Relationship Aspects

David J. Pearce
Computer Science
Victoria University of Wellington, NZ
djp@mcs.vuw.ac.nz

James Noble
Computer Science
Victoria University of Wellington, NZ
kjx@mcs.vuw.ac.nz

ABSTRACT

The relationships between objects in object-oriented programs are as important as the objects themselves. Unfortunately, most object-oriented programming languages provide little support for such relationships, leaving the task of implementing them entirely to the programmer. Relationships are typically hard-coded into the participating classes, resulting in tangled code that unnecessarily couples these classes together. The classes become harder to understand and cannot be reused independently. Aspect-oriented programs can model relationships explicitly, treating them as separate concerns that cross-cut their participants. We show how relationships can be implemented as a library of aspects in Aspect/J. Aspects keep relationships independent of their participants, making the resulting programs easier to read, write and reuse, and as efficient as hand-written code.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; D.2.11 [**Software Engineering**]: Software Architectures—*Languages*

General Terms

Design, Languages

Keywords

Relationships, Associations, Aspect-Oriented Programming

1. INTRODUCTION

As John Donne once famously said: "no man is an island, entire of itself". The same situation exists in our programs — no object stands alone. Indeed, without relationships, most objects are meaningless: what use is a button widget if it cannot be placed into a window for display? or a student which cannot be enrolled in a course? or an album without any tracks? For this reason, relationships are as important as objects in most object-oriented analysis and design methodologies: class or instance diagrams show not only objects (or their defining classes), but also, crucially, the relationships or associations between them, as well as specific attributes of those relationships (e.g. that one window may contain *many* button widgets; that disposing of the window also disposes of its widgets and so on).

Modern object-oriented languages, however, carry relatively few of these relationships through from design into implementation. The exception that proves the rule is, of course, inheritance in its various forms — that one class is a specialisation (or subtype) of another or that it conforms to some interface. Support for this relationship is one of the defining features of the OO paradigm [40]. And yet, more mundane relationships — windows and widgets, dungeons and dragons, students and courses, products and sales, transactions and accounts — do not have this kind of direct support. Rather, their implementations are hand-crafted and smeared across the objects which participate in those relationships. Most OO languages do provide some support here, in the form of collection libraries, but this is fairly rudimentary. For example, a window might have a *vector* of widgets where each widget has a pointer to the window. The code necessary to maintain this relationship must ensure that adding a widget to a window updates the link — but this is outside the scope of an individual collection class and, hence, must be managed explicitly by the programmer.

In this paper, we tackle these problems by modelling relationships as separable, cross-cutting concerns. The definition of a relationship, its attributes, and the code required to implement it are moved out of the participating classes and into a separate *relationship aspect*. In this way, we increase the clarity and cohesion of each participating class, since it now focuses solely on the objects it is modelling, rather than their relationships with other objects. The implementation of the relationship can be improved, removed, or replaced, without affecting the participating classes. Furthermore, the program's coupling is reduced because, although the relationship may be coupled to participants, they themselves are not coupled to the relationship. This allows them to be reused without necessarily bringing along the implementation of the relationship and, in particular, all the other participating classes.

This paper makes the following contributions:

- We demonstrate that Aspect/J can be used to model relationships explicitly, separating them from their participating objects.
- We present an extensible implementation of relationship aspects, called the *Relationship Aspect Library (RAL)*, that provides a diverse, interchangeable, range of relationships using only features found in Java and Aspect/J.
- We provide preliminary performance data showing the Relationship Aspect Library is as efficient as hand-crafted code.

2. RELATIONSHIPS

Figure 1 shows the almost generic diagram of students attending courses. Versions of this diagram are found in many publications on relationships [2, 4, 28, 9]. Many students attend many courses; Courses have a course code, a title string and a teacher; students have numbers and (reluctantly, at least at our university's registry) names. Such relationships (also known as *associations* in UML) can be viewed as sets of pairs $\langle X, Y \rangle$ where $\langle$*Joe*,*COMP205*$\rangle \in$ *Attends* if *Joe* attends the course *COMP205* [33]. Here, *Joe* and *COMP205* are said to be *participants* in the relationship.

Figure 2 shows skeleton implementations of the student and course classes in standard object-oriented style. At the top of each class are the easy and obvious declarations for attributes (and signatures for methods) that provide the central functionality for each class. These are relatively simple, storing the key attributes and doing basic calculations. At the bottom of each declaration we find various pieces of code for explicitly representing and maintaining the relationships between these classes. This includes, for example, data structures to record which student is attending which course (and vice versa), as well as code for enrolling and withdrawing students, whilst ensuring that all data structures remain consistent and correct. But, consider the UML in Figure 1 again: we have three classes, each with a few attributes, and simple straightforward relationships between them. While the classes and attributes transfer directly into our object-oriented language, the relationships do not. Instead, their implementation is spread across the participating classes which are more distant from the original UML as a result. Clearly, there is some asymmetry here.

An inherent problem with the implementation of Figure 2 is that the participating classes are infected with code and data structures implementing the relationships. The classes are much larger, less cohesive, and more complex than they otherwise need to be. Care is needed to ensure both participants keep the relationship consistent (indeed, the code in Figure 2 contains such a mistake in `withdraw`). Changing any details of the relationship (such as storing `attends` information only in `Course`) necessitates changes in both participants. Nor does this approach scale well, as larger systems will see classes participating in more relationships. For example, students could be related to billing, alumni, clubs membership, campus housing, transport concessions — all of them further increasing the complexity of `Student`. The presence of relationship code in `Student` and `Course` reduces the possibility of their reuse. Perhaps we have a curriculum system detailing courses and their prerequisites. Such a system does not care about which students are enrolled in which courses. To reuse `Course` in this system as it is implemented in Figure 2 necessarily means including `Student` (even if not used) and allowing students to be enrolled (which does not make sense in that application).

2.1 Relationship Aspects

To address these issues, we model relationships explicitly as aspects, called *Relationship Aspects*. Aspects are a natural fit for modelling relationships. Conceptually, relationships are not classes, rather they are separate concerns that crosscut several participating classes. Pragmatically, relationship implementations often crosscut their participating classes' code, as illustrated in Figure 2, although they can also be implemented as an additional class (e.g. using a separate `HashMap`). The difference between these two approaches is really an implementation detail — a trade-off for performance over storage, as introducing fields gives faster lookup, but requires space in all instances of the class. Relationship aspects abstract away these details, ensuring code belonging to the relationship is always centralised, regardless of what implementation is chosen.

