

Implementing First Class Relationships in Java

Stephen Nelson James Noble David J. Pearce

Victoria University of Wellington
{stephen,kjx,djp}@mcs.vuw.ac.nz

Abstract

Relationships have been an essential component of OO design since the 90s but OO languages still do not support *first-class* relationships. Hence, programmers must implement relationships in an ad-hoc fashion which results in unnecessarily complex code. We have developed a new model for relationships in OO which distinguishes tuples and relationships (mutable sets of tuples) and supports both as first-class constructs. This paper describes the design of a library for Java which provides this model.

1. Introduction

Object-oriented practitioners are frequently faced with a dilemma when they design and implement object-oriented systems: modelling languages describe object systems as a collections of objects connected by relationships [3, 14], but most object-oriented languages do not provide explicit relationship support. This results in a trade-off between high-level models which are decoupled from their implementations and low-level models which are confusing in their detail.

There have been many proposals for adding relationship support to object-oriented languages; Rumbaugh proposed a language with relationship support in 1987 [13] and there has been a recent resurgence of interest with proposals for language extensions [2, 4, 19] and library support [11, 12].

The potential benefits of such support are clear: improved traceability between design and implementation, reduced boilerplate code, better program understanding by programmers, and the opportunity to introduce new high-level language features such as queries, relational operators, and structural program constraints. In spite of these advantages none of the solutions proposed so far have achieved widespread use. We believe that existing solutions fail to

capture the intent of object-oriented models and so fail to realise those advantages.

We have previously described a new model for OO languages with first class relationships [9]. The essential difference between our model and existing models is that we distinguish between relationships and tuple extents. Tuples are links between objects, and relationships are groups of links. Both tuples and relationships have types, and both can be instantiated multiple times.

In this paper we discuss our ongoing work in implementing our model for relationships as a library in an existing language: Java. The next section summarises the three level model for relationships [9]. The following section discusses some of the related work in this area. Section 4 discusses the design of our library with some examples of use, and finally Section 5 outlines some of the interesting problems we identified during the library design process.

2. The Three Level Model for Relationships

The primary goal of this work is to demonstrate the feasibility of our three level model of relationships [9] by implementing it as a library for an existing language (Java). The three level model identifies three basic types, defined by mathematics, and interprets them as levels of abstraction for use in relationship systems. Figure 2 shows a three level model decomposition of the simple program described in Figure 1.

The first level of the three level model consists of objects, mathematical units (e.g., integers). Objects in our system are the same as those in existing systems. An object is any element of a system which has a unique identity without reference to other identities. The second level identifies tuples, similar to the relation types in the work of Vaziri *et al.* [19] and correspond loosely to named n -to- n associations in UML. Tuples in our system, like mathematical tuples, are identified by a ordered lists of other elements in the system. In our model we also allow them to have state. Tuple types (classes) are called Associations and define the types of the elements that the tuples may contain and the state which the tuples may have. The top level of our model describes mathematical sets which we interpret as relationships. Relationships in our model are tuples characterised by a type, or relation, which specifies the type of the tuples that the set may

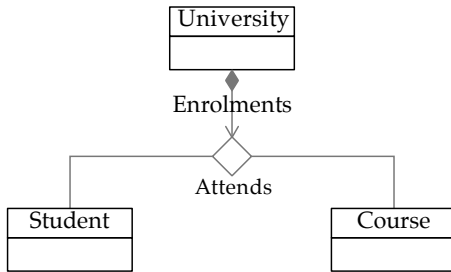


Figure 1. This figure describes simple program which uses a relationship. The university has a set of student, course tuples representing students enrolled in courses. The tuples are owned by university objects.

contain, and the state associated with each instance of the set. The key difference between our model and other models used for relationships in OO languages is that we allow multiple instances of each relation type. Most other models are based on Chen’s ER diagrams [5] where model elements may be objects or tuples, modelled by classes and relations, and leave out support for relationships as sets of tuples modelled by relations. Relationships in our model are similar to collections in traditional OO languages but with one important difference: we recognise that objects and tuples in a relationship may have state and behaviour which is specific to that object or tuple in that relationship. We encapsulate this state as a role for the object associated with the relationship.

Our model decomposition comes from the observation that UML associations usually have a wider context than simply the objects they relate. In our example system in Figure 1, the *Attends* association between Students and Courses is important for the University, not for the student and course objects themselves. While it is possible to distinguish the tuple for *Alice attending Programming in 2006* and *Alice attending Programming in 2007* by adding the year as an additional element (the year) to the tuple, this adds information which is actually irrelevant to the use of the tuple. If we consider a particular student enrolled in a particular course in multiple years, multiple universities, and even multiple faculties which offer the same course for different programs then it is clear that we need to have a relationship abstraction which is not simply the extent of the tuple.

