First-Class Relationships for Object-Orientated Programming Languages

by

Stephen Nelson

A proposal
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Doctor of Philosophy
in Computer Science.

Victoria University of Wellington
2008

Abstract

The object-oriented paradigm describes a powerful system where complex systems can be modelled by a network of communicating objects. The object model is powerful because it appeals to our understanding of the world, yet it has serious shortcomings where the model doesn't match our expectations. In particular, relationships are completely ignored by most object-oriented programming languages. Programmers must rely on error prone code using references which are a poor substitute for many of the complex relationships programs must represent.

There have been several attempts to add relationships to the object-oriented paradigm and object-oriented design and object-oriented design and specification languages now include relationships, however there has been little progress made in adding relationships at the implementation level. We believe this is because existing attempts have not refined the object-oriented model, but simply added relationships to existing systems.

In this proposal we present a prototype for a novel model of relationships in the object-oriented paradigm in which relationships, rather than objects, are the dominant constructs. This shift in focus allows programmers to gain more leverage from the object-oriented abstraction by bringing the language closer to the systems which are modelled. In addition, our model produces a stronger hierarchy than existing object-oriented systems which makes program decomposition easier and reduces coupling between objects.

Acknowledgments

Thanks to my supervisors James Noble and David Pearce for their invaluable suggestions and support throughout the last few years.

Thanks also to Melanie Kissell (soon to be Nelson) for her support and tolerance.

Contents

Chapter 1

Introduction

The object-oriented paradigm describes a collection of objects which interact by sending messages between each other. Each object is composed of some state (data) and some behaviour (methods/functions). When objects communicate, one will initiate the 'conversation' by passing a message to another. The object which initiates conversation needs to know how to send a message to another object; it needs an address to send it to. This means that objects need to know about other objects. An object 'knowing' about another object creates a relationship (or association) between the objects. These relationships are usually represented explicitly in object-oriented design languages[17], however in most object-oriented implementation languages the relationships are implicitly defined.

Throughout this proposal we will refer to a simple example of a relationship presented in Figure 1.1 to illustrate relationship-related developments in object-oriented systems. The figure describes a relationship in an object-oriented system: the association labelled *Attends* between students and courses means that "*students can attend courses*". Figure 1.2 shows part of a typical implementation of this system in the Java programming language. This example is used extensively in the literature to demonstrate object relationships [4, 20, 2].

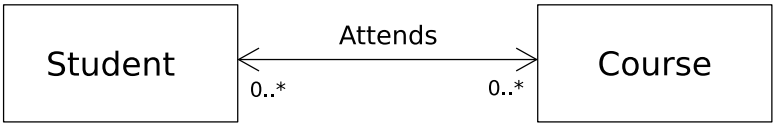


Figure 1.1: This UML (Unified Modelling Language [17]) diagram describes a simple system consisting of three parts: a *Student* class, a *Course* class, and a relationship between them called *Attends* represented by a line connecting the two classes. The *Attends* relationship is intended to mean that "*a student can attend courses*". A typical implementation of this in a traditional object-oriented language is presented in Figure 1.2.

```
class Student {
    private Set<Course> attended;
    public void addAttended(Course c) {...}
    public void removeAttended(Course c) {...}
    public Set<Course> getAttended() {...}
}
class Course {
    private Set<Student> attendees;
    public void addAttendee(Student c) {...}
    public void removeAttendee(Student c) {...}
    public Set<Student> getAttendees() {...}
}
```

Figure 1.2: This Java code demonstrates part of the traditional object-oriented implementation of the *attends* relationship described in Figure 1.1. The methods for maintaining the relationship (add, remove, get, etc) have been omitted for brevity.

1.1 The Problem

Armstrong identifies eight "*Quarks*", or fundamental building blocks of object-oriented development: *Abstraction, Class, Encapsulation, Inheritance, Object, Message Passing, Method* and *Polymorphism* [1]. These eight *Quarks* are the eight concepts most commonly identified in object-oriented development literature between 1966 and 2005. Notably absent from this list is *Relationship*, which takes a poor 13th position on a list of object-oriented development concepts. In fact, only 14% of the papers surveyed mentioned relationships, whereas each of the top eight *Quarks* received mention in at least 50% of the papers surveyed.

This is perhaps because relationships are second class citizens in object-oriented development. Few languages provide support for explicit relationship, yet Cunningham and Beck claim that relationships are one of the distinguishing features of object-oriented design [3] and relationships have been an essential part of the Unified Modelling Language (the dominant modelling language for the object-oriented methodology) since its infancy [24].

This disconnect between design and implementation has been identified by several groups. Pearce and Noble echo Cunningham and Beck quoting John Donne's "no man is an island, entire of itself" to illustrate the requirement for relationships in object-oriented languages [20]. Balzer, Eugster and Gross claim object *collaborations* (modeled by relationships) are key to understanding large object-oriented programs [2], and Bierman and Wren claim that "the programmer is poorly served when trying to represent many natural *relationships*" [4].

If relationships are essential to object-oriented design then why are they not supported in object-oriented languages, and why are they not in the top eight "Quarks" of object-oriented development? There have been several attempts to add relationships to OO languages [2, 4, 20, 19, 21] but none have achieved widespread use. We believe that this not because

relationships are irrelevant to implementation, but rather that there is no accepted understanding of what constitutes a good relationship model.

1.2 The Solution

This work aims to address the disconnect between object-oriented design and implementation by rethinking the way object-oriented languages are structured. We have developed a set of requirements with which to identify good relationship models, and used these requirements to develop a new model for the object-oriented paradigm which focuses on relationships rather than objects. We will test the effectiveness of the model by designing a language which uses it, along with a formal specification and a practical implementation for the language. We will measure the effectiveness of our model by conducting case studies comparing development in existing languages with development in our relationship-based language.

1.3 Proposal

This proposal begins by discussing the related work in this field, then outlines the requirements which we have identified and presents our early work towards developing a new relationship model and an associated language. This proposal is split into the following chapters:

- Background: This chapter examines the history of the object-oriented paradigm and the evolution of relationships as a part of object-oriented methodology. We examine the literature related to relationships in object-oriented languages and identify the important points of each proposed solution to the problem of relationships in an object-oriented system.

- Requirements: The Requirements chapter presents a list of requirements for a meaningful relationship model. We motivate each re-

quirement with reference to the literature and examine the related work introduced in the background chapter to identify the approach others have taken to each issue.

- Model: This chapter introduces the first iteration of our model for object-oriented systems which combine relationships with the existing object-oriented paradigm. We introduce the model and discuss our prototype language which implements this model. Finally, we compare our language with existing relationship-based languages and discuss the anticipated issues with implementing our language.
- Validation: This chapter discusses the steps we will take to verify the effectiveness of our model in comparison with the object-oriented model and the models for relationships presented in the related work.
- Work Plan: Finally, we will present an outline of our thesis, and a plan for its completion.

Chapter 2

Background

This Chapter discusses the place of first-class relationships in object-oriented programming languages. We begin with a brief history of the object-oriented paradigm, particularly in relation to the development of relationships as a part of the paradigm up until the development of UML in the mid-nineties. We then discuss UML's associations, and their relevance to first-class relationships in the object-oriented paradigm. Finally, we survey existing work which has discussed relationships in object-oriented programming languages.

2.1 A Brief History of Relationships

Object-oriented programming traces its origins back to simulation languages developed early in the history of computer science and digital simulation [16]. In particular, the Simula language described a system for modelling a collection of concurrent processes as a single program [7]. Simula broke away from the procedural programming languages of the day by allowing methods to maintain local state between calls. This changed the nature of a program trace from a strict tree structure to a more complex graph structure modelled by data and behaviour grouped together to create objects, creating the illusion of multiple program threads and simulat-

ing real-world collaborative processes [16].

The Simula language provided support for relationships through references to objects. Common practice in Simula dictated that objects were subordinate to the objects which created them (composition relationship) and generally could not refer to their creators. As a result there was little need for multidirectional relationships, which are harder to model with references. As the object-oriented paradigm evolved and models became more sophisticated multidirectional relationships became more common, exposing the limitations of references as a model for relationships.

2.1.1 Entity-Relationships Diagrams

Simula models were object (entity) centric, but Chen identified that systems could be modelled more elegantly using entities and relationships [6].

The Entity-Relationship model gives equal importance to entities (objects) and the relationships between them (associations). Entities are composite values, for example an entity representing a student enrolled in a university might be described by the tuple:

$$(\text{“300001234”}, \text{“Joe Bloggs”}, \text{“2002”})$$

Entities in Chen's system are classified by *entity sets* (corresponding to classes in the object-oriented paradigm). For example, the set of all students enrolled in a university can be classified as the set of entities with a student id, name, and year of enrolment. Entities nominate a particular value which can be used to distinguish them as their key. Figure 2.1 shows part of a student entity set expressed in an Entity diagram where students are distinguished by their student id.

Chen's system defines relationships as entities which cannot be uniquely distinguished by any single attribute. They are distinguished instead by several attributes, generally corresponding to other entities. For example, students may attend courses and obtain a grade for their performance in the course. The grade is a property of the student attending the course and

Attribute	**Student ID**	First Name	Last Name	Year of Enrolment
Entity	300002424	Amy	Anderson	2002
	300001234	Joe	Bloggs	2002
	300004132	James	Clarke	2003
	...	...	...	...

Figure 2.1: A *STUDENT* Entity Set (Primary key is Student ID)

Entity Relation Name	Student	Course	
Role	Attendee	Attended	
Entity Attribute	**Student ID**	**Course ID**	Grade
Relationship Tuple	300002424	COMP102	A
	300001234	ECON101	C
	300001234	BIOL103	B
	...	...	...

Figure 2.2: *ATTENDS* Relationship expressed in Chen's Relation notation. Primary Key is the (Student ID, Course ID) tuple.

is associated with, but independent of, both the student and the course. Consequentially, we define a class of relations between students and courses called *ATTENDS* and the grade becomes a property of the relation as demonstrated in Figure 2.2.

2.1.2 Relations as Semantic Constructs

Relationships as described by Chen were well used in databases and semantic data models but were, initially at least, ignored by object-oriented programming language designers. Rumbaugh was the first, in 1987, to propose that they should be added to object-oriented languages. Rumbaugh recognised that objects in an object-oriented system correspond to the entities described by Chen: an object can be regarded as a record which is distinguished by a unique key (the object's memory location). However,

object-oriented systems do not have a component corresponding to Chen's relationships. Rather, object-oriented systems implemented relationships using boiler-plate code which is irrelevant to the applications, as demonstrated in Figure 1.2.

> It is possible to program [relationships] using existing object-oriented constructs, but only by writing a particular implementation in which the programmer is forced to specify details irrelevant to the logic of an application. *(Rumbaugh, [21])*

There are two typical approaches to implementing relationships in traditional object-oriented languages: a binary relationship can be represented as a pair of references, or sets of references (see Figure 1.2) or reified as a class. In the first case, there is burden placed on the programmer to ensure that the pairs remain consistent and it is difficult to associate extra properties (for example, grades) with the individual links because there is no clear place to store them. In the second case the programmer must take care to preserve link uniqueness as the links will gain an identity from their memory location which is independent of the objects they link, unlike relationships in Chen's system which are distinguished by the entities (objects) that they relate. In addition, both systems create strong coupling between the related classes, limiting their reuse potential and creating fragile code.

