Relationship Aspect Patterns

David J. Pearce and **James Noble**
Computer Science
Victoria University of Wellington
New Zealand
{djp,kjx}@mcs.vuw.ac.nz

June 9, 2006

Abstract

Relationships between objects are almost as important to designs as the objects themselves. Most programming languages do not support relationships well, so programmers must implement relationships in terms of more primitive constructs. This paper presents a selection of proto-patterns which describe how aspects can be used to model relationships within programs. By using these patterns, programs and designs can be made smaller, more flexible, and easier to understand and maintain.

Introduction

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. Aspect-oriented programming, at last, gives us a new and exciting way to think about and implement relationships in programs and this paper explores these ideas.

We present five proto-patterns for using aspects to design and implement relationships between objects — what we call *relationship aspects* [9]. The patterns address the most basic kind of relationships where one object needs to be able to refer to another object at runtime. These relationships are often called associations or collaborations, to distinguish them from the more complex concepts of aggregation and inheritance. The first two patterns (**Relationship Aspect** and **Relationship Interface**) describe the core concepts behind relationship aspects; the next three (**Relationship Pair**, **Static Relationship**, **Dynamic Relationship**) describe the major kinds of aspects (or classes) used by relationship aspects.

Aspect-oriented design and programming is an emerging field; as a result, the patterns presented here, unlike most others, **do not** attempt to record well-known design and programming techniques. Rather, in the spirit of *Documenting Frameworks Using Patterns* [3], we are experimenting with using the pattern form to describe techniques in an emerging field of software design. This makes it difficult to list three known uses of each pattern! Still, we aim to describe techniques to new programmers and illustrate when particular techniques are appropriate. Each pattern presentation includes example code based upon the *Relationship Aspect Library (RAL)* [9]. This library contains several implementations of the patterns presented here and is available for download under an open source license from `http://www.mcs.vuw.ac.nz/~djp/RAL/`. The general techniques have also also been advocated by other aspect-oriented designers [1, 2]. The example code is presented in Aspect/J (see e.g. [4]) since this is the mostly widely adopted AOP language. The patterns are mostly language independent and should apply to any sufficiently dynamic object- or aspect-oriented language, including Smalltalk and Python. They will not, however, work in languages like Java with fixed, singular notions of a class definition. Figure 1 summarises the problems dealt with by this collection of patterns, and the solutions they provide.

Pattern	Problem	Solution
Relationship Aspect	How do you design a relationship between objects?	Make a Relationship Aspect to represent the relationship.
Relationship Interface	How do you use a relationship aspect?	Access the relationship via its Relationship Interface.
Relationship Pair	How can you store additional information along with a relationship?	Make a relationship pair class to store additional information.
Static Relationship	How do you model a permanent relationship?	Use a Static Relationship aspect to implement a permanent relationship.
Dynamic Relationship	How do you model temporary relationship?	Use a Dynamic Relationship to model a temporary relationship.

Figure 1: Summary of the Patterns

Forces

Each of the patterns resolves a number of different forces, and some conflicting patterns (such as **Static Relationship (4)** and **Dynamic Relationship (5)**) resolve similar problems in different ways. Many of the patterns consider the *complexity* and *ease* of *reading* or *writing* of a particular solution. Generally, solutions which are easy to write are more likely to be chosen by programmers, and solutions which are easy to read are likely to be easier to maintain. Several patterns also address the *cohesion* and *coupling* of the resulting designs, since designs with high cohesion within objects and low coupling between them are more flexible, understandable, and easier to maintain. This is often related to whether a relationship is represented *explicitly* by a single element of a design, or whether it is *dispersed* across several objects, attributes, and methods, and whether a change of state in a relationship is *local*, affecting only those objects participating in the relationship, or *global*, affecting other objects in the program. Representing a relationship explicitly makes it easier to *identify* the relationship within the design, to *change the implementation* of the relationship if necessary, to maintain *consistency* in two-way relationships, and to *reuse* both the relationship and other participating objects elsewhere. The patterns are marginally concerned with *efficiency* — the *time* and *space* cost of a design, and the *number of objects* it requires.

Comments

This paper builds upon our two earlier papers: *Relationship Aspects* [9] and *Basic Relationship Patterns* [7]. The latter itself is an extension of still earlier work on patterns and relationships (see [8, 5, 6]). All these papers have a similar theme — how to model inter-object relationships in programs. The contribution of the patterns in this paper, we believe, lies in identifying the advantages and disadvantages of using aspect-oriented programming to achieve this aim.

1 Relationship Aspect

How do you design a relationship between objects?