The next section discusses the related work in this area and highlights the differences between the other models used and our model.

3. Related Work

There are several recent publications which discuss adding relationships to object-oriented languages [2, 4, 10, 11, 12, 17, 19]. These can be divided into language extensions and libraries, but the general approach is the same: take an OO language and extend it with additional constructs to represent relationships or associations. These generally have two

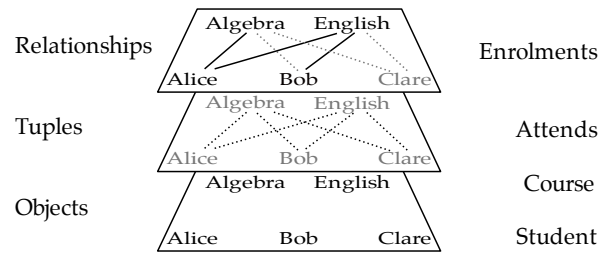


Figure 2. This figure shows decomposition of the program in figure 1 using the three level model [9]. On the bottom level we see standard OO objects, in this case student and course objects. In the centre layer we see tuples; student-course pairs representing the attends association. On the top level we have relationships (sets of tuples); in this case one enrolment relationship.

or more participant objects which are declared as a part of the signature of the relationship, either as a tuple [4], as generic parameters to the relationship [12, 11], or as annotated fields [19, 11].

Proposals for adding relationships as language features include RelJ [4], Balzer *et al.*’s constraints on relationships [2] and Vaziri *et al.*’s *Relation Types* [19]. While not directly applicable to a relationship library design, their models and constructs have influenced our model. RelJ adds tuples with state and behaviour to a Java-like language based on Featherweight Java [7]. It does not include an implementation, but the mechanism they use for relationship inheritance has influenced our design for roles. RelJ relationships are tuple extent sets and this causes complications when a tuple escapes its extent set (when a tuple is removed from a relationship but a reference survives). It does not support multiple instantiations of a single relationship. Balzer *et al.*’s relationship constraints build on RelJ to add constraints on relationships and to add fields to objects in relationships. While these are desirable features they highlight the limitations of RelJ’s abstractions: the field declarations seem to be in the context of a particular tuple but are actually global to all tuples of that type. Vaziri *et al.*’s *Relation Types* introduces tuples to Java as a language extension by annotating fields which contribute to the identity of an object and replacing object equality with an identity operator. They do not discuss the extension of tuples to relationships. This work builds heavily on Relation Types but focuses on relationships as sets of tuples and introduces tuples with a library rather than a language extension.

There are few existing libraries for introducing relationships to existing OO languages. Pearce and Noble’s Relationship Aspect Library adds support for relationships using Aspect/J to weave relationships into Java code [12]. RAL, like our model, implements relationships as mutable sets of pairs. It does not, however, provide support for polymorphic pair types; the pair implementation cannot be changed with-

out changing the relationship declaration as well, limiting reuse of generic relationships. RAL also does not support n -ary tuples, and its support for multiple instantiations of relationships is limited. They provide a *dynamic* relationship interface which can be instantiated multiple times but the focus of the work is on relationships as tuple extent sets. For example, dynamic relationships cannot add state or behaviour to participants like static relationships.

Østerbye's *No Object Is An Island* [11] takes a similar approach to our library to add relationships to C#, although its relationship model is closer to that used by RelJ. It takes advantage of C#'s features such as access to generics using reflection to achieve an impressive level of integration of relationships into code: automatically maintaining sets of referenced tuples as object fields for example. NOIAI does not support n -ary tuples or roles, and it does not allow multiple relationship instances.

Soukup's company *CodeFarms Inc.* has produced several commercial products for managing relationships [15, 16, 17]. These libraries introduce relationships as 'invasive' data-structures, using a pre-compiler to weave relationships into existing classes in a similar manner to Pearce and Noble's Relationships Aspects [12]. The underlying models used are very similar to that used by Østerbye's library, with similar limitations.

4. Library Design

This section describes the design of a library which implements the three level model. The primary goals are that each element of the model should be first-class, i.e. reified, in the system; that each element should support state and behaviour defined by the library user; and that elements should be polymorphic: any one of object, tuple, or relationship should be able to be replaced by a subtype without modification of other components. Finally, as many of the implementation details as possible should be hidden within the library: the library user should be able to use elements from the library with minimal boilerplate code.