Rumbaugh introduced a relationship construct for object-oriented languages which adds relationships to object-oriented systems. His relationships are tuples of object references which are similar to relations in Chen's Entity-Relation diagrams; they are distinguished by the objects they refer to. Rumbaugh's relationships are declared at the class-level as a tuple of class references which are instantiated as tuples of object references. Figure 2.3 demonstrates using Rumbaugh's language: a relationship can be declared in a similar manner to a class, and links can be added between objects of compatible type. The language maintains the links for the pro-

```
RELATION Attends
    (attendee: Student, attended: Course)
...
Attends.add(joe, comp);
joe.put_attended(math);
comp.put_attendee(jane);
```

Figure 2.3: A demonstration of programming with DSM-style relationships.

grammer and preserves link uniqueness by maintaining relationship instances as sets.

Rumbaugh and others implemented a language which provided relationship support called DSM [26]. DSM was based on C, with extensions providing OO and Relationship features. The language did not achieve widespread use, possibly due to competition from C++ and Smalltalk, but Rumbaugh's work laid the foundations for much of the relationship literature which has followed.

2.1.3 Object-oriented modelling

The early '90s saw the introduction of several design methodologies for object-oriented systems which modelled relationships explicitly [3, 5, 23]. Among other advantages, these mitigated many of the problems associated with the lack of relationships support which Rumbaugh identified. Object-oriented practitioners could model relationships at the design level without requiring explicit support at the implementation level. Programmers still had to write the boilerplate code, but they could use a conceptual model which didn't consider the mundane details of relationship implementation.

Two of the most popular object-oriented development systems were

OMT (Rumbaugh *et al.*) [23] and the Booch Method (Booch) [5]. Both systems included a diagramatic description of the classes in a system and their interactions, expressed as a graph where nodes represented classes or objects and edges represented relationships. Edges in both systems were more similar to the relation tables in Chen's work than to common implementation idioms based on references because they were separate from the types they related.

Concurrently with the development of OMT and the Booch Method, Cunningham and Beck developed a brainstorming tool called the Class Responsibility Collaborator (CRC) cards [3]. The CRC card system modelled each class in isolation with a piece of card detailing the name of the class, its responsibilities, and the classes it collaborated with. The collaborations defined implicit unidirectional relationships between classes which could be directly implemented as references, unlike the predominantly bi-directional relationships in Booch and Rumbaugh's systems. Although the CRC card model was closer to object-oriented implementation than the graph models of Rumbaugh and Booch, it ultimately proved to be less popular suggesting that bi-directional graph-based models are more suitable for modelling object-oriented systems.

2.1.4 The Unified Modelling Language (UML)

During the mid '90s Rumbaugh, Booch and Jacobson produced a new modelling system, designed to unify their various techniques and provide an industry standard. The resulting system was the Unified Modelling Language (UML) which was accepted as a standard for modelling object-oriented systems by the Object Modelling Group (OMG) in 1997 [25]. UML's approach to relationships followed OML and Booch's method; modelling relationships as explicit entities independent of the participating classes [5, 23]. UML has subsequently become the dominant modelling language for object-oriented systems and the standard has been updated several times.

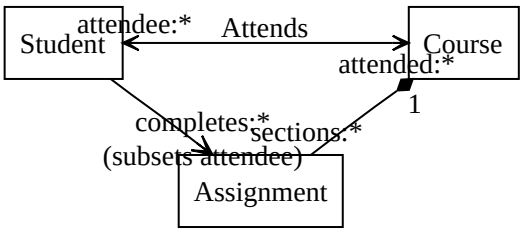


Figure 2.4: A UML class diagram describing a system for a university enrolment system based on the example introduced in Figure 1.1. This diagram includes two new relationships: *Requirement*, a composition relationship between Course and Test which means that the course has zero or more tests which are components of the course, and *Takes*, a relationship between Student and Test which means that a student may *take* a test.

One of the primary tools UML provides is the *class diagram* which describes the structure of an object-oriented system as a collection of classes and relationships, similar to the diagrams in OMT and Booch's method but with some extensions. We discuss the class diagram here because, as the dominant model for object-oriented systems, its definitions of objects and relationships have become the standards for the object-oriented paradigm. Any new model which we define must be described in relation to UML's model.

UML class diagrams are graphs composed of nodes and edges. Nodes may be *Class*es or *Interface*s and edges can be *Association*, *Aggregation*, *Composition*, *Dependency*, *Generalization*, *InterfaceRealization*, and *Realization* [17]. We do not discuss Interfaces and Generalization / Realization as they are largely orthogonal to the relationships this work considers. UML also describes several properties of associations which are useful in a discussion of relationships. They are demonstrated in Figure 2.4.

Association

An association between classes (participants) in a UML diagram in-

dicates that there can be links between those classes' instances in a running system. Associations can have two or more participants (binary or greater arity). Associations are important because they are closely related to relationships. As there is no generally accepted definition of what a relationship is it is difficult to determine where associations and relationships differ, indeed, some relationship work equates the concepts [19].

Association End

UML allows the ends of an association to be named, and have various attributes. Association ends are generally properties of the association, but the system designer may specify that they should be properties of the class. This is specified with a dot on the end, as demonstrated in Figure 2.4 on the *Takes* association on the Student end. This is has an implied meaning for implementers, indicating that the association should be embedded as a field (or group of fields) in that class.

The names on association ends describe the role that the object plays in that association. For example, the course in the 'Attends' relationship is 'attended'. These names also allow UML diagrams to refer to the sets of objects which are in a particular association. For example, the students which are attending courses can be referred to using the association end 'attendee'.

Multiplicity

Annotating an association end with its multiplicity indicates the number of links that an instance of the participating classes can have. For example, the association *Requirement* specifies that the class *Course* will have zero or more links to *Test* objects, while each test object will have exactly one link to a course object.

The constraints expressed by multiplicity annotations on associa-

tions are one of the more popular constraints which can be expressed with relationships; several groups have discussed implementing these in relationship systems [2, 4, 21].

Navigability

An association end may be annotated with an arrow which indicates that it is navigable in that direction. For example, the arrow on the *Takes* association in Figure 2.4 specifies that an implementation should provide efficient access from Students to Tests. The absence of an arrow does not indicate that the association should not be navigable (it may be possible to access Students from Tests) but the access may not be efficient.

Composition

A filled diamond on an association end indicates that the association is a composition association. Composition associations describe part-whole relationships between two objects, known as the composed object (part) and the composite object (whole). Composition associations must be binary because the classes involved must each conform to one of the two roles.

Composition associations are indicated with a filled diamond on the composite type's end of the association, as demonstrated by the 'Requirement' association between courses and tests in Figure 2.4.

Aggregation

An empty diamond on an association end indicates an aggregation association. Aggregation is similar to, but weaker than composition. Composition requires that the composed object *belongs* to the composite object, it cannot be shared. Aggregation allows the aggregated object to be shared between other objects. It is more of a delegation relationship than a part-whole relationship: the aggregate object

delegates behaviour to the aggregated object without requiring it to maintain state.

These properties defined in UML are particularly interesting in the study of relationships because they are properties of relationships which are commonly used by object-oriented practitioners. As such, they are good candidates for integration into object-oriented languages.

2.2 Implementing Relationships

This section discusses the implementation of relationships in existing object-oriented programming languages. First, we discuss two common strategies for implementing relationships and their associated benefits and problems. Then we discuss some of the related work describing techniques for implementing associations without language support of libraries.

2.2.1 Common Implementations

There is a clear distinction between UML models and the code which implements them. For example, the university enrolment system described in Figure 1.1 specifies that the association should be navigable efficiently in both directions, and that there can be any number of links which reference each participant. There are a number of ways to implement this association. The most common implementation would be to embed the relationship in the participants, as outlined in Figure 1.2 and detailed more in Figure 2.5.

The relationship implementation in Figure 2.5 uses references stored in the participating objects to represent the relationship. The objects must ensure that the relationship remains consistent: if a `Student` is added to a `Course` then the `Course` must be added to the `Student` and vice versa. Even with such a simple relationship there is more complexity than we would like in the implementation; only the methods for adding new links

```
class Student {
    Set<Course> attended;

    void addAttends(Course c) {
      if (!attended.contains(c)) {
        attended.add(c);
        c.addAttends(this);
      }
    }
}
class Course {
    Set<Student> attendees;

    void addAttends(Student s) {
      if (!attendees.contains(s)) {
        attendees.add(s);
        s.addAttends(this);
      }
    }
}
```

Figure 2.5: Implementing `Student` *attends* `Course` with embedded references.

are shown, there must also be methods to access and remove links. All of these must maintain consistency between the two collections of link ends. If the relationship is particularly sparse (very few of the existing students actually attend courses) then this representation wastes memory by allocating a collection in every student object. At an engineering level, this solution increases coupling between the classes: adding another relationship of a different type between `Student` and `Course` would require both classes to be modified. All of these problems are exacerbated when objects participate in many relationships.

Figure 2.6 outlines a different implementation for the attends relationship. Rather than embedding fields in the participants, this implementation uses a HashMap in a separate class to store the tuples. This is particularly memory-efficient for sparse relationships (where there are lots of students but not many attend courses) because the many students who do not attend courses would not need to have an extra field. This implementation would also allow many different relationships with similar participants to coexist without requiring an attribute for each to be added to the participants.

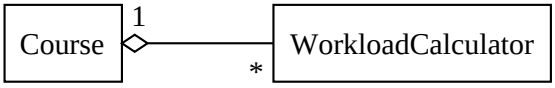
The problem with implementing relationships in existing object-oriented languages is that they either lose the relationship as a program entity (the relationship is spread between several classes), or encapsulate the relationship in another object at the expense of efficient navigability from the participants and clear identity for the links. If we choose either option then we are unable to switch between implementations without substantial refactoring if requirements chance. There is also a substantial burden for the programmer to implement and maintain boilerplate code.

Aggregation and Composition

Not all relationships suffer from boiler-plate code in implementation: relationships like Composition and Aggregation are quite simple to implement and maintain.

```
class Student {
    ...
}
class Course {
    ...
}
static class Attends {
    static void add(Student s, Course c) {
        Pair p = new Pair(s,c);
        attendees.add(p);
        if (students.containsKey(s)) {
          students.get(s).add(p);
        } else {
          Set<Pair> set = new HashSet<Pair>();
          set.add(p);
          students.put(s, set);
        }
        // and similarly for courses
    }
    static HashMap<Student, Set<Pair>> students;
    static HashMap<Course, Set<Pair>> courses;
    static Set<Pair> attendees;
    ...
    private static class Pair {
        Student student;
        Course course;
    }
}
```

Figure 2.6: Implementing `Student` *attends* `Course` with a hash set.



In an aggregation association an object delegates behaviour to a sub-component, which may be shared between aggregates. Because the aggregated object may exist independently of the aggregate object it cannot rely on the presence of the aggregate object, so it is unusual for the aggregated object to maintain references to the aggregate. As a result, communication between the objects is initialised by the aggregate. If the aggregated object ever references the aggregate then this is done with a call-back reference, rather than a reference stored in a field.

Aggregate relationships can easily be implemented with a single reference. For example:

```
class Course {
    WorkloadCalculator calculator;
}
```

The `WorkloadCalculator` performs services for multiple Courses, so it will not initiate communication with a course and so does not need a reference to courses.