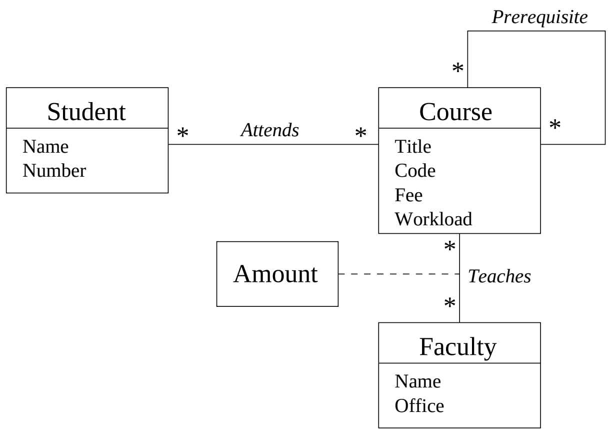

Figure 1: A simple UML diagram, describing students that attend courses.

```
class Student {
 String name;
 Integer number;
 HashSet<Course> attends;

 int totalWorkload() {
  int total = 0;
  for(Course c : attends) {
   total = total + c.workload;
  }
  return total;
}}

class Course {
 String code;
 String title;
 int workload;

 HashSet<Student> attendees;
 HashSet<Course> prerequisites;
 HashMap<Faculty,Amount> teacher;

 void enrol(Student s) {
  if(s.totalWorkload() < 40) {
   attendees.add(s);
   s.attends.add(this);
 }}
 void withdraw(Student s) {
  attendees.remove(s);
}}
```

Figure 2: A typical implementation of Fig. 1. Protection specifiers on fields and accessor methods are omitted for simplicity.

```
Course comp205 = new Course(...);
Student Joe = new Student(...);

comp205.enrol(Joe);
for(Student x : comp205.attendees) {
 System.out.println(x + " is enrolled"); }
```

Figure 3: Illustrating how the implementation of Figure 2 might be used.

```
class Student {
 String name;
 Integer number;
}
class Course {
 String code;
 String title;
 int workload;
}

aspect Attends extends
 SimpleStaticRel<Student,Course> {

 int totalWorkload(Student s) {
  int total = 0;
  for(Course c : from(s)) {
    total = total + c.workload;
  }
  return total;
 }
 void add(Student s, Course c) {
  if(totalWorkload(s) < 40) super.add(s,c);
}}

aspect Teaches extends
 StaticRel<Course,Faculty,Amount>{}

aspect Prerequisites extends
 SimpleStaticReflexiveRel<Course>{}
```

Figure 4: Implementation of Figure 1 with relationship aspects. Again, protection specifiers are omitted for simplicity.

```
Course comp205 = new Course(...);
Student Joe = new Student(...);

Attends.aspectOf().add(Joe,comp205);
for(Student x : Attends.to(comp205))
 System.out.println(x + " is enrolled");
```

Figure 5: Illustrating how the implementation of Figure 4 might be used.

Relationship aspects provide a uniform interface for manipulating the relationship (navigating, adding and removing participants) allowing relationship implementations to be interchanged without affecting client code.

Figures 4 and 5 show the basic idea. First, the extraneous code and data structures required to implement the relationships are removed from `Student` and `Course` — thus, these classes correspond more closely to the UML design. Then, we introduce three relationship aspects which correspond directly to the three relationships *Attends*, *Prerequisites* and *Teaches* in the design of Figure 1.

`Attends` is declared as a simple, static relationship from students to courses. This is a many-to-many relationship, where a student can be enrolled at most once in any given course. The declaration extends a *generic* Aspect/J aspect (`SimpleStaticRel`) from our library, supplying `Student` and `Course` for its generic parameters. Because this aspect implements a "static" relationship, fields and methods implementing the relationship will be embedded directly into the `Student` and `Course` classes, resulting in something almost identical to the hand-coded version of Figure 2 (see section 3 for details). The operations for enrolling, withdrawing and so on have been replaced by standard methods for manipulating relationships, such as `add()` and `remove()`, provided by the relationship aspect. Access to the relationship is via `from()` and `to()` methods, where `from(x)` returns the set of `Course`s attended by `x`. Likewise, the set of `Student`s enrolled in `y` is obtainable with `to(y)`.

We can immediately see from Figure 4 that the advantages of relationship aspects lie in separating out the relationship concerns. In the original implementation of Figure 2, `Student` and `Course` were tightly coupled — one always needed the other. In the relationship aspect implementation, however, this is not the case — `Student` and `Course` are independent. Thus, they can be reused without modification in other applications where *Attends* does not make sense. The code for checking a student is eligible to enrol is now in one place, rather than being spread out over the participants. This makes the program clearer, as this code is relevant only to the relationship, rather than any of the participants. This design is also more flexible as relationships can be added, removed or replaced (by extending different relationship aspects) with ease. Perhaps we need to change the *Attends* relationship to be implemented using a `HashMap`, to allow efficient enumeration of all $\langle Student, Course \rangle$ pairs (e.g. for printing). Making this change to the object-oriented implementation in Figure 2 is quite substantial — code from both `Student` and `Course` must be removed and engineered into a new class; worse still, we could try and retrofit the new requirements around the existing code (perhaps by using more complex data structures in `Course` and `Student`) leading to more tangling of code and coupling to this specific application. With relationship aspects, this change is trivial: we simply change `Attends` to inherit from `SimpleHashRel` and we are done.

The `Teaches` relationship aspect in Figure 4 is implemented as a static relationship (`StaticRel`). This differs from a *simple* relationship in that a third type parameter specifies the class used to represent pairs of objects — and so individual pairs can be objects in their own right. In the example, `Amount` is a pair which, in addition to storing a course and faculty member, stores a value representing the amount of the course taught by the faculty member. In UML terminology, this corresponds to an *association class*. The final relationship declared in Figure 4 is *Prerequisites*. This is a *reflexive* relationship, meaning it connects objects of the same type. As we will see, reflexive relationships also permit a richer set of operations, compared with standard relationships.

In proposing relationship aspects, we are arguing that both relationships and classes should be represented explicitly, and separately, in a program's text. This philosophy is fundamental to the idea of relationship aspects and, we believe, promotes a more reusable, flexible and simpler approach to design and to programming. Of course, one might argue that this is simply a case of over engineering — where time is wasted building in support for future changes that never happen. Code using relationship aspects is simpler than code that implements relationships manually, however, so using relationship aspects from the outset should come at no additional programming cost, whilst still providing greater flexibility.

2.2 Dynamic Relationships

Some relationships do not exist for the entire life of the program. Consider again the prerequisites relationship. This describes a graph structure which is applicable to a large number of graph algorithms, such as depth-first search and transitive closure. We can imagine transitively closing the prerequisite graph to determine the complete set of dependencies for each course. This might be done each year to ensure a given degree can be completed in three years. Once this check has been made, however, the transitive closure relationship can be discarded. Unfortunately, using a static relationship to implement the transitive closure would add fields to the `Course` class. Every course object would contain those fields, consuming storage even when the relationship is not being used.

To support these situations, our library includes *dynamic* relationship aspects. While static relationship aspects represent relationships that are fixed (as typically intended in UML class diagrams), dynamic relationship aspects can be created by the programmer as and when required. For example, we can create a dynamic relationship (perhaps implemented by an external hash table), use it to calculate the transitive closure, and dispose of it when we are done.

3. RELATIONSHIP ASPECT LIBRARY

We have designed and implemented the *Relationship Aspect Library (RAL)* to allow programmers to use relationships aspects. The relationship aspect library contains static and dynamic implementations for a range of relationships. The library is significant because it successfully abstracts quite different implementations *using only features present in Java and Aspect/J*. In this section, we explore the design of the library focusing particularly on the relationship interface provided by RAL, and the way both static and dynamic relationships can implement this interface. The library itself is available for downloading under an open source license from `http://www.mcs.vuw.ac.nz/~djp/RAL/`.