Imagine implementing a university enrollment system for recording which courses a student is attending. In an object-oriented programming language, you will most likely end up with something similar to this:

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

class Course {
 String code;
 String title;
 HashSet<Student> attendees;

 void enrol(Student s) {
  attendees.add(s);
  s.attends.add(this);
 }

 void withdraw(Student s) {
  attendees.remove(s);
  s.attends.remove(this);
 }
}
```

Here, the participating classes (i.e. `Student` and `Course`) include all the data structures necessary for representing and manipulating the `Attends` relationship. This corresponds to the *Relationship As Attribute* pattern previously presented by Noble [7]. The primary disadvantages of this pattern are the *unnecessary coupling* of participants and *poor cohesion* of the relationship implementation (since this is distributed across participants).

An alternative formulation is the *Relation As Object* pattern [7]. In this case, the relationship code is housed in a separate class utilising, for example, `HashMap`s to relate participant objects. This has *lower coupling* between participants and *better cohesion* (as all relationship code is in the same class). The disadvantage, however, is an additional level of indirection (i.e. `HashMap` lookup) leading to *decreased performance*. Furthermore, if the relationship is *dense* (i.e. most, if not all, instances of `Student` and `Course` are actively participating) then this approach consumes *more storage* than *Relationship As Attribute* (since the `HashMap`s consume additional space proportional to the number of participating pairs). If the converse is true (i.e. the relationship is *sparse*), then it consumes *less storage* than *Relationship As Attribute*, since space is consumed by participating instances *only when they are actively participating*.

The difference between these two approaches for the `Attends` relationship is really an implementation detail — a trade-off for performance over storage. And yet, switching between them is not trivial, since this requires modifying both `Student` and `Course` to either remove or add the necessary relationship code. In a typical object-oriented language, there is little we can do about this. With aspect-oriented languages, however, this is no longer the case.

Therefore: *Make a Relationship Aspect to represent the relationship.*

Aspects are a natural fit for modelling relationships. They allow us to centralise the relationship code and interface, regardless of whether the implementation follows *Relationship As Attribute* or *Relationship As Object*. This is made possible in languages like Aspect/J through the *inter-type declaration* (sometimes known as the *introduction*). This language feature permits fields to be added to classes *a posteri* their definition. In this way, the *Relationship As Attribute* pattern can be implemented with an aspect which *introduces* those attributes needed for the relationship into the participants.

Example

Rewriting our example code using a relationship aspect allows the extraneous code to be removed from the `Student` and `Course` classes:

```
class Student {
 String name;
 int number;
}

class Course {
 String code;
 String title;
}

aspect Attends {
 HashSet<Course> Student.attends;     // inter-type declaration
 HashSet<Student> Course.attendees;   // inter-type declaration

 void enrol(Student s, Course c) {
  c.attendees.add(s);
  s.attends.add(c);
 }
 void withdraw(Student s, Course c) {
  c.attendees.remove(s);
  s.attends.remove(c);
 }
}
```

This code implements the *Relationship As Attribute* pattern using a simple Aspect/J aspect. This employs inter-type declarations to insert fields directly into the participating classes (i.e. `Student` and `Course`) and yields something identical to the original implementation. The difference, however, is that the relationship code is not distributed amongst the participating classes. Rather, it is centralised within the aspect itself.

In the Relationship Aspect Library, we take this one step further by providing a set of *generic* Relationship Aspects. Thus, instead of hand-coding the `Attends` aspect (as done above), we can simply extend one of our generic aspects as follows:

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

Here, the participants are identified by the type parameters of the generic aspect. The aspect modifies them accordingly, again resulting in something almost identical to the original implementation. Furthermore, changing the relationship implementation to one based on `HashMap`s (i.e. *Relationship As Object*) is easy: we simply make `Attends` extend `SimpleHashRel<Student,Course>` instead.

Consequences

The key advantages of Relationship Aspects lie in separating out the relationship concerns. Classes and relationships can be reused independently without modification and, hence, there is *lower coupling*. The design is also *more flexible*, since relationship implementations can be interchanged easily.

However:

The main disadvantage of a Relationship Aspect is simply that it is aspect-oriented: programmers are more used to reading and writing the object-oriented implementation.

Related Patterns

A **Relationship Interface (2)** can provide a uniform interface for manipulating the relationship (navigating, adding and removing participants) allowing relationship implementations to be interchanged without affecting client code. **Static Relationship (4)** and **Dynamic Relationship (5)** distinguish between long-term and short-term relationships. A **Relationship Pair (3)** allows extra information to be stored within relationships.

2 Relationship Interface

How do you use a relationship aspect?

In object-oriented programs, relationship interfaces are generally somewhat ad-hoc (e.g. `enroll`/`withdraw` in the original university system). This makes it difficult to write code which is polymorphic across different relationships. Part of the issue here stems from the differing style of interface dictated by particular relationship implementations. For example, with *Relationship As Attribute*, the interface may be distributed across the participants (since the implementation is); whilst, for *Relationship As Object*, the interface will be centralised with the relationship object. With Relationship Aspects, however, we always have a central access point for the relationship (i.e. the aspect); the value of a generic interface for manipulating relationships is, thus, increased.

Therefore: *Provide a generic Relationship Interface.*

A common interface ensures that all relationships, regardless of underlying implementation, can be manipulated in the same way. A key advantage is that this allows us to write code which is *relationship polymorphic* — that can work with any implementation of a relationship — by depending upon only the relationship interface.

Example

The Relationship Aspects in the Relationship Aspect Library are based around the `Relationship<FROM,TO>` interface. This provides methods to add and remove pairs from the relationship, and to traverse it.