2.2.2 Design Patterns

One of the primary problems with relationships in object-oriented languages is the boiler-plate code required to implement them. Many types of relationships can be implemented in a reasonably straight-forward manner, however there are other relationships which are quite complex. Gamma *et al.* describe a method for documenting complex techniques for solving recurring problems known as "design patterns" [8]. In their landmark book the authors describe several patterns for designing systems which, while they do not deal directly with relationships, depend heavily on them

for their solutions. In fact, the majority of the patterns deal with complex structures composed of relationships and objects described in UML.

Noble uses design patterns to describe common implementation strategies for relationships [13]. He claims that the prevelence of relationships in object-oriented design means they should be *easy to write*, *simple to represent*, and *immediately understood by later programmers*[13]. These concerns are addressed by identifying five common relationship uses and describing them as design patterns.

As demonstrated in Figures 2.5 and 2.6 it is possible for a simple relationship to require a reasonably large amount of code. Noble describes five different patterns which simplify the identification and implementation of common relationships. For example, the *Relationship As Attribute* pattern describes the double reference solution similar to the code demonstrated in Figure 2.5 for implementing one-to-one relationships. A more complex many-to-many relationship, for example the Attends relationship introduced earlier, can be implemented using the *Mutual Friends* pattern.

Noble provides five different patterns for relationships in object-oriented systems: *Relationship as Attribute* and *Relationship Object* describe two basic techniques for implementing relationships, *Collection Object* and *Active Value* describe particular subsets of *Relationship Object*, and *Mutual Friends* describes a technique for implementing two-way relationships as two one-way relationships in order to reduce complexity.

Noble's paper is particularly important because it identifies the lack of relationship support, observes that this causes similar problems to be solved many times, and provides codified solutions which can be used for both implementing relationship and identifying relationships from their implementation. The specific relationship patterns which Noble identifies are quite straightforward, but like the patterns identified by Gamma *et al.* the act of identifying and codifying similar problems reduces the burden on both creator and maintainer by moving the solutions from the domain of the individual to the community.

2.2.3 Object-Oriented Relationships

Noble and Grundy describe an alternative approach to object-oriented modelling; they suggest programmers reify relationships as classes during system design so that they can be clearly translated to implementation [15]. They identify three phases of software engineering; analysis, design and implementation, and observe that while relationships are explicit in the first two they are obscured in the third, reducing traceability and code comprehension.

Noble and Grundy argue that many-to-many relationships and other relationships with complex behaviour cannot be directly implemented, and therefore should not be present as such in the system design. Instead, the relationships should be reified as classes which encapsulate the representation and behaviour of the relationship. The new class is responsible for maintaining the relationship, which reduces the coupling between the participants, and the explicit representation improves traceability between design and implementation.

Figure 2.7 shows a revised design of the enrolment system from Figure 1.1 which follows their recommendations. The only difference between the examples is the attends relationship (many-to-many) which has been reified as a class with two one-to-many relationships. The one-to-many relationships have no semantic meaning in the system, they are merely a syntactic means of connecting the components. One-to-many relationships are easier to implement: the *one* end is implemented with a reference, and the *many* end with a set. The new relationship object becomes the authoritative store for the relationship, responsible for maintaining the relationship.

The authors conducted a case study using their approach and compare it favourably with traditional design. They observe that objects tend to be smaller and more reusable using their technique, that design and implementation are easier to understand and modify, and that development is more traceable.

Figure 2.7: A revised version of Figure 1.1 which uses Noble and Grundy's suggestions on system design [15]. Note that the attends relationship is now an explicit object and the new associations are unnamed one-to-many associations. These can be implemented more easily than many-to-many relationships.

The authors do not consider the implications of their approach on system design. As they observe: *"There ain't no such thing as a free lunch"* (Heinlein). A powerful feature is missing from the implementation phase, but is ignoring it in design an acceptable solution? We believe that the answer lies in the opposite direction; relationships should be explicit in implementation in the form of relationships, not objects.

2.3 Libraries of Associations

As an alternative to describing relationship implementation techniques in patterns, several authors have proposed libraries of relationships. A library implementation removes the burden of boiler-plate code from the programmer, allowing them to use general solutions instead. As with any library, a good relationship library needs to be unobtrusive, hiding implementation detail without requiring excessive overhead or additional complexity.

2.3.1 Noiai

Østerbye presents a library of associations for C# which is particularly elegant in its implementation [19]. Aptly named *Noiai* (No Object Is An

Island), the library makes use of C# runtime generics and some useful tricks to avoid much of the boilerplate code commonly encountered when implementing relationships.

Østerbye identifies that relationship representation and relationship access are largely orthogonal issues [18]. Relationships can be represented as properties of the participants, or independent of the participants (see Figures 2.5 and 2.6). Independently, they can be accessed directly from the class (via attributes) or through an external association interface. Østerbye demonstrates how the Noiai library can support any of the combinations presented, and shows how these features are provided by the library. This means that programmers can use internal fields or external association references to access associations, and the associations can be represented at runtime either internally or externally to the participants.

Østerbye's work is the most capable work of its kind to be implemented in a mainstream language without modification. This is largely because it requires features which have only recently become available in mainstream languages. It remains to be seen whether this approach to using relationships in object-oriented languages will prove popular.

2.3.2 Relationship Aspects

Pearce and Noble present the *Relationship Aspect Library* (RAL), a library of aspects implemented in Aspect/J which defines a standard relationship interface and provides various relationship implementations. Like Østerbye's Noiai, the RAL allows either internal or external implementation of relationships, however they limit programmers to external access: associations are accessed via a separate class, rather than using fields of the participants.

The authors identify a hierarchy of relationships and provide generic interfaces which ensure loose coupling between relationship implementation and use. Like Noiai, this allows programmers using the RAL to switch

```
aspect Attends extends
        SimpleStaticRel<Student, Course> {  }
SimpleRelationship<Student, Course> a =
    Attends.aspectOf();

Student joe = new Student(...);
Course comp = new Course(...);

a.add(joe, comp);

for (Course c: a.from(joe)) {
    System.out.println("joe attends " + c);
}
```

Figure 2.8: Implementing the attends relationship using the RAL. This example is based on an example by Pearce and Noble [20].

between different relationship representations without having to refactor code.

Figure 2.8 demonstrates an implementation of the attends relationship with the RAL. The `Attends` aspect extends `SimpleStaticRel`, which is a simple relationship (a relationship without attributes on the relationship edges). The relationship is called *static* because it exists in the program scope; it can be referenced by its type without reference to an instance. If we wish to use a dynamic (instance-based) relationship instead we can extend `SimpleHashRel` instead of `SimpleStaticRel`. We could also add fields to the relationship edges by using `StaticRel` and passing in a `Pair` type as a generic parameter to be used as the relationship edge. Either of these changes can be effected without changing code outside the declaration of `Attends`.

Unlike other work in this area[21, 24, 4, 2] Pearce and Noble consider

a relationship to be the extent set of the relationship links. This facilitates the creation of multiple instances of a single relationship in a single program. For example, the attends relationship could be instantiated twice to represent enrolments for two different years. It would also be possible to implement this with a ternary relationship between students, courses, and years. However, as the year is a property of the set of links, rather than the individual links, creating separate sets of links is a better model for the problem.

Extent set support also allows the RAL to implement constraints on the relationship instance, for example "tree" or "transitive". The constraints must still be implemented by hand, but the modular nature of the library means that code related to relationship constraints is encapsulated within the relationship rather than the class.

The authors claim that using the RAL raises the level of abstraction in programs. Instead of explicitly managing relationship implementation programmers are able to use sensible defaults, but still have the freedom to implement specific behaviour without having to work from scratch. We agree that the RAL raises the level of abstraction in a good way, but we do not believe that the abstraction which they introduce is the best way to describe relationships. For example, the RAL does not provide a way to define properties of relationship participants which are dependent on the relationship and the participant without embedding the properties in the original object.

2.4 Relationship Languages

As an alternative to library implementations several authors have described language extensions to support relationships. The earliest example of this was Rumbaugh, whose work we discussed in §2.1.2, however there was little follow-up work. There has been a recent resurgence of proposals for relationship extensions however, some of which we describe in this sec-

tion.

2.4.1 First-class Relationships in an Object-oriented Language

Bierman and Wren define a language called *RelJ* which extends a small, functional subset of Java to provide first-class support for relationships [4]. RelJ allows programmers to define relationships between objects, specify attributes and methods on the relationships, and create relationship hierarchies. The authors provide a complete formalism for RelJ with a type system and small-step operational semantics for the system, however they do not provide an implementation.

Like UML associations, the relationship model presented by Bierman and Wren is a class-level definition of a runtime link. RelJ defines a relationship as tuples with fields and methods. Unlike UML, relationships exist in a single-inheritance type hierarchy similar to Java's class hierarchy, with *Relationship* as its root. Relationship declarations share most of the properties of Java classes. The attends relationship between students and courses would be declared as follows:

```
relationship Attends from Student to Course {
    int mark;
}
```

The attributes would become properties of the links in a running system, so they can be accessed through link instances:

```
alice = new Student();
programming = new Course();
aliceAttends = (alice.Attends += programming);
aliceAttends.mark = 'A';
```

RelJ's relationship inheritance is based on delegation. Relationships do not subsume their super-relationship when they are instantiated: rather

both the subtype and the super-type are instantiated with the subtype passing all calls to the super-type in the manner of the inheritance described in Self[30]. For example, we can extend a relationship twice and have a single link which conforms to both types:

```
relationship ReluctantlyAttends extends Attends
    from Student to Course {}
relationship CompulsorilyAttends extends Attends
    from Student to Course {}
aliceRAttends=(alice.ReluctantlyAttends+=programming);
aliceCAttends=(alice.CompulsorilyAttends+=programming);
aliceRAttends.mark = 'C';
System.out.println(aliceCAttends.mark) //prints "C"
```

Structuring inheritance in this manner at runtime means that several subtypes may share a single super-instance. This would cause some particularly interesting issues at runtime if the system were implemented: in particular, it is not clear how or whether the authors would implement virtual method invocation. A single super-type instance might have several subtype instances delegating to it so it would not be clear which subtype's method should be invoked unless the authors implemented some type of context, which they do not discuss.

2.4.2 A Relational Model of Object Collaborations and its Use in Reasoning about Relationships

Balzer *et al.* describe a relationship model which supports complex relationship constraints. They build on the RelJ language [4] and describe an invariant constraint language based on mathematical relations for expressing relationship constraints [2].

Balzer *et al.* don't provide a formal language or implementation, but there are several interesting issues raised in the paper. To support constraints, the authors introduce *Member Interposition*, a technique which al-

lows relationship declarations to modify the relationship participants. Relationships can inject fields into the participants which are only accessible from the context of the relationship. This concept is similar to Aspect/J's private member declarations which Noble and Pearce exploit for the RAL [20]. It is particularly useful for relationships as it aids in de-coupling relationship implementation from the participants.

The authors also identify four unique types of relationship constraints:

Structural Intra-Relationship Invariants

Much like multiplicity annotations in UML Structural Invariants restrict the possible runtime instances of a particular Relationship. An intra-relationship invariant restricts the allowable set of relationship instances of a particular relationship in isolation from other relationships. Examples include multiplicity constraints like those expressed in UML, but also requirements like "this relationship is surjective, asymmetric, and irreflexive".