3.1 Relationships

The key concept in the library is the *Relationship* interface, which provides the following main features:

```
interface Relationship<FROM,TO,
     P extends Pair<FROM,TO>> {
 public void add(P);
 public void remove(P);
 public Set<P> toPairs(TO t);
 public Set<P> fromPairs(FROM f);
 public Set<FROM> to(TO t);
 public Set<TO> from(FROM f);
 ...
}
```

This interface represents a relationship as a set of pairs $\langle FROM, TO \rangle$, such that two objects f and t are related if $\langle f, t \rangle$ is in the set. The type parameters `FROM` and `TO` dictate the permissible types of f and t. The third type parameter, `P`, determines the actual type of the individual associations, and must extend `Pair<FROM,TO>`:

```
interface Pair<FROM,TO> {
 public FROM from();
 public TO to();
}
```

The advantage of this third type parameter `P` is that it allows us to treat individual pairs as objects, allowing them to model UML-style association classes. Programmers can add state and behaviour to their own `Pair` subclasses, benefiting from inheritance and polymorphism. For example, to store additional information within the associations (such as *Amount* in Figure 1) we implement a specialised `Pair` class which includes this information.

Many relationships (such as *Attends* in Figure 1) do not need this extra behaviour. We call these *simple* relationships, and provide a `SimpleRelationship` interface that does not require an explicit `Pair` type parameter (we abbreviate `FROM` and `TO` with `F` and `T` for space reasons):

```
interface SimpleRelationship<F,T>
  extends Relationship<F,T,FixedPair<F,T>> {
 public boolean add(F f, T t);
 public boolean remove(F f, T t);
}
```

For simple relationships, pairs are instances of `FixedPair`, which is a `final` class containing just two `final` fields holding the `FROM` and `TO` references (hence, its instances are guaranteed to be immutable). This frees simple relationship implementations from the burden of having to store pairs explicitly — although they must still be generated when either `toPairs()` or `fromPairs()` are called. Thus, simple relationships benefit from leaving the `Pair` implementation up to the particular underlying relationship aspect. For general relationships with user-defined pairs, this is not possible as we expect to get back the actual pair object put in.

Relationships are *bidirectional*. Given a `FROM` instance f, we can get the `TO` instances it is associated with through `from(f)`. Likewise, given a `TO` instance t, we get its `FROM` instances via `to(t)`. Thus, `from()` enables *forward* traversal, while `to()` gives *backward* traversal. In a similar way, `fromPairs(f)` and `toPairs(t)` return the `Pair` objects matching $\langle f, * \rangle$ and $\langle *, t \rangle$ respectively.

3.2 Static Relationships

A static relationship allows the programmer to indicate the relationship will persist for the duration of the program. The idea is that this information can be exploited to obtain a more efficient implementation. The library provides a range of static relationship aspects. Programmers can then implement a relationship which cross-cuts the participants simply by extending a library aspect, such as `StaticRel<FROM,TO,P>`, rather than modifying the participants by hand. We employ Aspect/J's *inter-type declarations* to implement the static relationship aspects. The other main features of Aspect/J (i.e. pointcuts / advice) are not needed for the implementation. The basic static relationship aspect is defined as:

```
aspect StaticRel<F,T,P extends Pair<F,T>>
         implements Relationship<F,T,P> {
 interface Fwd<X> {}
 interface Bwd<X> {}
 declare parents : F implements Fwd<P>;
 declare parents : T implements Bwd<P>;
 HashSet<X> Fwd.fwd = new HashSet<X>();
 HashSet<X> Bwd.bwd = new HashSet<X>();

 public void add(P t) {
  t.from().fwd.add(t);
  t.to().bwd.add(t);
 }
 ...
}
```

`StaticRel` is a generic aspect whose type parameters match those of the `Relationship` interface. The `declares` syntax is used to modify the actual types of `F` and `T` to implement the `Fwd<P>` and `Bwd<P>` interfaces respectively. Then, a field `fwd` is defined for `Fwd<P>`, which has the effect of defining `fwd` for every class that implements `Fwd<P>` (i.e. the class participating in the `FROM` position of the relationship). A similar situation holds for `bwd` and

Bwd<P>. An interesting question is what its means to define a field for some class. In the current Aspect/J compiler (i.e. ajc), it means the field is physically inserted into the class. Thus, we see how extending StaticRel gives some almost identical to a hand-coded implementation where fields are placed directly into the participants (such as attends and attendees from Figure 2). However, it is important to realise the semantics of Aspect/J do not dictate that fields must be physically inserted and, in some environments, this may differ. Nevertheless, we assume the most efficient implementation available for permanently associating a field with a class will be chosen and this corresponds with the idea of a static relationship.

An important point to understand here is that generic aspects in Aspect/J are semantically different from generic classes in Java where type parameters are purely syntactic (due to erasure). This is because the actual values supplied for type parameters determine the effect that extending a generic aspect has. Consider the following:

```
aspect X extends StaticRel<A,B,P> {};
aspect Y extends StaticRel<B,C,P> {};
```

The concrete aspect X results in fields Fwd.fwd and Bwd.bwd being defined for classes A and B respectively. Likewise, Y results in Fwd.fwd and Bwd.bwd being defined for B and C. Thus, the effect of extending StaticRel depends very much on the actual type parameters supplied.

While multiple concrete StaticRels (such as X and Y above) can coexist in the same program, an issue arises when they share the same FROM or TO type. This is because both aspects define a Fwd.fwd (resp. Bwd.bwd) field on the same class. In the current Aspect/J implementation (ajc), this results in a collision. We believe such fields should be distinguished according to the concrete aspect they are instantiated within. Indeed, there are many examples where this behaviour would be valuable, such as the parent-child relationship [1] and the subject-observer pattern [13, 16]. As a workaround, the library provides several copies of StaticRel, named StaticRelX for $X \geq 1$. Thus, when two static relationships share either the FROM or TO position, the situation can be resolved by extending one of these alternative (but otherwise identical) implementations.

Finally, in practice the inter-type declarations for Fwd.fwd and Bwd.bwd are marked private — meaning these fields are visible only to code within the aspect itself. This ensures proper encapsulation by preventing other code from relying on the presence of these fields.

3.3 Dynamic Relationships

Dynamic relationship aspects have fairly straightforward implementations. A good example is HashRel:

```
class HashRel<F,T,P extends Pair<F,T>>
   implements Relationship<FROM,TO,P> {
 private HashMap<F,HashSet<P>> fwd;
 private HashMap<T,HashSet<P>> bwd;
 ...
}
```

Thus we see that, contrasting with StaticRel, this implementation is completely dynamic — neither participants are actually modified (i.e. have fields defined on them) and, instead, the relationship is implemented using two HashMaps for efficient bidirectional access. This means the relationship can be created as

```
// --- polymorphic method ---

void printFrom(Student s,
 SimpleRelationship<Student,Course> r) {

 for(Course c : r.from(s)) {
  System.out.println(s + " attends " + c);
}}

// --- client code ---

aspect Attends extends
 SimpleStaticRel<Student,Course> { }

SimpleRelationship<Student,Course> a =
 Attends.aspectOf();

SimpleRelationship<Student,Course> b =
 new SimpleHashRel<Student,Course>();

Student joe = new Student(...);
Course comp205 = new Course(...);

a.add(joe,comp205);
b.add(joe,comp205);
printFrom(joe,a);
printFrom(joe,b);
```

Figure 6: Static and dynamic relationships can be used interchangeably.

necessary and discarded when no longer needed. Note, the implementation of our dynamic relationship aspects do not use Aspect/J constructs at all. Nevertheless, we regard them as aspects since they replace code which might otherwise be tangled in the participants.

