ 07632, USA, second edition, 1997.

[87] B. Meyer. Component and object technology: On to components. *Computer*, 32(1):139–140, January 1999.

[88] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *International Conference on Aspect-Oriented Software Development (AOSD '03)*, Boston, USA, 2003.

[89] N. H. Minsky. Towards alias-free pointers. *Lecture Notes in Computer Science*, 1098:189–209, 1996.

[90] Tzilla Elrad (moderator) Mehmet Aksit Gregor Kiczales Karl Lieberherr Harold Ossher (panelists). Discussing aspects of AOP. *Communications of the ACM*, 44(10):33–38, October 2001.

[91] P. Müller and A. Poetzsch-Heffter. Universes: A type system for conrolling representation exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming.* Fernuniversität Hagen, 1999.

[92] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. *Lecture Notes in Computer Science*, 1445:158–185, 1998.

[93] W. F. Opdyke. *Refactoring Object-Oriented Frameworks.* Ph.D. thesis, University of Illinois, 1992.

[94] H. Ossher, M. Kaplan, W. Harrison, and A. Katz. Subject-oriented composition rules. *ACM SIGPLAN Notices*, 30(10):235–250, October 1995.

[95] H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal. Specifying subject-oriented composition. *Theory and Practice of Object Systems*, 2(3):179–202, 1996.

[96] J. Palsberg and M. I. Schwartzbach. *Object-Oriented Type Systems.* John Wiley & Sons, 1994.

[97] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM 15(12)*, 1972.

[98] R. Pawlak. *Interactional Aspect Oriented Programming to Construct Multiple Concerns Applications.* PhD thesis, CEDRIC Computer Science Laboratory of CNAM, Paris, France, 2002.

[99] S. L. Pfleeger. *Software Engineering: Theory and Practice.* Prentice-Hall, Upper Saddle River, NJ, 1998.

[100] M. Plath and M. Ryan. *Feature Interactions in Telecommunications and Software Systems*, volume V, chapter Plug-and-Play Features, pages 150–164. IOS Press, 1998.

[101] Harry H. Porter. Separating the subtype hierarchy from the inheritance of implementation. *Journal of Object-Oriented Programming*, 4(9):20–21,24–29, February 1992.

[102] J. Potter, J. Noble, and D. Clarke. The ins and outs of objects. In *Australian Software Engineering Conference*, Adelaide, Australia, November 1998. IEEE Press.

[103] J. S. Poulin. *Measuring Software Reuse: principles practices, and economic models.* Addison-Wesley Longman Inc., 1997.

[104] C. V. Ramamoorthy, V. Garg, and A. Prakash. Support for reusability in Genesis. *IEEE Transactions on Software Engineering*, 14(8):1145–1154, August 1988. Special Section on COMPSAC '86.

[105] T. Ravichandran and M. A. Rothenberger. Software reuse strategies and component markets. *Communications of the ACM*, 46(8):109–114, August 2003.

[106] J. C. Reynolds. Syntactic control of interference. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 39–46, January 1978.

[107] R. Rivest, A. Shamir, and L. Aldeman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(1), 1978.

[108] D. Rogerson. *Inside COM: Microsoft's Component Object Model.* Microsoft Press, 1997.

[109] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya. Association aspects. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 16–25. ACM Press, 2004.

[110] D. C. Schmidt. Why software reuse has failed and how to make it work for you. *C++ Report*, January 1999.

[111] I. Smaragdakis. *Implementing large-scale object-oriented components.* PhD thesis, University of Texas at Austin, 1999.

[112] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In Eric Jul, editor, *ECOOP'98 – Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 550–570. Springer, 1998.

[113] M. Sparling. Lessons learned: through six years of component-based development. *Communications of the ACM*, 43(10):47–53, October 2000.

[114] R. Van Der Straeten and J. Brichau. Features and feature interactions in software engineering using logic. *ECOOP'01 Workshop Reader*, 2001.

[115] B. Stroustrup. The C++ programming language (2nd edition). *Addison Wesley, ISBN 0-201-53992-6*, June 1991.

[116] K. Sullivan, L. Gu, and Y. Cai. Non-modularity in aspect-oriented languages: Integration as a crosscutting concern for aspectj. In *AOSD 2002 Conference Proceedings*, pages 19–27, 2002.

[117] K. J. Sullivan. *Easing the Design and Evolution of Integrated Systems.* PhD thesis, University of Washington, 1994.

[118] K. J. Sullivan and D. Notkin. Reconciling environment integration and software evolution. *ACM Transactions on Software Engineering and Methodology 1*, pages 229–268, 1992.

[119] Sun Microsystems. *JavaBeans Specification 1.0*, July 1997.

[120] S. Sutton and I. Rouvellou. Modeling of software concerns in Cosmos. In Gregor Kiczales, editor, *AOSD2002*, pages 127–133. ACM, 2002.

[121] P. Tarr and H Ossher. Hyper/J user and installation manual. Available from http://www.research.ibm.com/hyperspace, 2000.

[122] P. Tarr, H. L. Ossher, W. H. Harrison, and S. M. Sutton, Jr. N degrees of separation: Multi-dimentional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, May 1999.

[123] B. Tekinerdogan, L. Bergmans, M. Glandrup, and M. Aksit. On composing separated concerns: Composability composition anomalies. October 2000.

[124] L. Tokuda and D. Batory. Evolving object-oriented designs with refactorings. In *14th IEEE International Conference on Automated Software Engineering*, pages 174–182. IEEE Computer Society Press, 1999.

[125] M. VanHilst and D. Notkin. Using C++ templates to implement role-based designs. *Lecture Notes in Computer Science*, 1049:22–37, 1996.

[126] M. VanHilst and D. Notkin. Using role components to implement collaboration-based designs. *ACM SIGPLAN Notices*, 31(10):359–369, October 1996.

[127] J. Vitek and B. Bokowski. Confined types. *ACM SIGPLAN Notices*, 34(10):82–96, October 1999.

[128] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '03)*, ACM SIGPLAN Notices. ACM, August 2003.

[129] R. J. Walker. Eliminating cycles in composed class hierarchies. Technical Report TR-2000-07, Department of Computer Science, University of British Columbia, July 2000.

[130] M. J. Wooldridge. *An Introduction to MultiAgent Systems.* Chichester: Wiley, 2002.