Structural Inter-Relationship Invariants

Like the Structural Intra-Relationship Invariants, these invariants restrict allowable relationship instances based on structural constraints. However, these invariants can be expressed across multiple relationships. For example, a student can only take a test if they are enrolled in the Course the Test is for (see Figure 2.4).

Value-Based Invariants

Like structural invariants these come in intra- and inter-relationship variants, but the essential concepts are the same: value invariants maintain constraints on the values of class and relationship fields of the objects and relationships in a program. For example, a value-based constraint would be a requirement that students can only enrol in a course if they have paid their fees for previous course. This relationship requires that the language enforces constraints on mutable

state of objects, which is obviously a complex and difficult task. The authors do not provide a mechanism for enforcing these constraints but they do discuss some of the issues in implementing their model.

2.4.3 Declarative Object Identity Using Relation Types

Vaziri *et al.* define a new language construct called a relation which has a dependent identity similar to Chen's relationships discussed in §2.1.1 [31]. Though not designed specifically to support relationships relation types are declared as relations between existing objects, and they have an identity which may be computed from one or more other objects/relations. For example, a relation type could be declared for the attends relationship which *keys* on Student and Course. Instances of the relation would require a reference to a Student and a Course which would define the identity for that relation instance and the language would guarantee that the instance reference cannot change and that only one relation instance could be created with that pair of instances. However, Vaziri *et al.*'s system does not provide access to the extend set of the relations so the relation instances would still have to be stored in a similar manner to the relationship implementation described in Figure 2.6.

Varizi *et al.*'s work is not a whole answer, however they provide an important contribution to relationships in object-oriented languages: links can be created which do not have independent identity, removing from the programmer the burden of preserving link uniqueness. This feature would certainly be useful to programmers attempting to implement relationship support in a library.

2.5 Conclusion

This chapter introduced relationships in the context of object-oriented programming, discussed various reasons for providing relationship support,

and described relationships in modelling. We introduced some definitions for relationships in this thesis, and described implementation techniques. We also covered the literature relating to providing relationships in object-oriented languages.

Chapter 3

Requirements for a Relationship Model

In Chapter 2 we identified that relationships are an important part of the object-oriented methodology, and that they are represented in object-oriented programming languages. This results in programmers writing boiler-plate code which is error-prone, difficult to maintain, and irrelevant to the logic of applications. As a result, code is less portable, less reusable, and there is less traceability between design and implementation.

All of these problems can be addressed by introducing a relationship model which eliminates the boiler-plate code. In addition, a relationship model would improve representation independence, separation of concerns and requirements representation for relationships. Model users would be able to switch between different implementations without refactoring (representation independence) [18, 20], code relevant to relationships would be associated with the relationships rather than the participants (separation of concerns) [20], and many program requirements could be clearly expressed as relationship constraints (requirements representation) [2].

Chapter 2 also identified several attempts to develop a relationship abstraction for object-oriented programming languages. Each one has addressed some of the issues associated with relationships but each has as-

sociated problems which prevent it being a good general solution.

We plan to design a new relationship abstraction for object-oriented programming languages which builds on the existing work while addressing the problems associated with it. To do this we will examine the advantages and disadvantages of each existing abstraction to provide a list of requirements for the abstraction.

The relationship abstraction we develop will be presented as an extension to an existing object-oriented system which adds relationships as a first-class (explicit) construct. It will most likely be a combination of language extensions and a library of relationships. The requirements for the abstraction will affect how much of the system can be provided by a library and how much must be provided via language extensions. It may be necessary to prioritise some requirements at the expense of others.

The relationship system we develop will be described theoretically and practically as a formal model and an implemented system. The formal model will be described in a similar manner to other work in this area [4, 9]. The implementation will consist of a compiler (or pre-compiler to an existing language) and a supporting library system.

The system will be verified both theoretically and practically. The theoretical verification will be provided by literature-standard techniques such as proofs of type-soundness, correctness, and completeness. The practical verification will consist of a series of case studies to demonstrate that the system does solve the various issues identified in a reasonable manner.

This chapter identifies the issues which need to be addressed for a relationship abstraction to be successful. We identify two categories of issues: theoretical (abstract) issues and implementation issues. These are discussed in the following two sections. The third section of this chapter presents our thesis; a prototype for the relationship model which we will develop to solve the issues we have identified. It also introduces some early work in implementing our abstraction. Finally, we discuss the validation techniques we will use to identify whether the issues have been

addressed.

3.1 Issues

This section describes the issues which we have identified as important components of a relationship abstraction. These issues have been drawn from a variety of sources, particularly Armstrong's *Quarks of Object-Oriented Development* [1], Booch's *Object-Oriented Analysis and Design* [5] and related work discussed in the previous chapter. Several of these issues were also identified by Noble as important properties of a relationship system [14].

3.1.1 Abstraction

An *Abstraction* in object-oriented programming describes "*the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries*" (Booch [5]). Information hiding is key to abstraction, allowing programmers to consider the essential nature of a component rather than the details of its implementation. Booch identifies abstraction as one of the four major elements of the object model so it is likely to be important to provide abstraction in a relationship system also if we are to preserve the strengths of object-orientation.

Abstraction for objects in object-oriented programming is provided by the object's interface; the interface details what the object can do, rather than how it does it. One of the major problems with relationships in object-oriented programming which we have discussed, boiler-plate code, is essentially due to the lack of abstraction for relationships. If object-oriented languages provided relationship abstraction then the programmer would be free to consider the essence of the relationship rather than its implementation as the relationship's implementation would be hidden behind a clear interface.

Most of the related work provides strong relationship abstraction; Rum-

baugh provides limited abstraction as his relationships consist only of pairs [21]. The essence of simple relationships is captured, but abstraction doesn't support more complex relationships where the relationship may have variables and methods. Bierman and Wren and Vaziri *et al.* support full abstraction for relationships at a similar level to objects; relationships may have fields and methods, and have a meta-level construct akin to a class which defines their interface [4, 31]. Pearce, Noble and Østerbye reify relationships as classes so natively support the same abstraction as the system they inhabit [20, 19].

We see abstraction as a key component of a relationship system. If a relationship system is to be used, it is essential that it improves the conceptual understanding of relationships so that programmers can focus on the details of what, not how.

3.1.2 Polymorphism

Polymorphism in the object-oriented paradigm allows different implementations for a common interface [1]. Two implementations may be used interchangeably if they provide a common interface. This allows practitioners to define an interface without specifying an implementation, or modify the internal behaviour of an object without affecting its neighbours. This is an important mechanism for code reuse, and an important property of the object-oriented paradigm so a relationship abstraction should support relationship polymorphism as well.

There are several types of polymorphism possible for relationships. We consider two to be particularly useful: relationship instance polymorphism and link polymorphism.

Relationship Instance Polymorphism

Relationship instance polymorphism describes polymorphism on the instantiated relationship. This allows different implementations of a rela-

tionship instance to be used interchangeably. For example, an enrolment system with students attending courses might define several implementations of the attends relationship, one using a hash set representation and another storing links in the participating objects. A third implementation might add hooks to the relationship to store links in a persistent database and maintain consistency between the database and the program.

Pearce and Noble provide some support for relationship instance polymorphism with dynamic relationships in the RAL [20]. Dynamic relationships can be created explicitly and passed as references so programs using dynamic relationships can interchange the relationship implementations. Static relationships on the other hand are accessed via their static type; this limits polymorphism because the code would have to be re-factored to refer to another implementation. Similarly, the other related work does not provide polymorphic access to the relationship instance [21, 4, 31, 19]. This may be because most work in this area considers the relationship instance to be analogous to a class instance accessed via reflection; the relationship is used to create new links and compare types, but little else.

Relationship instance polymorphism is an aspect of relationships which is certainly desirable for relationship models to support, but it is not clear whether it is necessary for this to be provided as a separate feature. It may be that link polymorphism is sufficiently expressive for the types of polymorphism programmers require, as demonstrated by the UML (link based) definitions of multiplicity, uniqueness, and navigability [17], but this does seem unlikely.

Link Polymorphism

Link polymorphism allows multiple link implementations with a consistent interface which can be used in the same relationship. For example, one *Attends* link implementation may represent courses which have been completed and have a grade attribute, while another implementation protects the privacy of the students attending a course by restricting access to

their properties. Both provide the basic functionality of a link between students and courses, but provide extra functionality for specific situations.

It is not clear how polymorphic links should be created and this depends on other choices for the system. For example, if the relationship instance is not responsible for creating links then it is reasonable to allow different links to be stored within the instance so long as they conform to the interface for the relationship (they relate the appropriate types and they have any properties that the relationship requires). Assuming that the abstraction allows polymorphic links to be created (there are various patterns which are relevant, particularly *Abstract Factory* [8]) then link polymorphism would behave in a similar manner to object polymorphism.

Pearce, Noble and Østerbye allow link polymorphism by taking a parameter corresponding to the link implementation to be used for that relationship. They do not, however, allow arbitrary implementations to be interchanged at runtime. Bierman and Wren's system does not support link polymorphism within the relationship as implementations are tied to the static relationship which creates them. They do allow link sub-typing however, and as links can be passed as references this means that there is link polymorphism support as long as the relationship type is not referenced.

Link polymorphism seems to be a useful feature which we would like to support. It is not clear whether we should provide some mechanism for dynamic creation of links so that a relationship may consist of different implementations, or follow Pearce *et al.* and provide support for switching implementations when a relationship is instantiated.

Participant Polymorphism

Participant polymorphism is a third type of polymorphism which, we assume, any reasonable relationship abstraction will allow. It allows participant's implementations to be interchanged. This is traditionally supported by *dynamic binding* in object-oriented programming [5] and as links refer

to references in relationship models, there is no reason why it should not be supported in relationship systems via the same mechanism. Indeed, all of the related work which supports object polymorphism supports participant polymorphism as a consequence. We intend to support participant polymorphism.

3.1.3 Specialisation

Specialisation is a means for defining a type hierarchy. A class which specialises another class describes a subset of the later class. This is typically associated with *inheritance* in object-oriented programs as a specialised class will often extend a more general class and override functionality when it is necessary, rather than re-implementing the whole class. There is no reason why a relationship system should not support code reuse through this mechanism, however there are some caveats if the system also supports polymorphism.

Specialisation is closely tied with polymorphism: objects of a specialised type can be used in place of more generic types as long as they are compatible with the generic type's interface. The interaction between specialisation and polymorphism can be problematic if it allows variance. For example, a method which is overridden for a specialised class must be contravariant on its parameters and covariant on its returned type(s). This also affects languages with generic types: in general generics are invariant, however the invariance can be relaxed if certain guarantees are made. For example, if a generic type is final (read only) then it is covariant, and if it is write-only then it is contravariant.