3.4 Relationship Polymorphism

The ability to use dynamic and static relationships polymorphically is an important advantage of our design. Figure 6 illustrates the idea, showing both a static and dynamic relationship being passed to a method that is unaware of the differences in their implementation. A key point is that a static relationship provides a Relationship object through which it can be accessed. This is a singleton object, which makes sense as it corresponds to a static compile-time relationship. Notice that we exploit the fact that Aspect/J implements aspects as singletons (in the normal case, at least), providing access to their instances via aspectOf(). Of course, all this comes at some cost to efficiency as, inside print(), the fields inserted into Student to implement Attends must be accessed indirectly via the singleton object (although sufficient type information is present that these accesses can be inlined by an optimising VM).

With hand-coded implementations, this kind of polymorphism is much harder to achieve as the relationship code is spread across — and tightly coupled with — the participants. Considering again the example in Figure 2 it is quite hard to see how a single method could print both the hardcoded relationship and an external collection without significant extra work.

3.5 Specialised Relationships

The relationship aspect library contains a number of different types of relationship. This is important so that it can offer a realistic alternative to hand-coded relationship implementations, since it must be able to describe many of the diverse types of relationship used in practice. Figure 7 gives an overview of the main interfaces in the library. A curious point is that `Relationship` is not the root of this hierarchy! This reflects the fact that, while we believe `Relationship` captures the most fundamental concept, it is by no means the simplest.

Figure 7 shows the structure of the key interfaces in the library. The library provides a range of implementations for these, supporting both dynamic (e.g. `ReflexiveHashRel`) and static (e.g. `ReflexiveStaticRel`) relationships. Compared to the basic relationship implementations (`StaticRel` and `HashRel`), these interfaces either provide more functionality or enforce more constraints. For example, implementations of `OneToManyRelationship` enforce the invariant that each `TO` participant has at most one `FROM` participant and raises an exception if this invariant is broken. In the same way, implementations of `UndirectedRelationship` represent individual relationships internally in such a way as there is no difference between $\langle x, y\rangle$ and $\langle y, x\rangle$.

A **uni(directional) relationship** represents a relationship which can be traversed only in one direction. This corresponds to a *unidirectional* association in UML and represents a common structure in programs, where only one of the participants has a reference to the other. We can think of this as providing a space optimisation in the case that only one direction of traversal is required. Thus, it differs from `Relationship` only in that neither `to()` nor `toPairs()` are necessary.

A **multi-relationship** may contain identical pairs, meaning it describes a *bag* of pairs rather than a set. This would correspond, in our Student-Course example, to allowing Joe to enrol in COMP205 more than once (presumably to allow him to attempt to sit the exam twice). A multi-relationship implementation can be more efficient that a (set) relationship, because it can store pairs using a simpler and more efficient container (such as `ArrayList`, rather than `HashSet`). If we know that adding identical pairs can never happen in our program (for whatever reason), choosing a multi-relationship implementation can be more efficient. Observe that `Attends` would correspond to a multi-relationship in Figure 2, if it was implemented using an `ArrayList`, rather than a `HashSet`.

A **reflexive relationship** represents the case where *FROM*$=$*TO*. That is, where the association goes between objects of the same type (as for `Prerequisites` in Figure 1). Thus, reflexive relationships correspond to reflexive associations in UML. This case is special because it can describe complex structures such as trees and graphs which permit a richer variety of operations. Reflexive relationships can be traversed using depth-first or breadth-first search and are amenable to a large variety of graph algorithms (such as transitive closure, shortest-paths, cycle detection etc). A subtle point is that reflexive relationships are directed, *not* undirected. This may seem strange, since reflexive relationships provide bidirectional access via `from()` and `to()`. However, it is important to realise that, given a reflexive relationship over some type T and two instances x and y of T, we can still have two distinct tuples $\langle x, y\rangle$ (where x is the *FROM* part) and $\langle y, x\rangle$ (where y is the *FROM* part).

In the sense that reflexive relationships describe *directed* graphs, **undirected relationships** describe *undirected* graphs. Thus, the distinction between the pairs $\langle x, y\rangle$ and $\langle y, x\rangle$ is lost, meaning that if $\langle y, x\rangle$ is present then, by definition, so is $\langle x, y\rangle$. This raises the question as to whether an undirected relationship is a reflexive relationship or vice versa. Certainly, any undirected relationship can be represented as a reflexive relationship, while the converse is not true — a reflexive relationship containing just $\langle x, y\rangle$ is not

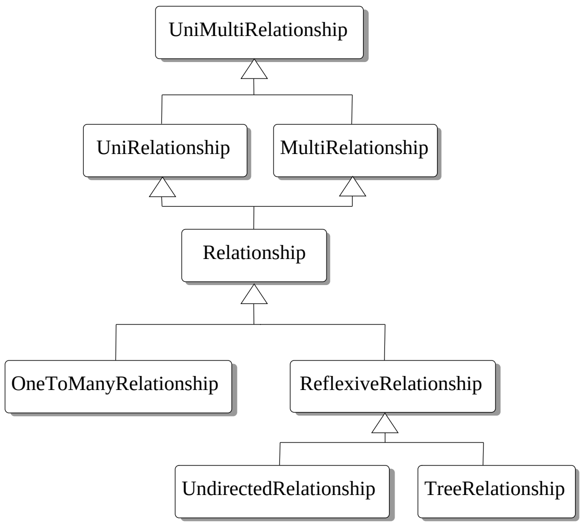

Figure 7: An overview of the interface hierarchy for the Relationship Aspect Library. Since these correspond to Java interfaces, rather than classes, multiple inheritance is permitted. To simplify the diagram, a some inheritance relationships have been left out. For example, a tree relationship *is-a* one-to-many relationship.

undirected since $\langle y, x\rangle$ is missing. The difficulty is that a method expecting a reflexive relationship may be surprised when adding $\langle x, y\rangle$ automatically adds $\langle y, x\rangle$ as well!

A **one-to-many relationship** corresponds to a relationship where each participant has at most one parent. Thus, in UML terminology it is a $\langle 0..1\rangle \longleftrightarrow *$ association. In the case that the relationship is also reflexive, then this constraint describes a *tree* (hence, a one-to-many, reflexive relationship is a **tree relationship**). One-to-many relationships are implemented by suitably simplified versions of the many-to-many relationship aspects.

Finally, a **managed relationship** allows the entire set of associations it represents to be iterated. It is useful to separate out this concept, since many relationship implementations do not naturally support this operation. For example, `StaticRel` is not a managed relationship, because its associations are distributed across participants, but the relationship does not track those participants. The library provides `StaticManagedRel` as an extension of `StaticRel` which also records the set of participants. In this way, all of its associations can be iterated, although this comes at a cost — both in terms of performance and space usage. The library does not provide an interface `ManagedRelationship`; rather, we use the `Set` interface from the Java Collections Library. Thus, `StaticManagedRel` implements `Set<Pair<FROM,TO>>`, but `StaticRel` does not.

These different concepts support different operations. For example, `Relationship` includes the following:

```
interface Relationship<FROM,TO,P>
 extends ... {
 ...
 void clearFrom(FROM X);
 void clearTo(TO X);
}
```

Thus, `clearFrom(x)` clears all pairs with x in the `FROM` position, while `clearTo(x)` clears all those with x in the `TO` position.