```
interface Relationship<FROM,TO> {
 public boolean add(FROM f, TO t);
 public boolean remove(FROM f, TO t);
 public Set<FROM> to(TO t);
 public Set<TO> from(FROM f);
}
```

The generic parameters capture those classes the relationship is `FROM` and `TO` (in fact, there is a third generic parameter which we will discuss later). A relationship between `Student`s and `Course`s, for example, would implement the interface `Relationship<Student,Course>`, and provide the corresponding methods, such as `add(Student,Course)`, `remove(Student,Course)`, etc.

From the above definition, it is reasonably clear that a `Relationship` is *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. We can imagine other variations on this which, for example, enable traversal in only one direction (as, for many implementations, traversal in both directions is either impossible or inefficient). The above interface does not permit a `Student` to be enrolled in the same `Course` more than once. This is apparent because `to()` and `from()` return `Set`s, rather than e.g. `Collection`s. Lifting this restriction corresponds to thinking of the relationship as a *multi-graph*, rather than a *graph*.

An example of some relationship polymorphic code is the following print method, which accepts any relation between `Student`s and `Course`s:

```
void print(Student s, Relationship<Student,Course> r) {
 for(Course c : r.from(s)) {
  System.out.println(s + " attends " + c);
}}
```

This code will work regardless of what kind of relationship is passed to it.

Consequences

The ability to use different relationship implementations polymorphically is an important advantage of relationship aspects.

However:

Because programmers are unused to relationship interfaces, code written using these interfaces may be harder to understand. Part of the problem is that a general interface does not provide intuitive insight into its meaning in the real world. For example, with the original implementation of `Course` from **Relationship Aspect (1)**, there was an `enroll()` method. In our `Relationship` interface, this has been supplanted with the generic and less intuitive `add()` method.

Related Patterns

A relationship interface may express its APIs both in terms of **Relationship Pairs (3)** and in terms of the participants; supporting Relationship Pairs is generally more cumbersome but also more flexible.

3 Relationship Pair

How can you store additional information along with a relationship?

Many relationships need to store additional information along with the two participating objects. Consider implementing a *road network* system, where cities are connected together by roads of different length. The length of a road cannot be stored as an attribute of `City`, since each city may have several incoming/outgoing roads. Given v cities, we may need to store $O(v^2)$ road lengths if every city is connected with every other (assuming at most one road between any two cities). In other words, we need a length for every *pair* of cities. In the UML, this could be modelled with an *association class*. This is not supported by the relationship interfaces outlined as part of **Relationship Aspect (1)** and **Relationship Interface (2)**. The reason being that they managed the concept of a *pair* internally, rather than exposing it to the user.

Therefore: *Make a relationship pair class to store additional information.*

Some relationship aspects include a third type parameter allowing the user to specify the internal representation of pairs. By making this concept explicit, the user can include whatever attributes and other functionality are appropriate for a pair of objects in a particular relationship (such as road length).

Example

In the Relationship Aspect Library, the `Relationship` interface actually accepts three type parameters:

```
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> to(FROM t);
 ...
}
```

The third type parameter, `P`, determines the actual type of individual pair objects stored inside the relationship. All must implement `Pair<FROM,TO>`:

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

The user can provide their own pair classes by implementing `Pair` as they wish. Since we expect that, in many cases, this will not be necessary, the library provides a `SimpleRelationship` class which uses a default pair implementation (`FixedPair`):