A relationship abstraction which allows specialisation on participants and also supports link polymorphism corresponds to a system which allows generic types. For example, consider the system described in Figure 3.1. The classes *LazyStudent* and *HardCourse* are specialisations (subtypes) of student and course respectively. There is a relationship between stu-

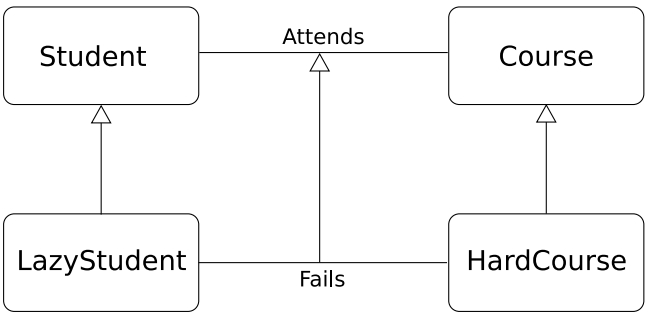


Figure 3.1: This figure illustrates a program model which is not supported by traditional object-oriented systems.

dents and courses called *Attends* and a relationship between HardCourses and LazyStudents called Fails. This system is entirely reasonable as long as the specialisation between Fails and Attends is not included: LazyStudents can attend courses and students can attend HardCourses. LazyStudents can even attend HardCourses. However, a Fails relationship cannot be used in place of an Attends relationship because a Fails relationship cannot create or store Student-Course tuples, and an Attends relationship cannot be used in place of a Fails relationship because clients retrieving a link from the relationship would receive Student-Course tuples when they expect LazyStudent-HardCourse tuples.

Most of the relationship abstractions in the literature have taken a conservative approach to this issue. Rumbaugh allows specialisation but not polymorphism, avoiding the issue [21]. Bierman and Wren also allow specialisation but not relationship polymorphism, however their system uses delegation for specialisation which allows link polymorphism [4]. Pearce and Noble provide both polymorphism and specialisation, however they use generics to do this so their relationships are invariant [20]. Østerbye does not allow relationship polymorphism or link polymorphism but does support several types of specialisation [19].

Specialisation is a useful property which is supported by UML and

by most relationship abstractions in the literature so any new abstractions should also provide specialisation. It is theoretically possible to allow covariant or contravariant use of relationships under certain circumstances but the relationship abstraction would need to present this in a natural and intuitive manner for it to be useful.

3.1.4 Reusability

The OO paradigm uses abstraction to improve reusability: a class which was written for one system can be reused in the same system or another because it is not statically coupled with the rest of the system. In the same way, a relationship abstraction can allow relationships to be reused: for example a relationship might be instantiated multiple times in the same system. This cannot be done in traditional OO systems where the code implementing a relationship is statically woven into the rest of the system.

Relationship reuse is one of the features of object-oriented languages which is not well supported by existing relationship systems. Most systems couple relationships with code by referring to them by their static names [19, 4, 21]. This is akin to using global fields: it becomes impossible to reuse code in another system which uses that field name for a different purpose.

The only relationship system which does support relationship reuse is the RAL [20] which supports multiple instances of dynamic relationships. Relationship instances can be created and passed as references. Unfortunately, dynamic relationships are not able to use the RAL's aspect features which limits their usefulness.

In addition to reusing objects and relationships, most object-oriented languages support code reuse through inheritance. This is typically associated with specialisation but is in fact a separate issue [28]; specialisation provides a type hierarchy for polymorphism, while inheritance provides code reuse. Traditionally inheritance is tied to specialisation, so we will

need to consider whether it makes sense to separate these issues in our abstraction.

3.1.5 Composition

Composition is an important concept of the OO paradigm: components of the system can be composed together to allow implementation to be shared, and to reduce complexity. In the same way a good relationship abstraction should allow relationships to be composed.

UML allows relationship composition for tuples in specific cases, namely, when the relationship has been reified as an *Association Class* [17]. This is also the case for relationship systems which reify links as the link can then be referenced directly [4].

It is also possible to support relationship composition at the relationship instance level. This typically takes the form of a set operation, for example UML supports *derived associations* which are unions of other associations. In such a system, the derived relationship is a composition of the relationships it derives. Østerbye supports these in Noiai library [19] although they are read-only. Balzer *et al.* describe intra-relationship constraints which allow relationships to be composed and derived in a variety of ways. For example, *"the set of (student, course) tuples where the student is a graduate and the course is an undergraduate course"*. Object-oriented query languages like LINQ [11] and JQL [32] provide support for constructing tuples with queries which are similar to the constraints expressed by Balzer *et al.*

3.1.6 Separation of Concerns

The Aspect Oriented methodology [10] claims that separating different program concerns produces programs which are more traceable, less tightly coupled, and more reusable. Pearce and Noble claim that relationships are cross-cutting concerns which can be implemented as aspects [20]. It

seems reasonable that relationship implementations should be removed from participating classes. This is one of the potential advantages of a relationship abstraction: implementation of the relationship is hidden behind the relationship abstraction which reduces the interleaving of the relationship and its participants.

Relationship abstraction does not, however, provide a complete answer to the separation of concerns. Pearce and Noble identify that reifying relationships allows the participating classes to be completely oblivious to the relationship [20]. For example, if a student doesn't reference the attends relationships then the student can be reused in another system which doesn't have an attends relationship. There is a counter argument to this suggestion though: in some cases the student is defined by the attends relationship. In fact, the student can be considered to be a person in the *role* of a student when they participate in the attends relationship. In such a case it does not make sense to talk about the student without the relationship. Østerbye suggests that this dichotomy can be resolved by providing two different ways to access relationships: if the participant should be oblivious to the relationship then the relationship should be accessed directly, whereas if the relationship defines the participant (as is the case with student) then the system should support direct access to the relationship from the student [19].

There are arguments for and against the complete separation of concerns for relationships and it may well be that this is something the system's user should decide. However, the principles of separation of concerns are well established and the advantages well recognised, so a relationship abstraction should certainly address the issue.

3.2 Implementation Issues

The issues in this section are predominantly concerned with the expression of a relationship system, rather than its conceptual model.

3.2.1 Uniqueness

A relationship system which provides *link uniqueness* must ensure that a link occurs only once in a relationship instance (subject to some definition of identity for links). For example, a student 'Anna' who is enrolled in 'ACCY101' may not be allowed to re-enrol in the same course. If a system enforces uniqueness then relationship instances may not contain duplicate links as in this example. Consequently, relationship instances in a system with uniqueness are sets.

A system may maintain link uniqueness by a variety of means: duplicate link creation can throw an exception, fail silently, or return an alias to the existing link. Similarly, systems which do not preserve uniqueness may retrieve links in different ways. For example, the program may request the courses a student attends. If there are duplicate links then it is not necessarily clear whether the system should return a set of the courses, or a list containing duplicates. These and other issues, while resolvable, affect the behaviour of the system from the developer's perspective. This means the choices made to implement a relationship system need to be consistent and apparent from the system's abstraction. There is no general consensus in the literature on what the correct choices are.

The *Entity-Relation* model (§2.1.1) requires that tuples are uniquely identified by their participants [6] and Rumbaugh requires the same property, observing that "*The value of a relationship is a set, so adding an element that already exists does not change its value*" (Rumbaugh [21]). UML does not require uniqueness, but notes that if links are not unique then "*links [must] carry an additional identifier appart from their end value*" (UML Specification [17]).

Bierman and Wren's specialisation model requires that links are unique:

> Invariant 2. For every relationship r and pair of objects o_1 and o_2, there is at most one instance of r between o_1 and o_2 (Bierman and Wren [4]).

Østerbye disagrees with this, claiming that "*choosing between the two approaches is better done by the library user than the library designer*" (Østerbye [19]). This is similar to the approach of Pearce and Noble, who provide abstract relationships with both unique and duplicate links in their Relationship Aspect Library [20].

Vaziri *et al.* use relations to introduce instances which depend on other instances for identity. In their system a relation (link) cannot be duplicated as this would break the identity requirements their system emposes. Systems which do allow duplicates [20, 19, 17] do not specify how links should be uniquely distinguished when duplicates may occur, but this is not unusual for object-oriented systems which generally rely on memory location as the primary identifier for objects.

Systems with unique links are not necessarily less expressive than systems with duplicate links: Relationships composed of unique links correspond to (strict) graphs whereas relationships with duplicates correspond to multi-graphs. Multi-graphs can always be reduced to graphs by splitting all duplicate edges with a new node which allows them to be distinguished (this is similar to UML requiring that duplicate edges are distinguishable [17]). This principle applies to relationships as well: to reduce a relationship with duplicate links to a relationship with unique links simply add another participant which allows the links to be distinguished. This solution is effective, but it may not make sense for the relationship abstraction. It is not clear whether links should be unique, but this is something which must be addressed by a relationship system.

3.2.2 Navigability

UML allows association ends to be annotated with navigability recommendations (§2.1.4). Navigability annotations allow system designers to specify when relationships should be efficiently navigable. This is typically implemented manually in mainstream object-oriented programming

languages but a relationship abstraction should allow similar functionality so that programs using first-class relationships do not suffer a performance hit.

Pearce and Noble's library of relationship aspects provide different relationship representations which have different runtime performance characteristics [20]. These directly affect the runtime performance characteristics of the system and the programmer can select and change implementations without significant re-factoring, thanks to the strong abstraction boundary imposed by the library. Similarly, Østerbye allows different relationship implementations to be used which give different performance characteristics, though there is more re-factoring required to change implementations; the library does not entirely separate representation and implementation. None of the other related work provides a means for changing implementations for navigability concerns.

There are other UML annotations such as aggregation and association end ownership which are probably important considerations for an object-oriented practitioner. In some cases these may become irrelevant in a first-class relationship system, however these should certainly be taken into consideration when designing a relationship abstraction.

3.2.3 Operation Propagation

Rumbaugh proposed that relationships could be used to manage the propagation of operations such as clone, dispose and serialise [22]. Traditionally programming languages support shallow (single object) and occasionally deep (all object) versions of these operations but programmers must implement *sheep* operations (operations which propagate as far as some abstract boundary) manually. More recently languages such as Java have added support for annotations which allow programmers to specify object properties such as *serialisable* which allow languages to do a better job of propagated operations like serialisation. If first-class relationships are

introduced to object-oriented programs then there is an opportunity for more complex and subtle operations to be implemented as meta-level operations on the extent of the relationship instance. These are not supported by existing relationship implementations.

3.2.4 Serialisation

It is increasingly common for modern programs to require persistent state and be able to migrate objects and groups of objects between different running applications. Most modern object-oriented languages provide some form of serialisation (or pickling) of objects for this purpose, allowing objects to be stored, transported and retrieved (for interacting with a database, for example). One of the particular challenges in this domain is re-linking object systems which maintain relationships with references, as circular references are difficult to preserve. It is possible that relationship support might simplify this issue as the relationship can encapsulate the references independently of the objects.

Serialisation is not covered by any of the work directly related to relationships in object-oriented programs, however it is increasingly common for object-oriented programming languages to communicate with relational databases for persistent storage. Systems like *Ruby on Rails* which facilitate this are becoming increasingly popular [29]. There are numerous projects working towards better integration between object-oriented programs and relational databases, and first-class relationships may well be able to play a role in improving this integration.

3.2.5 Querying

Relational databases provide excellent support for creating and performing queries across large data-sets to retrieve subsets matching specific criteria. Historically there has been little support for declarative queries in object-oriented languages but there have been several attempts recently to

add this feature [32, 11]. It would be interesting to consider using queries to construct dependent relationships in a relationship system. This would further increase the similarities between relationships in object-oriented languages and relational databases, potentially allowing object-oriented practitioners and language designers to use the wealth of literature which exists in the database community to improve object-oriented systems.