In a reflexive relationship, `FROM` = `TO` and, hence, the following is now possible:

```
interface ReflexiveRelationship<T,P>
 extends Relationship<T,T,P> {
 ...
 void clear(T x);
}
```

This clears all pairs where x is in either the `FROM` or `TO` position.

4. PERFORMANCE

We consider that performance is a crucial factor in determining whether relationship aspects are likely to be adopted in practice. Relationship aspects are unlike many other applications of AOP, such as debugging or logging, which can be removed before final program deployment. Rather relationships are crucial to the functionality of the program and, as such, their performance translates directly into program performance. Therefore, we have evaluated the performance of our relationship aspects, and compared them with the hand-coded alternatives. We are not attempting to provide a definitive study — this is beyond the scope of this paper and the space available. Instead, we present some straightforward examples as a proof-of-concept that the performance of relationship aspects is comparable with that of hand-written code.

For each study that follows, we constructed two versions of the test programs: a relationship aspect version and a hand-coded version. The two were entirely separate, although very similar, with the only difference being in the implementation of the relationships. For the hand-coded version, these were implemented by manually adding fields and accessor methods as necessary to the participants. For the relationship aspect version, two relationship implementations were tested: `StaticRel` and `HashRel`. Thus, three implementations were evaluated (i.e. `StaticRel`, `HashRel` and *hand coded*) for each study. The experiments were repeated on three different machines, with different processors and/or operating systems in an effort to see general trends. Details of these are given in Table 1. In each case, Sun's Java 1.5.0 (J2SE 5.0) Runtime Environment and Aspect/J 1.5 (development snapshot 20060125) were used[1]. Note, this development snapshot fixed a number of bugs with the 1.5.0 release of Aspect/J needed to compile the Relationship Aspect Library. Timing was performed using the standard `System.currentTimeMillis()` method, which has a resolution of 10ms or less, depending on the operating system. The complete source for all the code used in each study can be obtained from `http://www.mcs.vuw.ac.nz/~djp/RAL/`.

4.1 Study 1 — Students and Courses

For the first case study, we used the student-course example of Figures 2 + 4. We looked at the cost of enrolling students, iterating the courses attended by each student and the students attending each course. Our benchmark code consisted of creating 20,000 courses and 100,000 students, performing 500,000 random enrolments and running a tight loop which traversed the courses attended by each student, as well as the students enrolled in those courses (hence, testing both directions of traversal), as shown below:

```
for(Student s : students) {
 for(Course c : r.from(s)) {
  for(Student s2 : r.to(c)) {
}}}
```

[1]We were unable to compare against the abc compiler, as this does not currently support Java 1.5 generics.

Name	Machine
Pentium M	Intel Pentium M 1.6Ghz, 1GB RAM, Windows XP
Pentium IV	Intel Pentium IV 2.5Ghz, 1GB RAM, NetBSD v3.99.7
PowerMac G5	64Bit PowerPC 1.8Ghz, 1GB RAM, MAC OS X v10.4.2

Table 1: The three machines used in our experiments.

Here, `r` is the relationship object. For the hand-coded version, we have `s.from()` rather than `r.from(s)`. The entire procedure was repeated 10 times to get an average runtime for each relationship implementation.

4.2 Study 2 — Road Networks

For our second study, we looked at a relationship representing a road network connecting cities together. This describes a graph structure which is both reflexive (as cities are associated with cities) and weighted (as roads have distance). Many interesting algorithms are relevant to this relationship. A representative example is Dijkstra's classic algorithm for determining the *shortest-path* between two points [8]. Our benchmark is an implementation of Dijkstra's algorithm computing the shortest path between every city (known as the all-pairs, shortest paths problem). As input, we used a randomly generated graph with 100 cities and 5000 roads of varying distance. Each relationship implementation was tested on this same input to ensure a fair comparison.

All three relationship implementations used the following implementation of `Pair` to represent roads:

```
class Road implements Pair<City,City>
 private final City from;
 private final City to;
 private int distance;
 ... // accessor methods as needed
}
```

Thus, we have `StaticRel<City,City,Road>` and, likewise, `HashRel<City,City,Road>` for the relationship aspect implementations, while the hand coded version stored two `Road` sets inside each `City` object (one for incoming roads and one for outgoing roads).

4.3 Results

The results are shown in Figures 8 and 9. The main observation is that `StaticRel` is competitive with, but consistently slower than the hand-coded implementation. The majority of this overhead arises from our relationship implementations wrapping the sets returned by `from()` / `to()` / `toPairs()` / `fromPairs()` with immutable adaptors. This prevents clients modifying these sets and potentially putting the relationship in an inconsistent state. These adaptors are created on-the-fly when these methods are called and, hence, some cost is incurred. The hand coded implementations cannot use such adaptors (as `Course.enrol(Student s)` must be able to modify `s.attends`) and, thus, they have a slight advantage.

Figure 9 presents a few surprises. The relative behaviour of the three implementations is remarkably consistent across all three machines. More significant, is that `HashRel` is consistently faster

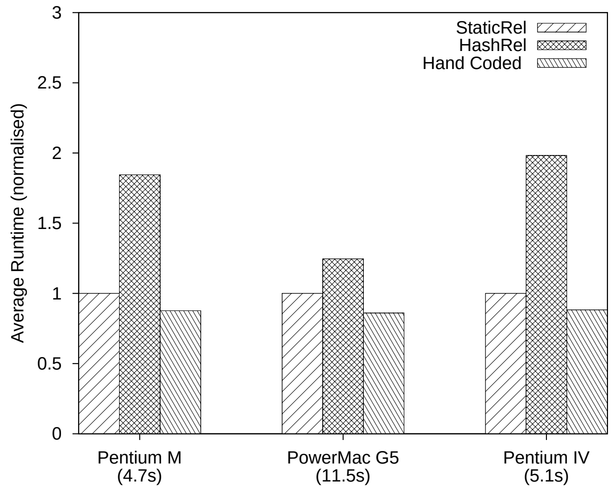

Figure 8: Charts of our experimental data for the student-course benchmark (Study 1). For each machine, the data is given relative to the `StaticRel` implementation, to allow them to be shown on the same chart. Underneath each machine name, the absolute average time (in seconds) for the `StaticRel` implementation is shown for reference.

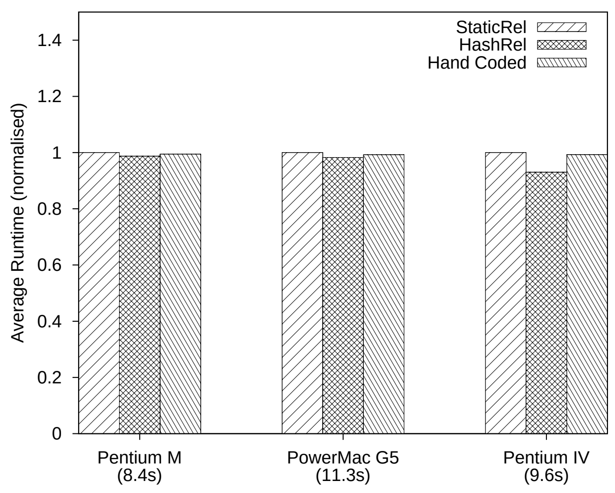

Figure 9: Charts of our experimental data for the road-network benchmark (Study 2). For each machine, the data is given relative to the `StaticRel` implementation, with the absolute average time shown underneath (as for Figure 8).