```
interface SimpleRelationship<FROM, TO>
  extends Relationship<FROM, TO, FixedPair<FROM,TO>> {
 public void add(FROM f, TO t);
 public void remove(FROM f, TO t);
}
```

This interface corresponds more closely, in fact, with the example `Relationship` interface presented in **Relationship Interface (2)**.

Consequences

Pairs allow extra information to be stored easily into relationships. This is essential for a proper decomposition of some problems (e.g. the road network example above).

However:

Pairs *complicate* the relationship interface and, in many cases, will not be necessary.

Related Patterns

The inclusion of a `Pair` concept in the relationship interface clearly affects that interface. Hence, this pattern is is related to **Relationship Interface (2)**. Exposing `Pair` does not affect the meaning of operations for manipulating a relationship. Thus, this pattern compliments, rather than conflicts with, **Relationship Interface (2)**.

4 Static Relationship

How do you model a permanent relationship?

Many relationships exist for the duration of a program. In the university enrollment example, students are *always* attending courses; course *always* have prerequisites; and, courses *always* have teachers. Because these relationships are long-lived, and generally affect every instance of the participant classes, it is important they are as efficient as possible; that they do not require extra additional objects or memory allocation; and that they can be accessed and traversed quickly and easily.

Therefore: *Use a Static Relationship aspect to implement a permanent relationship.*

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 (by e.g. the compiler, runtime system etc.) to obtain a more efficient implementation.

Example

The Relationship Aspect Library provides a range of static relationship aspects, such as `StaticRel<FROM,TO,P>` and `SimpleStaticRel<FROM,TO>`. In our implementation, these always correspond to *Relationship As Attribute* (although, in other languages, different strategies may also be appropriate). As discussed previously, these aspects use Aspect/J's *inter-type declarations* to insert fields directly into the participants. The advantage of this is that it gives constant-time access to the relationship information (e.g. for traversal).

Consequences

Explicitly declaring that a relationship is static can enable *greater performance*; it also conveys information about the program's structure, making it *easier to understand*.

However:

In some situations, optimisations performed on behalf of a static relationship may lead to *worse performance*. In the Relationship Aspect Library, this can happen when a static relationship is *sparse* (i.e. most participants are not actively involved). This occurs because fields added as part of *Relationship As Attribute* are rarely used and consume valuable storage (especially in the machine's cache).

Related Patterns

A **Dynamic Relationship (5)** is an alternative to this pattern for short-lived relationships.

5 Dynamic Relationship

How do you model a temporary relationship?

Some relationships do not exist for the entire life of a program. Consider a prerequisites relationship for the university enrollment system. This describes a graph structure of `Course`s. As such, it is applicable to a large number of graph algorithms, including depth-first search and transitive closure. Suppose we transitively close the prerequisite graph to determine the complete set of dependencies for each course. Perhaps this is done occasionally to ensure a given degree can be completed in three years and, once completed, is discarded. Using a static relationship to

implement this is not appropriate. In the Relationship Aspect Library, for example, this would mean adding extra fields to `Course` containing its transitive closure, *even though these were only used occasionally*.

Therefore: *Use a Dynamic Relationship aspect to model a temporary relationship.*

A dynamic relationship allows the programmer to indicate the relationship will only exist temporarily. The idea is that this information can be exploited (by e.g. the compiler, runtime system etc.) to obtain better resource utilisation and, hence, a more efficient implementation overall.

Example

The Relationship Aspect Library provides a range of dynamic relationship aspects, such as `HashRel<FROM,TO,P>` and `SimpleHashRel<FROM,TO>`. In our implementation, these always correspond to *Relationship As Object* (although, in other languages, different strategies may also be appropriate). The advantage of this is that fields are not added to the participant classes. Rather the relationship is contained entirely within a separate object and, hence, destroying that object eliminates all resources consumed by the relationship.

Consequences

Explicitly declaring that a relationship is dynamic can enable *better resource utilisation*, which can lead to *better overall performance*; it also conveys information about the program's structure, making it *easier to understand*.
However:

Accessing information in the relationship may be slower as a result of the indirection (e.g. `HashMap` lookup) needed to fully separate a relationship from its participant objects. In some situations, this may lead to *worse performance overall*. In the Relationship Aspect Library, this can happen when a dynamic relationship is used as part of a computationally hard algorithm (transitive closure may be an example here). If the time-complexity of the algorithm is high, small changes in performance can have dramatic consequences.

Related Patterns

A **Static Relationship (4)** is an alternative to this pattern for long-lived relationships.

6 Acknowledgements

Thanks to Neil Leslie for some helpful formatting tips!

References

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

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

[3] Ralph E. Johnson. Documenting frameworks using patterns. In *OOPSLA Proceedings*, October 1992.

[4] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP Proceedings*, pages 327–355, 2001.

[5] James Noble. Some patterns for relationships. In *TOOLS 21*, Melbourne, 1996.

[6] James Noble. Basic relationship patterns. In *EuroPLOP Proceedings*, 1997.

[7] James Noble. Basic relationship patterns. In Neil Harrison, Brian Foote, and Hans Rohnert, editors, *Pattern Languages of Program Design 4*, chapter 6, pages 73–94. Addison-Wesley, 2000.

[8] James Noble and John Grundy. Explicit relationships in object-oriented development. In *Proceedings of the conference on Technology of Object-Oriented Languages and Systems (TOOLS)*. Prentice-Hall, 1995.

[9] David J. Pearce and James Noble. Relationship aspects. In *Proceedings of the ACM conference on Aspect-Oriented Software Development (AOSD)*, pages 75–86. ACM Press, 2005.