3.2.6 Roles

Chen described the *role* of an entity in a relationship as "*the function it performs in that relationship*" (Chen [6]). Thus, a person which is attending a course is called a *student*. As we discussed in §2.3.1, Østerbye identifies that there are two different ways of accessing a relationship: from the participants (*role-based*) or via a distinct entity (*association-based*) [18], and demonstrated that the access to a relationship can be independent of the representation.

The access to the relationship is important because it affects potential polymorphism of the system. A system which accesses a relationship via roles might find it difficult to provide polymorphism for the participants as they are strongly coupled with code using the relationship. The same system would easily be able to provide relationship polymorphism however because the relationship is not referenced directly. On the other hand, if the access to the relationship is association-based then there is much more freedom for polymorphism of the participants, but stronger coupling between the relationship and code using it. Bierman and Wren *et al.* provide role-based access to relationships [4] whereas Pearce and Noble provide association-based access [20]. Østerbye allows the programmer to choose whether to provide role-based access and association-based access is always available [19].

Roles also provide an interesting question regarding where relationship-related behaviour should be implemented: in most object-oriented pro-

grams relationship behaviour is implemented predominantly within the relationship's clients, however there are important relationship applications where this is not the case. For example graphs, maps, and other large relationship-based structures. If relationship behaviour is implemented within client then the client is tied to the relationship: it cannot be reused independently of the particular relationship implementation. It is also possible to implement the relationship-related behaviour within the relationship but this can increase coupling in the other direction. For example, some relationship-related behaviour needs to store state related to the client. Balzer *et al.* suggest one solution to this: add relationship specific attributes to the client using member interposition (§2.4.2) [2]. Another solution would be to write all of the code relevant to the participants within the relationship, injecting not only attributes but also methods into the client, potentially changing its behaviour within the context of the relationship. This could vastly increase the potential for polymorphism and reuse of both relationship and participant, but there are significant issues to resolve. In particular, this would change the type characteristics of the system: within the context of the relationship an object would have one type, and outside another. Roles differ from standard specialisation because objects which make use of standard specialisation have fixed types, in role-based systems the object can change its type based on the context of the system. There is some work which discusses context-dependent systems, but not with regard to relationships and roles [27].

Chapter 4

An Model for Relationships in Object-Oriented Languages

Chapter 3 discussed a list of requirements for a relationship system and identified where other work falls short of the requirements. This chapter proposes a new relationship system which can satisfy those requirements.

The first section of this chapter introduces our new system. The second discusses the abstraction in the context of the requirements identified in Chapter 3, and the third discusses the techniques for validating this work.

4.1 Relationship Model

Traditional object-oriented systems consist of groups of interacting objects. These objects are modelled by classes, which describe the characteristics of a group of objects at the meta-level (M1-Layer of the Meta-Object Facility (MOF), where objects are part of the data layer, or M0-layer [12]). Typical relationship systems introduce relationships or associations at the modelling level (M1-layer) which describe links, or groups of links, at the data level (M0-layer) [21, 18, 4, 20, 2, 31]. We believe that these systems do not adequately describe relationships because they either fail to consider groups of links at the data level or focus on groups of links (relationships)

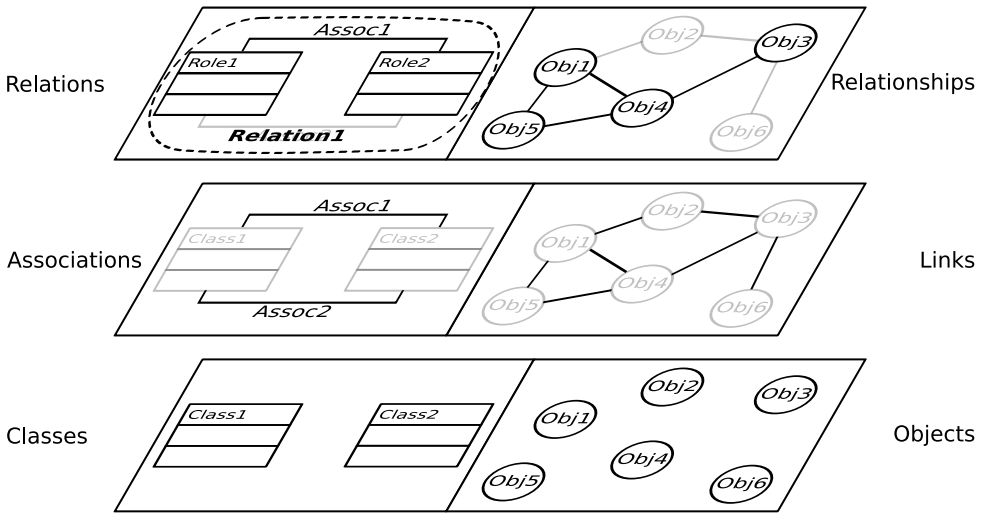


Figure 4.1: The three tiers in our relationship system. The bottom tier (object tier) describes the objects in the system. The middle (link tier) adds the links between objects, and the top (relationship tier) describes properties of groups of objects and links, including roles, relationship constraints, and collections. The three tiers are discussed in §4.1

at the expense of the individual link at the modelling level. This introduces confusion in the literature because it is not clear whether a relationship is a link or a group of links, and whether the term can be used at the modelling or the data level, or both.

We believe that by modelling objects, links, and relationships separately as classes, associations and relations, the resulting system will be easier to understand and consequently easier for programmers to use. To clarify the distinctions between the modelling elements in our system we introduce a three tiered system for describing programs. This section discusses the three tiers, which are shown in Figure 4.1.

Figure 4.2: Person and Course classes without reference to any other objects. There is very little useful functionality that can be expressed without reference to other parts of the system. Although our final system will relate students and courses, in the object tier it makes sense to declare a `Person` class instead of a `Student` class because people can be reused outside of the university system. Conversely, courses do not exist outside the university system so we have declared a `Course` class. Note however, that the attributes associated with the Course class are in no way related to or dependent on the presence of students in the system.

4.1.1 Object Tier

The object tier consists of objects (modelled by classes), similar to the objects defined in traditional object-oriented models. Objects may have state and behaviour, but they may not communicate with other objects on this tier unless there is a strong composition relationship between them. Figure 4.2 describes two classes, Person and Course, in UML without reference to other objects.

The UML description of the Person class in Figure 4.2 might be encoded in Java like this:

```
class Person {
    String name;
}
```

Note that the student references the class *String*. Strings in Java are objects, but because they are immutable it is safe to assume that there is strong composition between Person objects and the String object representing their names (no other class can modify the value of a Person's name because Strings are immutable).

Objects defined on the Object Tier may contain state and behaviour just like traditional objects, however we require that they only reference objects which they contain so that there are no relationships at this level more complex than composition.

4.1.2 Link Tier

The next tier of our relationship system builds on the objects and classes defined in the object tier by adding links between objects described as associations between classes. An association between classes in this system indicates that the classes are related in some way. For example, an association between the classes Person and Course in Figure 4.2 indicates that people may be linked with courses. Figure 4.3 shows the classes with an association between them.

Links in our system are very basic constructs: they are essentially tuples containing one instance (object) from every class in their association. The associations may be named, but this is merely for convenience and has no semantic meaning: a link between a person and a course has the same identity whether it is called "Attends" or "Fails". Links may not have any state or behaviour associated with them. This definition of links and associations removes a lot of the complexity related to associations in UML. For example, there is no distinction between sub-typing and subsetting of associations in this system: an association between subtypes is naturally a subset of the related association between super-types because all possible tuples in the subtype association are also tuples in the super-type association (see Figure 4.3). UML does not have this property; association sub-

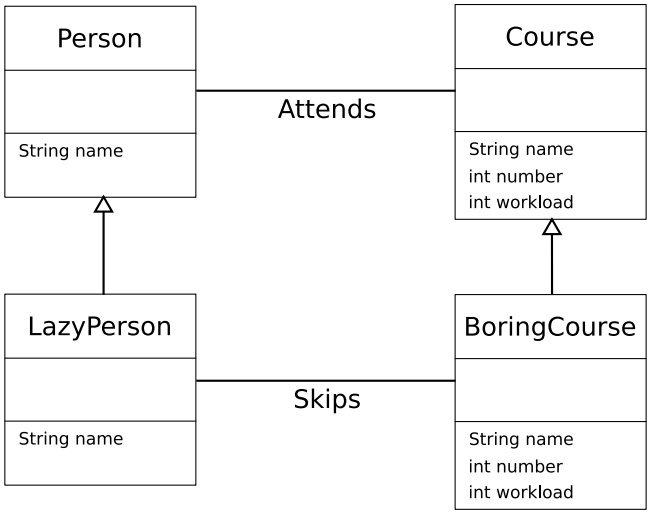


Figure 4.3: Person and Course classes, this time with an association (*Attends*) between them. Note that the name is purely for convenience when referring to the association: it has no effect on the semantic behaviour of the system because associations in our system are entirely defined by the types that they relate. The subclasses of Person and Course, LazyPerson and HardCourse respectively, have an assocation between them also. This association is a subset of the *Attends* association because *Attends* defines all possible links between Students and Courses, and the set of possible links between LazyStudents and BoringCourses (*Skips*) is a subset of these.

typing and subsetting are distinct properties because there is no runtime connection between sub-typed associations [17].

We define an association as the cross product of two types, and links as an element of this cross-product:

DEFINITION 1 (Links and Associations). *Support we have classes $X_0...X_n$, where an object x has type X iff $x \in X$. The association A been $X_0...X_n$ is the cross-product of $X_0...X_n$, thus $A = X_0 \times ... \times X_n$. A link a is a tuple consisting of a pair of objects $(x_0, ..., x_n)$ where $x_0 \in X_0 \wedge ... \wedge x_n \in X_n$. Thus, $a \in A$.*

4.1.3 Relationship Tier

Once the system has associations between objects (links in our system) it becomes possible to identify sets of objects which are associated with each other by sets of links. For example, we can describe the notion of people attending courses as a subset of people and a subset of courses which are linked by some subset of the possible links between the set of people and the set of courses. This leads us to the definition of relationships in our system:

DEFINITION 2 (Relationships and Relations). *Suppose we have classes of objects $X_0, ..., X_n$ which are associated by an association A where $A = X_0 \times ... \times X_n$. A Relationship r is a subset of A, and the Relation R is the set of all possible relationships r, thus $r \subseteq X_0 \times ... \times X_n$ and $R = \mathcal{P}(X_0 \times ... \times X_n)$.*

Unlike links, for which names are irrelevant so long as the types they relate match, relationships have a type defined by their relation in the same way as objects have a type defined by their class. For example, there may be two relationships between People and Courses, one of type "Attends" and the other of type "Passed" which contain the same links but are not equivalent because they have incompatible type. There may even be two instances of type "Attends" which contain the same links and are not equivalent: relationships have independent identity in the same way as objects.

Roles

In addition to defining relationships between objects, relations may define roles. Roles are like mixins; they define state and behaviour for objects separate from the object's main class which is added to the object at runtime. The role is associated both with the class of objects (e.g. Person) and the relation of relationships (e.g. Attends) and adds functionality to the objects which participate in the relationship at runtime. For example, a person who is attending courses is a student. This does not subsume the identity of the person — the person may exist in other relationships which are unrelated to their role as a student. Rather, the role of a person as a student adds more functionality to the person: in the context of the person as a student they may have a student id, enrolment details, and functionality for calculating fees and checking their workload. These details are irrelevant outside of the student context and they do not affect the identity of the person however, so it does not make sense to include them in the standard definition of a student.