than the other two (albeit by a small amount). This is strange because `HashRel` must perform a hash table lookup in, for example, `from(x)` to get the set of objects associated with `x`. In contrast, the other two implementations store this set in objects' fields. Thus, they should benefit from a static lookup in `from(x)` (hence, we would expect them to be somewhat faster as in Figure 8). While the exact reasons for the performance of `HashRel` remain unclear, there are some observations to make. The relationships used in the road-network are far smaller than for the student-course benchmark — 100 objects and 5000 pairs, compared with 120,000 objects and 500,000 pairs. The road-network benchmark spends more time traversing the relationship as the algorithm involved is more complex. Thus, it seems plausible that the machine's cache is coming into play here. The idea is that `HashRel` may fit entirely into the cache where the other two do not. This could happen because the shortest paths algorithm does not actually need access to any of the additional state contained in a `City` object. Rather, it needs the reference for a `City` object in order to identify those it is connected to. Thus, if the relationship is spread across the `City` objects, these will be loaded in their entirety into the cache. For `HashRel`, however, the `City` objects are not needed and, hence, they do not occupy precious space in the cache. We must caution, of course, that this is entirely speculative and further work is needed to fully understand the performance of `HashRel`.

In spite of this unexpected behaviour, Figure 9 does highlight an interesting advantage of relationship aspects: different relationship implementations can be quickly and easily tested since the relationship code is properly abstracted. Exactly the same client code is used in our static and dynamic relationship examples: the only difference is the declaration of a `StaticRel` or a `HashRel`. With hand-coded relationships, however, this is not the case: converting between different implementations means making large changes to the code.

5. DISCUSSION

In this section, we discuss various issues in the design and implementation of relationship aspects.

5.1 Encapsulation and Attributes

Encapsulation is an important principle of software design: an object's implementation details should not be visible outside that object. Representing relationships as aspects could be seen as breaking encapsulation, because (especially static) relationships may need direct access to the internals of their participating objects. However, our aspect-oriented design strongly encapsulates the implementation of the relationships themselves. The relationship's clients (including their participating objects) manipulate the relationship via the `Relationship` interface. Participants cannot access any details of a particular implementation, and any fields or methods inserted into participating objects are visible only to the relationship. This is in contrast to most object-oriented relationship designs [25] — especially those that scatter the relationship implementations across multiple classes (see figure 2) — which break objects' encapsulation rather more directly than our aspect-oriented designs.

Reifying each relationship as an aspect also makes it possible to apply visibility constraints to each relationship individually, independently of its participating objects: in Aspect/J, relationship aspects can be encapsulated within their defining package, their defining class, or that class and its subclasses by using the standard Java-style access specifiers. Relationship aspects also make explicit the choice between modelling associations by relationship aspects and by object attributes (and thus fields or collections). Our practice is to use fields or collections when the relationship is completely straightforward and unimportant in itself, that is, when there are no invariants, constraints, or behaviours on the association, and when in UML it would be modelled by an object's attribute. For example, a Student object could have a “name” String field, an “id” Number field, and an “address” field holding a vector of strings; none of these merit being implemented as a relationship. But once the relationship is at all complex, has design invariants, constraints, and behaviours of its own, then it should certainly be implemented explicitly using a relationship aspect. Students attending courses, making payment transactions, and their tenancy in halls of residence, for example, are all sufficiently important or complex to model explicitly and implement with relationship aspects.

5.2 Relationship Reuse

Treating relationships as aspects facilitates code reuse. The common `Relationship` interface makes it easier to reuse code, since they permit relationship implementations to be interchanged without changing the client code. Because Aspect/J allows inheritance between aspects, we obviously permit inheritance among relationship aspects. The Relationship Aspect Library uses inheritance to organise its implementations, and programmers using the library can also employ inheritance to organise their code.

There is one drawback to using inheritance to add common operations to relationship aspects. Aspect/J requires that each concrete relationship aspect extends a generic abstract in the library — as in the definition of `Attends` in Figure 4:

```
aspect Attends extends
  SimpleStaticRel<Student,Course> {

  int totalWorkload(Student s) { }
  ...
}
```

At this point, any code added into the relationship, such as the `totalWorkload()` method is tied to that relationship: it cannot be reused across many different relationship implementations. In Aspect/J, inheritance-based reuse is possible only with abstract aspects, and then only via single inheritance. This can be partially mitigated by making `Attends` abstract, so many concrete relationships can inherit it and reuse `totalWorkload()` — this is, after all, the principle underlying the design of the library. All these concrete relationships, however, are now forced to use the same relationship implementation.

There are a number of alternative designs that avoid this Aspect/J limitation. Using a classically object-oriented approach based on the Decorator pattern [10], we can make a new relationship aspect that contains the extra code, but which delegates the core relationship operations to a second relationship aspect. Any existing relationships accessed via this decorator effectively gain the additional behaviour. Given that Aspect/J is an aspect-oriented language, however, we can place the behaviour we wish to reuse into a separate aspect that mixes it into those relationship aspects needing the extra behaviour.

The issue discussed here is an example of a more general problem identified with reusing Aspects in Aspect/J [23]. The scenario arises when an abstract aspect containing the primary implementation defers its *binding* via an abstract pointcut. A *binding aspect* extending this then provides the concrete pointcut (i.e. the binding). The problem (as above) is that any code in the binding aspect (including the binding itself) cannot be reused with different abstract aspects (perhaps providing alternative, but equivalent implementations). More advanced aspect-oriented languages, such as CaesarJ [23, 22], address this problem directly and would alleviate the issue identified here with our relationship aspects.

5.3 Relationships and Behaviour

This paper has focused exclusively on the structural aspects of modelling relationships, such as the interfaces for adding, removing and navigating pairs. Behavioural aspects have not been considered. That is, the possibility that objects may behave differently when participating in a relationship from when they are not. Consider again the student-course example of Figure 1. In practice, a course will have many more attributes, such as a curriculum, than we have shown. Such attributes will change over time in line with changes to the course. A useful constraint would be to prevent any changes when students are attending the course — it would be unfair if the curriculum changed just before the exam! Thus, `Course` objects behave differently (i.e. they don't accept changes) when they are participating in a relationship from when they are not (i.e. they do accept changes).

Aspect-oriented languages should provide a natural means of expressing these behavioural aspects of relationships. We hope to consider this in more detail in the future and it will be interesting to see whether Aspect/J and the Relationship Aspect Library can fully support these ideas.

5.4 Relationships and Patterns

Using aspects to represent relationships has much in common with using aspects to represent role models and patterns [18, 13, 14]. From our perspective, many important patterns are primarily relationships with added behaviour. The Composite and Interpreter patterns [10], for example, are primarily tree relationships with additional behaviour to distribute operations on an internal node down to the leaf nodes for Composite, or to evaluate expressions for Interpreter; the Observer pattern is a many-to-many relationship between subjects and observers, with additional behaviour to detect changes in the subjects and notify observers; and the Chain of Responsibility pattern is essentially a one-to-many relationship between clients and handlers, with additional behaviour to distribute a client's requests to its handlers.

We expect that programmers will be able to use the Relationship Aspect Library to implement patterns in a number of ways. Patterns could still be implemented directly, in the object-oriented style, but using relationship aspects to support the relationships required by the patterns. Some patterns (in particular Observer) could be implemented by extending relationship aspects to add in the pattern specific behaviour. Most patterns, however, could be implemented as independent aspects in their own right, but again using relationship aspects to provide inter-object relationships.

The main difference between using aspects to implement patterns and to implement relationships is one of focus: the purposes and structures of patterns vary greatly, whereas relationships are essentially about maintaining associations between groups of objects. This common purpose supports our common relationship interface, which allows programmers to exchange implementations of relationships without changing the code that uses the relationships. On the other hand, many of the refactoring techniques that have been developed for replacing object-oriented pattern implementations with aspects [14, 24, 11] should also support refactoring object-oriented programs to use relationship aspects.

5.5 Library Interface Design

Interface design is one of the most important parts of library design. We have designed the `Relationship` interfaces to include all the key operations that are required on relationships. Thus, relationship implementations that do not explicitly store `Pair` objects offer an API that can supply pairs, and generates them as required. Different operations on different implementations will have different performance characteristics: the common interface allows implementations to be changed without affecting the rest of the program.

Our `Relationship` interfaces are not generally compatible with Java's existing Collections interfaces. This is because we consider that relationships are fundamentally different to collections: relationships are aspects, rather than classes; they are typically bidirectional, rather than unidirectional; their implementation often cross-cuts multiple participating classes. Of course, relationships use classes from the collections library both in their interfaces (`Sets` and `Iterators`) and also in their implementations (particularly `HashSet`s and `HashMap`s), and some special cases (ManagedRelationships) do implement `Collection`s interfaces as is.

The main area where we would like to improve the design of library is in the complex declarations required to instantiate relationship aspects. Each different kind of relationship — unidirectional or bidirectional navigability; static or dynamic implementation; many-to-many or one-to-many arity; and ideally other variants on relationships such as keeping pairs indexed either by insertion order, or by an attribute of the participating objects — needs a separate abstract generic aspect implementation, and that implementation must be named explicitly in the aspect declaration. Even the distinction between the "standard" relationships (that take an explicit `Pair` class argument to represent the individual association tuples) and "simple" relationships (that use a standard `Pair` implementation implicitly) must be reflected in the names of the aspects.

Perhaps a future version of Aspect/J could support some form of type inference, default generic arguments, or even the use of annotations so that a declarations such as:

```
aspect Attends extends
 @Dynamic Relationship<@One Company,
   @Many @Unique @Sorted Employee> {
```

could replace the much more cumbersome (and overly precise) `DynamicMany2OneUniqueSortedRelationship`.

6. RELATED WORK

Relationships, associations, or collaborations have long been an important part of object-oriented analysis and design, in one way or another. Responsibility driven design's collaborations, for example, are primarily abstractions of message sends between objects [41], while other methods focus more on the structure of inter-object relationships [32, 3], inspired by entity-relationship modelling [5]. The unified modelling language [33] codified associations (in the entity-relationship style) and they have been a important part of UML modelling from the earliest version.

However relationships are modelled in programs' analyses or designs, object-oriented programmers are faced with the problems of translating those relationships into the basic features (references, collections) supported by programming languages. We have described a set of patterns describing how relationships can be hand-coded in common cases — ranging from using single reference attributes for straightforward one-to-one relationships, employing a collections library, to reifying relationships as individual objects in their own right [25]. The implementations of the relationship aspects in our library are based on these patterns. Hand-coding remains by far the most common practice for implementing relationships in object-oriented, and now aspect-oriented programming. In order to avoid programmers having to hand-code relationships, a number of more explicit approaches have been investigated [26]. Recently, many of these have focused on either executing UML models directly [29] or (more commonly) translating UML associations into code in an object-oriented programming language [34, 15, 12]. Directly executing UML has not yet proved effective in practice; translating associations suffers from the common problems of code generation: programmers need to work with their programs at two separate levels of abstraction, and then need to edit the resulting code, making "round trips" back to the designs harder. This is particularly difficult when code actually implementing relationships has to be edited — the relationships must be implemented in terms of lower-level programming language constructs, so the relationships cannot be made explicit in the resulting code.

For this reason, a number of researchers or language designers have proposed adding explicit relationship constructs to object-oriented programming languages, to support an entity-relationship modelling style more explicitly. Rumbaugh [30, 31, 36] proposed adding relationships to object-oriented languages relatively early, including support for operations to be automatically propagated across relationships. Rather more recently Bierman & Wren [2] have formally described a programming language where relationships are first class constructs, in parallel with classes. Instances of these relationships (individual relationship objects) can have their own state and behaviour similar to the "extensible pairs" in our relationship aspects. The design of the interface for our relationships is also inspired by their design, although, of course, we use generic aspects to represent relationships, while their design uses a specialised first-class language construct.

Given how natural modelling relationships with aspects turns out to be, it is surprising that there does not seem to be more work adopting this approach. Jiri Soukup developed pattern classes [38, 37] based on an ad-hoc aspect weaver for C++, that could be used to support static inter-object relationships (as well as to support object persistence). More recently, many of the aspect-oriented case studies investigating design patterns, especially Mediator and Observer [13, 27, 19] also describe aspects that implement inter-object relationships. Relationship aspects are designed to implement relationships between objects rather more specifically, and use a common interface to hide implementation details from client code.

The Adaptive Programming approach [21, 20] similarly separates the relationships between objects (in Demeter terminology the *class graph* and *object graph*) from the base code of the program. Because the class graph gives a global view of the program's structure, an adaptive programming system is able to navigate between any number of objects or classes automatically, and execute behavioural code wherever most appropriate. Our approach is more modest: relationship aspects are organised with the program's code as any other class or aspect definition. Although (like adaptive programs) relationship aspects abstract away the implementation details of relationships, a global class graph is not maintained explicitly, so programmers need to use each relationship to navigate around programs. Sullivan and Lieberherr earlier proposed a graph-based schema, where vertices and edges represent classes and their relationships [39]. This supports more complex types of relationship than we do, including the representation of functions as relationships mapping parameters to their results. As with UML class diagrams, their schema is somewhat abstracted from a real implementation and, in particular, does not consider the tradeoffs between different relationship implementations on which we focus.

Special purpose Association Aspects [35] have been proposed to model relationships, effectively as a kind of per-instance aspect that can be connected to more than one target object. Association aspects can also contain advice that executes in the context of each associated object. In comparison with relationship aspects, association aspects are a special-purpose extension (although to Aspect/J, rather than a standard object-oriented language) which focus on behavioural modification of the associated objects, rather than managing the relationships between them. Thus, while association aspects do provide a rudimentary interface for adding and removing associations, support for iterating over a relationship, or traversing from one end to the other seems to be lacking. Finally, association aspects are implemented in terms of hash tables, while — even in Aspect/J — relationship aspects can use a number of different implementation strategies, with different modifiability and performance tradeoffs.

Finally, inasmuch as it relates analysis and design models to code

— and uses aspect-orientation to keep designs explicit and untangle relationships from their participating classes — this work is also related to work on aspect-oriented analysis and design. The Theme approach, to take one example, identifies relatively large-scale concerns early in the analysis and design process: using an aspect-oriented language keeps those concerns separate in the implementation as well as the design [6, 7]. Jacobson et al. [17], use aspects to map the code from use cases onto domain objects, so avoiding tangling code between peer use-cases. Theme makes a major change to existing object-oriented development processes, while Jacobson et al. makes a more minor change: if you are already following a use-case based process, aspect design and composition replaces programmers manually apportioning use-cases to classes in the design. In contrast, our contribution is preserving relationships from either object- or aspect- oriented design explicitly into program code.