Just as an object may have a role, a link may have a role also. Relations may define a mixin for links which adds state and behaviour to the link which is associated with the Relation. For example, a link between a student and a course which is in an "Attends" relationship may have an associated field which records the number of lectures the student has attended. This piece of information is associated with both the student and the course as a pair, so it cannot be stored in either one. It is also associated with the relationship, so it should not be stored in the link itself. By adding a role (mixin) for the link the information can be added to links which are in an "Attends" relationship only.

In addition to state and behaviour associated with objects and links via roles, there may be additional state and behaviour associated with the relationship. For example, an "Attends" relationship for 2008 may have an associated "year" field, and functionality for mailing fees invoices to all of the students enrolled that year.

The three levels of relationship abstraction which we identify are typically collapsed into a single level in the object-oriented paradigm. This causes interwoven, fragile code which is confusing to create and to maintain. The dependence of some object state and behaviour on relationships is seldom recognised because object-oriented programming languages obscure relationship concerns by describing their implementation rather than their intention. Even languages which do address relationships as separate entities fail to separate the behaviour associated with relationships from the classes [21, 4], or lose the association between the state and behaviour, and the objects with which it is associated [15, 20]. We believe that our relationship system simplifies the categorisation of behaviour as object, link, or relationship (role) behaviour and provides a more natural way to describe the related behaviour than Aspect-Oriented programming [10] or subjective objects [27]. The next section describes our proposed implementation for this system.

Connections between Objects, Links, and Roles

It is interesting to consider the connections between objects, links, and roles. Links are defined in the tier above objects, but objects have independent identity whereas links do not. Roles and objects have fields and methods whereas links do not. In fact, we believe that objects are actually roles for special links: if we create a link with one participant: a memory address, we have the core identity of an object. From there, we can define a role for that link which adds the state and behaviour we expect from objects. In this way, objects become redundant, merely sugar for an underlying mechanism dependent on links and roles.

4.2 Implementing the Relationship System

The last section discussed our three-tier abstraction for a relationship system. For this system to be useful it needs to be taken from abstraction to realisation in a manner which maintains the consistency of the system while allowing programmers access to familiar concepts from the object-oriented paradigm.

We propose that the three tiers identified in the previous section should be explicitly implemented in the language: the first layer describes objects and classes, the second describes links and associations, and the third describes the behaviour of the system in terms of relationships and roles, expressed as relationships. Each layer consists of M0- and M1-layer components in UML parlance, with classes defining objects, associations defining links, and relations defining both relationships and roles.

The rest of this section presents the core components of our language in a partial grammar similar to that used in Featherweight Java [9] and RelJ [4]. The grammar is not intended to be complete, and we do not provide formal semantics for the language at this point: instead we give an informal description of their intentional semantics.

4.2.1 Objects

The first layer of the system describes object classes in a similar manner to Java:

$$
\begin{array}{rcl}
c \in \text{Class} & ::= & \textbf{class}\ c\ [\textbf{extends}\ c']\ \{ \\
& & \quad \text{Field}* \\
& & \quad \text{Method}* \\
& & \}
\end{array}
$$

Unlike standard Java fields, fields in our system represent composition relations; this means that an object stored in a field of an object is tied to that object for its lifetime. These are different from value types however

because they may still maintain their own state. We believe that it will be possible to enforce the composition constraints using ownership. This will encourage programmers to create good composition hierarchies and preserve a strong hierarchical structure for the system as a whole. If a programmer wants a more symmetric relationship than composition then that should be declared at a later layer.

4.2.2 Links

Links can be declared as associations in a similar manner to objects declared as classes. An association declaration is written:

$$a \in \text{Association} \quad ::= \quad \textbf{association}\ a\ \{ \\ (\textbf{participant}\ t\ \text{p})+ \\ \}$$

This definition of associations is similar to Vaziri *et al.*'s *relation type* declarations [31] except that the term *key* is replaced with *participant*, and associations may not define state and behaviour (this is added at a later stage).

Associations do not have inheritance (they are defined by the types they relate).

An association declaration between People and Course would look like this:

```
association Enrolled {
    participant Person attendee;
    participant Course attended;
}
```

The association construct in our system is intended to embody the associations discussed in UML [17]: a semantic connector between objects representing interaction or association between them. Unlike UML associations, a link has no identity independent of the participants it associates.

We feel that independent link identity weakens the link abstraction, making links more like objects. Links are valuable precisely because their identity is entirely dependent on the objects they associate.

We do not allow associations to have state or behaviour associated with them, and the participants of a link behave like final fields in Java. This means that two different associations declared with the same participants are equivalent and can exchange links; the name of the association is merely a convenience for the programmer. For the same reasons, there is no value in having association inheritance. There is no typing relation independent of the types of the participants, so the type-relations between associations can always be determined by examining the types of the association participants.

4.2.3 Relationships and Roles

Relationships are collections of links. A relationship may extend another relationship, and it nominates the type of the links which it contains, either by reference to an association or by direct reference to participant types. Relationships are intended to represent the extent set of an association. The closest analogy to this in a traditional object-oriented system would be access to the set of objects which implement a class. This would not be a good analogy however because there can only be one instance of this set, whereas there can be multiple relationships which use a given association and have different sets of links. A better comparison for relationships might be to a link factory: the relationship is responsible for creating and maintaining links between objects. It can provide access to them and add state and behaviour through roles. The distinction here would be that the relationships share the links which they create with the same participating objects.

Separating links from relationships provides a clear boundary between relationships and links at a conceptual level, but also provides a basis for

relationships to be able to perform set operations with other relationships of differing types but the same underlying association.

Relationships are defined in Relation constructs, which also contain role definitions. As we discussed in the previous section, a role is essentially a runtime mixin used to add state and behaviour which is associated with a particular relationship with an existing object or link. A relation declaration is written:

$$
\begin{array}{rcl}
r \in \text{Relation} & ::= & \textbf{relation}\ r\ [\textbf{extends}\ r']\ \textbf{contains}\ a\ \{ \\
& & \quad (\textbf{role}\ p\ \textbf{as}\ \text{T}\ \{\text{Field} * \text{Method}*\})* \\
& & \quad (\textbf{role}\ a\ \textbf{as}\ \text{T}\ \{\text{Field} * \text{Method}*\})? \\
& & \quad \text{Field}* \\
& & \quad \text{Method}* \\
& & \}
\end{array}
$$

where T is a type name for a role.

The relation has a name which corresponds to the type name of its relationships. Unlike associations, this is a proper type name; two relationships which are otherwise identical will be distinct if they have different type names. This means that inheritance makes sense in the system. We define inheritance as a sub-typing relation rather than a set operation, if a subtype relationship contains a link then its super-type relationship must also and vice-versa. This is possible because inherited relations must be declared on the same association type: there is no inheritance between associations to complicate this.

The final part of the relation declaration header is the association which the relation contains. The relation must be defined on a single association which is a reference for the type of the links which the relations' relationships may contain.

The body of a relation declaration is similar to that of a class: it defines the fields and methods which will be present in the relationships the relation defines. In addition, the relation may define roles for its participants

and its association. These roles add fields and methods to any objects or links which are contained in a relationship so that the relationship can access them. They will also be externally accessible in some cases, although we have not finalised how this works.

An example relation which uses the ”Enrolled” association would look like this:

```
relation Attends contains Enrolled {
    role attendee as Student {
        int studentId;
        boolean feesPaid;
    }
    role Enrolled as Attending {
        char grade;
        int lecturesAttended;
    }
    public Set<Student> getUnpaidStudents() { ... }
}
```

The relation ”Attends” contains ”Enrolled” links (Person, Course tuples) which are augmented with two fields: grade and lecturesAttended when they are present in a relationship. The fields are unique to the relationship, so two relationship instances of ”Attends” can store different values in ”grade” for the same link.

The relation also defines a role for Person objects called ”Student”: this augments them with a student id field and a boolean indicating whether or not they have paid their fees. Again, the fields are local to the relationship so two instances of ”Attends” can have different values in their students' fields, even if the students are present in both relationships.

In addition to the roles defined, the relation defines a method called *getUnpaidStudents()* which returns a set of students who have not paid their fees. Note that within the context of the relation the attendees have type *Student*, whereas they are actually of type *Person*. For the fields of

Student to be accessed the context must be aware that the student role is present, and it must know which relationship the role is for.

4.2.4 Comparison With Other Work

Most other relationship work declares the participants of a relationship as part of the signature of the relationship, either as a tuple [4], or as generic parameters to the relationship [20, 19]. This mingles the definitions of links and relationships which leads to confusion when state is added: it is not clear whether the state is associated with the link or the relationship.

By separating the definitions of associations and relationships, and adding related state with roles, it becomes clear which properties belong to links, which to objects and which to relationships. Declaring association participants within the association the declaration can be broken into several lines, avoiding the tendency for declarations to run on which has become the hallmark of extensions to Java. In addition, the participants are given relevant role names (attended, attendee) which correspond to the role names in UML and can be referred to within the context of the relationship and the association's roles. This avoids possible confusion caused by using role names like *from, to* to refer to participants where it may not be clear which participant is being referred to [20, 19]. It is also simplifies the syntax for relationships of greater arity than 2.

Role declarations capture properties which are associated with a relationship participant (object) but depend on the relationship also. Role properties are dependent on both the relationship and the object/link: if either is missing the role cannot be present. A single object may have many independent roles. These may independently declare conflicting properties but these do not map to the same variables, as any properties of the role are encapsulated within the scope formed by the relationship and the object/link together.

An object (or link) which has roles maintains the same static type, but

in the context of a relation it can be referred to by its role type. For example, a person is always a person, but sometimes they may be considered a student (when they are at university). Just as a person may be a student of various institutions independently, a person object may have several independent *Student* roles.

One consideration when constructing relationship systems is whether an attribute should be declared in a role or as a property of the object itself. If the attribute needs to be accessible to multiple relations then it may make sense for it to be moved into the object. It is also possible, however, that the relations are related, in which case it may be better to move that functionality into a base relation which both relations then extend. It is likely that further experience with roles will give clearer insights into this aspect of roles.

4.2.5 Language Extensions and Libraries

There have been several attempts to implement relationships with libraries rather than language extensions [20, 19]. Library implementations have an advantage over language extensions because the library can be added to existing code without modification of code which does not require the features, and library development can be performed independently from language development. Existing work suggests that many relationship features can be implemented within the confines of existing languages (as libraries), but there are also limitations apparent which can be resolved with language extensions. We believe that there are probably certain key features of relationships which are best implemented with language extensions. It may be that these already exist, for example Vaziri *et al.*'s relations dependent identity (see §2.4.3), or they may be discovered in the course of this work, or after its completion. Regardless, it is not clear from the outset which features will ultimately prove to be important so we will add the language features we deem necessary to implement the function-

ality which we would like, but use libraries where it is clear that they are sufficient. If it becomes clear with experience in relationship system development that some features are better implemented as libraries, then we will attempt to do so.

There are no doubt many issues with the language which we have proposed. We believe, however, that the ideas presented are novel and interesting; worthy of further discussion. It is unlikely that the final language we present will have identical features or syntax to this one, but this language presents a starting point for further research in this area.

Chapter 5

Validation

To ensure that our abstraction does satisfy the requirements we have identified in Chapter 3 we will design a language which implements our abstraction. We will present a grammar for the language, provide typing rules and semantics, and we will provide proofs of type safety, soundness and correctness. At this point we will be able to demonstrate whether the language satisfies the theoretical requirements presented in §3.1.

In addition to a formal verification of the language we will provide a practical implementation of the language. This will highlight any practical problems with the language which are not apparent from the theoretical model. A practical implementation will consist of a compiler and a library system which will allow programs to be written in our language.

Once we have completed the practical implementation of the language we will conduct various case studies; implementing existing systems in our language to demonstrate that it satisfies the practical requirements in Chapter 3. If possible, we will invite other programmers to use the language and provide feedback on their experience.

The rest of this section describes the methodology we will use to provide the three types of validation of our abstraction: theoretical description, practical implementation and experience.

5.1 Theoretical Verification

The first step towards a theoretical verification will consist of a formal description of the language. The language will be based on Java, so we will begin with a core fragment of Java such as Featherwieght Java [9] and extend it with the features which we require for relationships. We will present a grammar for our language which describes its syntax and provide a type system and a semantic model for the language. This will provide a formal description of the behaviour of programs. There are various types of semantics which we could use.

Once we have a formal description of the language we will provide proofs of soundness and correctness for the language. Soundness consists of proving type preservation and progress; guaranteeing that any well-formed expression in the language can be executed and will reach a well-defined error state or produce a value with the same type as the original expression. Correctness guarantees that a program which terminates will do so in a state which is correct with respect to the specification of the language. Together, these properties ensure that any valid program can be executed and produce correct output.

5.2 Practical Verification

We will provide a practical verification of the language by implementing a compiler for the language. We will compile the language to Java bytecode so that we can make use of the existing Java Virtual Machines. Our research group already has an extensible Java compiler which we can extend to support our language, avoiding having to implement a complete compiler from scratch. In addition, we will implement a set of libraries for the language which will provide essential features to make the language useful. Finally, we will attempt to provide an abstraction layer for either using or translating existing Java libraries for use within relationship-oriented

code so that existing libraries do not have to be re-implemented. This has been achieved by several other research languages which compile to Java byte-code.

5.3 Case Studies

A compiler for the language and libraries will allow us to run programs in the language. We will choose a suitable set of open-source programs written in existing object-oriented languages to re-implement in our language. This will provide valuable practical experience which we can feed back into the language design and, ultimately, determine whether we have satisfied the requirements for our abstraction detailed in Chapter 3. Ideally, we will be able demonstrate that languages with relationships do indeed provide substantial benefits for programmers over traditional object-oriented programming languages but this is probably beyond the scope of this PhD.

We will attempt to discern whether, and to what extent relationship implementations impose performance penalties over existing implementations, whether the relationship-oriented code is more modular (smaller files, smaller methods) and whether relationship-related code achieves better separation of concerns by comparing our case studies with implementations in existing languages.

Chapter 6

Work Plan

6.1 Contributions

The research described in this proposal represents several contributions to the field of object-oriented programming and methodology:

- A model for relationships object-oriented programming languages.
- A formal description of a language which implements the model as a proof of concept.
- Standard proofs of type-safety, soundness, and correctness for the language.
- An implementation of the language, and case studies verifying its contribution.

Together these contributions will demonstrate the benefits of using first-class relationships to object-oriented languages and provide a new model for adding relationships to existing languages which enables these benefits to be obtained.

Task	*Duration*	*Completion Date*
Language Design and Prototype	2 months	May 08
Compiler	1 month	June 08
Libraries	2 months	August 08
Formalism and Proofs	5 months	January 09
Case Studies	3 months	April 09
Writing Thesis	6 months	October 09

Figure 6.1: Plan for completion

6.2 Plan

I expect to complete my thesis in three years, from October 2006. I have identified six remaining components which need to be completed, which are itemised in Figure 6.1, along with the expected time requirements for each.

Developing the relationship model and language, which we have completed, are by far the largest components of work because they require careful consideration of existing work to determine what the requirements for a model are. We do not expect that the language we have developed so far will be the final product: this will likely change as a result of experiences writing the language prototype system and with the formal system and implementation. It is, however, sufficient to describe the intended semantics of the important components and provide a good prototype of the final language. It should certainly be possible to take the language we have developed so far and produce a prototype by May this year.

The next three tasks (Compiler, Libraries, Formalism and Proofs) will most likely happen somewhat concurrently as there will be some feedback between developing the compiler and the formal system and proofs. Once these have been completed the case studies can be conducted.

6.3 Thesis Outline

I intend to use the following structure for my thesis:

Introduction

This chapter will outline the motivation for this research in the context of object-oriented methodology. It will introduce relationships and discuss why they would be a valuable addition to object-oriented programming languages.

Background

As in this proposal, the background chapter will introduce the field of object-oriented methodology. It will discuss the disparity between relationships object-oriented modelling and implementation, and the various past attempts to solve it. Chapter 2 of this proposal is an initial version of this chapter of the thesis.

Requirements

This chapter will present the requirements for a successful relationship model. It will discuss the problems with existing relationships in programming languages and outline the requirements for a satisfactory model. This chapter will be derived from Chapter 3 of this proposal.

Relationship Model

This chapter will introduce our model for relationships in object-oriented systems, and demonstrate that this model does indeed address the requirements we have identified. This chapter will detail the final version of the system introduced in Chapter 4 of this proposal.

Relationship Language

This chapter will introduce the Relationship Language which we present. It will give the grammar for the language, its type system and semantics, and demonstrate that this language implements the model from the previous chapter while remaining object-oriented.

Language Proofs

This chapter will present the proofs of type-safety, soundness and completeness for the language.

Implementation

This chapter will describe the implementation of the language. It will discuss the implemented case studies and contrast them with implementations in traditional languages to demonstrate the advantages of using an object-oriented language with relationships over traditional object-oriented languages.

Case Studies

This chapter will describe the case studies conducted using our language.

Conclusion

This chapter will summarise the thesis and its contributions. It will discuss the work presented in the thesis and discuss its advantages and disadvantages. It will also discuss future work.

Bibliography

[1] ARMSTRONG, D. J. The quarks of object-oriented development. *Commun. ACM 49*, 2 (2006), 123–128.

[2] BALZER, S., GROSS, T. R., AND EUGSTER, P. A relational model of object collaborations and its use in reasoning about relationships. In *ECOOP 2007 – Object-Oriented Programming* (2007), E. Ernst, Ed., vol. 4609 of *LNCS*, Springer, pp. 323–346.

[3] BECK, K., AND CUNNINGHAM, W. A laboratory for teaching object oriented thinking. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications* (New York, NY, USA, 1989), ACM, pp. 1–6.

[4] BIERMAN, G. M., AND WREN, A. First-class relationships in an object-oriented language. In *ECOOP* (2005), pp. 262–286.

[5] BOOCH, G. *Object-oriented analysis and design with applications (2nd ed.)*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[6] CHEN, P. P. The entity-relationship model - toward a unified view of data. *ACM Trans. Database Syst. 1*, 1 (March 1976), 9–36.

[7] DAHL, O.-J., AND NYGAARD, K. Simula: an algol-based simulation language. *Commun. ACM 9*, 9 (1966), 671–678.

[8] GAMMA, E., HELM, R., JOHNSON, R. E., AND VLISSIDES, J. M. Design patterns: Abstraction and reuse of object-oriented design. In *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming* (London, UK, 1993), Springer-Verlag, pp. 406–431.

[9] IGARASHI, A., PIERCE, B., AND WADLER, P. Featherweight Java: A minimal core calculus for Java and GJ. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)* (N. Y., 1999), L. Meissner, Ed., vol. 34(10), pp. 132–146.

[10] KICZALES, G., LAMPING, J., MENHDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming*, M. Akşit and S. Matsuoka, Eds., vol. 1241. Springer-Verlag, Berlin, Heidelberg, and New York, 1997, pp. 220–242.

[11] MEIJER, E., BECKMAN, B., AND BIERMAN, G. Linq: reconciling object, relations and xml in the .net framework. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2006), ACM, pp. 706–706.

[12] MODELING, T. U. Uml and mof.

[13] NOBLE, J. Basic relationship patterns, 1997.

[14] NOBLE, J. Roles and relationships. Keynote at Roles and Relationships Workshop (ECOOP07), July 2007.

[15] NOBLE, J., AND GRUNDY, J. Explicit relationships in object oriented development. In *Proceedings of the conference on Technology of Object-Oriented Languages and Systems (Tools)* (1995), Prentice-Hall.

[16] NYGAARD, K. Basic concepts in object oriented programming. In *Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming* (New York, NY, USA, 1986), ACM, pp. 128–132.

[17] OBJECT MANAGEMENT GROUP. *Unified Modelling Language (UML) 2.1.1*, 02 2007.

[18] ØSTERBYE, K. Associations as a language construct. In *TOOLS* (1999).

[19] ØSTERBYE, K. Design of a class library for association relationships. In *LCSD* (2007).

[20] PEARCE, D. J., AND NOBLE, J. Relationship aspects. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development* (New York, NY, USA, 2006), ACM Press, pp. 75–86.

[21] RUMBAUGH, J. Relations as semantic constructs in an object-oriented language. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications* (New York, NY, USA, 1987), ACM Press, pp. 466–481.

[22] RUMBAUGH, J. Controlling propagation of operations using attributes on relations. In *OOPSLA '88: Conference proceedings on Object-oriented programming systems, languages and applications* (New York, NY, USA, 1988), ACM, pp. 285–296.

[23] RUMBAUGH, J., BLAHA, M., PREMERLANI, W., EDDY, F., AND LORENSEN, W. *Object-oriented modeling and design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[24] RUMBAUGH, J., JACOBSON, I., AND BOOCH, G. *The Unified Modelling Language Reference Manual*. Addison-Wesley, 1999.

[25] RUMBAUGH, J. E., BOOCH, G., AND JACOBSON, I. *Unified Modelling Language Specification*. Object Management Group, Framingham, Mass., 1998.

[26] SHAH, A. V., HAMEL, J. H., BORSARI, R. A., AND RUMBAUGH, J. E. Dsm: an object-relationship modeling language. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications* (New York, NY, USA, 1989), ACM, pp. 191–202.

[27] SMITH, R. B., AND UNGAR, D. A simple and unifying approach to subjective objects. *Theor. Pract. Object Syst. 2*, 3 (1996), 161–178.

[28] TAIVALSAARI, A. On the notion of inheritance. *ACM Comput. Surv. 28*, 3 (1996), 438–479.

[29] THOMAS, D., HANSSON, D., BREEDT, L., CLARK, M., DAVIDSON, J. D., GEHTLAND, J., AND SCHWARZ, A. *Agile Web Development with Rails*. Pragmatic Bookshelf, 2006.

[30] UNGAR, D., AND SMITH, R. B. Self: The power of simplicity. In *OOPSLA* (1987), pp. 227–242.

[31] VAZIRI, M., TIP, F., FINK, S., AND DOLBY, J. Declaritive object identity using relation types. In *European Conference on Object Oriented Programming* (2007), pp. 54–78.

[32] WILLIS, D., PEARCE, D. J., AND NOBLE, J. Efficient object querying for java. In *ECOOP 2006* (2006), vol. 4067, pp. 28–49.