7. CONCLUSION

Early programming languages provided little support for representing either objects or the relationships between them. FORTRAN, for example, began with integers, reals, and arrays; COBOL and Algol offered a great advance by adding a range of string data types and records. Structured languages, such as Pascal, added pointers and dynamic memory allocation (rudimentary objects) to these constructs: programmers could model real-world entities as dynamically allocated records linked together with pointers. While dynamic dispatch streamlined object-oriented programs, the actual "object model" in languages from Smalltalk to Java remains essentially the same as in Pascal: dynamically allocated records linked together by pointers — with, eventually, garbage collection, and a library offering sets, bags, and lists to supplement arrays.

In this paper, we have demonstrated how aspects can raise the level of abstraction of data structures, by modelling relationships between objects explicitly. Treating relationships as aspects increases the clarity and straightforwardness of a program's code, allows programmers to interchange relationship implementations with ease, and has a negligible effect on the program's performance. Many paradigmatic uses of aspects address non-functional, often peripheral concerns, such as logging, tracing, security, persistence, or synchronisation. In contrast, using aspects to represent relationships between objects places aspect-orientation at the very core of everyday program design.

Acknowledgements

Thanks to Adrian Colyer, Elisa Baniassad, Joerg Evermann, Paul H. J. Kelly, Todd Millstein and the anonymous AOSD referees for some excellent comments on various drafts of this paper and help with Aspect/J. This work is supported by the University Research Fund of Victoria University of Wellington, and the Royal Society of New Zealand Marsden Fund.

8. REFERENCES

[1] The AspectJ 5 development kit developers notebook, `http://www.aspectj.org/`.

[2] G. Bierman and A. Wren. First-class relationships in an object-oriented language. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, pages 262–282. Springer-Verlag, 2005.

[3] G. Booch. *Object Oriented Analysis and Design with Applications*. Benjamin Cummings, second edition, 1994.

[4] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[5] P. P. Chen. The entity-relationship model — toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976.

[6] S. Clarke and E. Baniassad. *Aspect-Oriented Analysis and Design: The Theme Approach*. Addison-Wesley, 2005.

[7] S. Clarke and R. J. Walker. Generic aspect-oriented design with Theme/UML. In R. Filman, T. Elrad, S. Clarke, and M. Aksit, editors, *Aspect-Oriented Software Development*, chapter 19. Addison-Wesley, 2004.

[8] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[9] D. F. D'Souza and A. C. Wills. *Objects, Components, and Frameworks With UML: The Catalysis Approach*. Addison-Wesley, 1998.

[10] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[11] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa. Modularizing design patterns with aspects: A quantitative study. In *Proceedings of the ACM Aspect-Oriented Software Development (AOSD) conference*, pages 3–14. ACM Press, 2005.

[12] G. Génova, C. R. del Castillo, and J. Llorens. Mapping UML associations into Java code. *Journal of Object Technology (JOT)*, 2(5):135–162, 2003.

[13] J. Hannemann and G. Kiczales. Design pattern implementations in Java and AspectJ. In *Proceedings of the ACM conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 161–173. ACM Press, 2002.

[14] J. Hannemann, G. C. Murphy, and G. Kiczales. Role-based refactoring of crosscutting concerns. In *Proceedings of the ACM Aspect-Oriented Software Development (AOSD) conference*, pages 135–145. ACM Press, 2005.

[15] W. Harrison, C. Barton, and M. Raghavanchari. Mapping UML design to Java. In *Proceedings of the ACM conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 178–188. ACM Press, 2000.

[16] W. Iseberg. Check out library aspects with AspectJ 5. In *AOP@Work series*, 2006. `http://www-128.ibm.com/developerworks/java/library/j-aopwork14/`.

[17] I. Jacobson and P.-W. Ng. *Aspect-Oriented Softare Development with Use Cases*. Addison-Wesley, 2005.

[18] E. A. Kendall. Role model designs and implementations with aspect-oriented programming. In *Proceedings of the ACM conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 353–370. ACM Press, 1999.

[19] G. Kiczales and M. Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 195–213. Springer-Verlag, 2005.

[20] K. Lieberherr. Controlling the complexity of software designs. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 2–11. ACM Press, 2004.

[21] K. Lieberherr and D. H. Lorenz. Coupling aspect-oriented and adaptive programming. In R. Filman, T. Elrad, S. Clarke, and M. Aksit, editors, *Aspect-Oriented Software Development*, chapter 6. Addison-Wesley, 2004.

[22] M. Mezini and K. Ostermann. Integrating independent components with on-demand remodularization. In *Proceedings of the ACM conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 52–67. ACM Press, 2002.

[23] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *Proceedings of the ACM Aspect-Oriented Software Development (AOSD) conference*, pages 90–99. ACM Press, 2003.

[24] M. P. Monteiro and J. M. Fernandes. Towards a catalog of aspect-oriented refactorings. In *Proceedings of the ACM Aspect-Oriented Software Development (AOSD) conference*, pages 111–122. ACM Press, 2005.

[25] J. Noble. Basic relationship patterns. In N. Harrison, B. Foote, and H. Rohnert, editors, *Pattern Languages of Program Design 4*, chapter 6, pages 73–94. Addison-Wesley, 2000.

[26] J. Noble and J. Grundy. Explicit relationships in object-oriented development. In *Proceedings of the conference on Technology of Object-Oriented Systems and Languages (TOOLS)*, pages 211–226. Prentice-Hall, 1995.

[27] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, pages 214–240. Springer-Verlag, 2005.

[28] R. Pooley and P. Stevens. *Using UML: Software Engineering with Objects and Components*. Addison-Wesley, 1999.

[29] D. Riehle, S. Fraleigh, D. Buckna-Lassen, and N. Omorogbe. The architecture of a UML virtual machine. In *Proceedings of the ACM conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 327–341. ACM Press, 2001.

[30] J. Rumbaugh. Relations as semantic constructs in an object-oriented language. In *Proceedings of the ACM conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 466–481. ACM Press, 1987.

[31] J. Rumbaugh. Controlling propagation of operations using attributes on relations. In *Proceedings of the ACM conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 285–296. ACM Press, 1988.

[32] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, New Jersey, 1991.

[33] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[34] J. E. Rumbaugh. Models for design: Generating code for associations. *Journal of Object-Oriented Programming (JOOP)*, 8(9):13–17, 1996.

[35] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya. Association aspects. In *Proceedings of the ACM Aspect-Oriented Software Development (AOSD) conference*, pages 16–24. ACM Press, 2004.

[36] A. V. Shah, J. H. Hamel, R. E. Borsari, and J. E. Rumbaugh. DSM: An object-relationship modeling language. In *Proceedings of the ACM conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 191–202. ACM Press, 1989.

[37] J. Soukup. Implementing patterns. In *Pattern Languages of Program Design*, pages 395–412. Addison-Wesley, 1994.

[38] J. Soukup. *Taming C++: Pattern Classes and Persistence for Large Projects*. Addison-Wesley, 1994.

[39] G. Sullivan and K. Lieberherr. An object-oriented design methodology. Technical Report NU-CCS-95-1, Northeastern University, 1995.

[40] P. Wegner. Concepts and paradigms of object-oriented programming. *OOPS Messenger*, 1(1):7–87, 1990.

[41] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. Prentice-Hall, 1990.