4.1 Association is to Tuple as Class is to Object

Our library supports tuples with dependent identity, that is, two tuples with the same participants are indistinguishable (we refer to this property as the *tuple invariant*). These tuples are similar to the *Relation Types* described by Vaziri *et al.* [19]. To implement the university enrolment system in Figure 1 the library user could use tuples to represent the links between student and course objects. Ideally, this should be as simple as writing:

```
association StudentCourse {
    participant Student student;
    participant Course course;
    ...
}
```

Indeed, C# and other languages which have tuples types would not need to be extended (except that general tuples,

unlike ours, do not provide state). However, to implement the same functionality in Java the following code is required:

```
1 class StudentCourse extends Pair<Student, Course> {
2     public final Student student;
3     public final Course course;
4     protected StudentCourse(Student student, Course course,
5         StudentCourseAssoc<? extends StudentCourse> type) {
6         super(student, course, type);
7         this.student = student;
8         this.course = course;
9     }
10
11     public static final StudentCourseAssoc<StudentCourse> type =
12         new StudentCourseAssoc<StudentCourse>();
13
14     private static class StudentCourseAssoc<T extends StudentCourse>
15         extends PairAssoc<Student, Course, T> {
16         protected T create(Student a, Course b) {
17             return (T)new StudentCourse(a, b, this);
18         }}}
```

There is, obviously, a large amount of boilerplate code here. Most of it is needed to handle constructing the tuples and is irrelevant once the tuple has been constructed. We will discuss each section of the code.

The first line of code begins the declaration of a StudentCourse association. It is not necessary to call the association *StudentCourse*, this is up to the programmer. The association is binary, so we extend the Pair base-class to represent the links between students and courses. The library also provides a base class for n -ary tuples using a recursive list structure.

Lines two and three define fields for accessing the tuple participants. This is not necessary, the Pair super type provides fields from and to for this purpose, but the explicit naming can make code more understandable. It is up to the library user whether or not they define these fields.

Lines four to nine define the constructor for StudentCourse tuples. The constructor is mandatory, and must be protected to ensure that the dependent type property of tuples is preserved (two tuples with the same values share identical state and behaviour). The constructor takes as arguments the two objects it relates, and an additional parameter: its type.

The type argument to the constructor is used by the base tuple class (Tuple) as a factory for tuples. A programmer using the tuple will never call the constructor directly but rather use the type to access a tuple:

```
StudentCourse tuple = StudentCourse.type.get(alice, typeSystems);
System.out.println(tuple.student + "_attends_" + tuple.course);
//prints "alice attends type systems"
```

The get method is the only public method that a type exposes. The get method is defined in the association base class (Association) and takes the same arguments as the create method defined in line 16 of the tuple declaration above.

The get method checks a cache of existing tuples to see whether a particular tuple has already been created and if

so returns that tuple. Otherwise, it will create a new tuple using the abstract create method which is defined by the library user (as above). This creation mechanism is similar to the flyweight pattern [6]. Unlike the flyweight pattern, this library uses an inner class rather than a static method to handle creation because static methods cannot be overwritten by subclasses. This supports polymorphic construction as a tuple of polymorphic type can be created anywhere by passing the factory type as an argument.

The advantage of this system over languages with tuple types is that these tuples can have state and behaviour. Tuple classes can define state and behaviour, but not constructors, in the same way as standard Java objects and the tuple invariant property is maintained by the library factory. This ensures that it is consistent between any two references to tuples with the same parameters (by ensuring that they are the same tuple).

4.2 Relation is to Relationships as Class is to Object

Relationships are defined as sets of tuples. Relationships have a type, or relation, which defines the set of possible sets of tuples. In the university example, the property enrolments of the university in Figure 1 is a relationship instance:

```
class Enrolments extends
    BinaryRel<Student, Course, StudentCourse> {

    protected Enrolments(
        StudentCourseAssoc<? extends StudentCourse> assoc,
        AttendsRelation<? extends Enrolments> type) {
        super(assoc, type);
    }

    public static final AttendsRelation<Enrolments> type
        = new AttendsRelation<Enrolments>();

    static class AttendsRelation<I extends Enrolments>
        extends Relation<I> {
        public I create() {
            return (I)new Enrolments(StudentCourse.type, this);
        }
    }
}
```

Much of the boilerplate code for relationships is the same as tuples. The overwritten class must define a protected constructor, a static type field, and a static inner class for creating relationships. The constructor take two arguments: an association type for creating new tuples, and, like the tuple constructor, a relation factory (type) for creating new relationships. Note that the boiler-plate code for all of these examples is associated with creation, not use. This code could be easily hidden from the programmer with syntactic sugaring in a future language extension. The code for using the relationships is quite straightforward:

```
Enrolled enrolled08 = Enrolled.type.create();
enrolled.add(alice, typeSystems);
enrolled.add(alice, programming);
enrolled.add(bruce, programming);
```

```
for (StudentCourse pair: enrolled08) {
    System.out.println(pair.student + "_attends_" + pair.course);
}
//prints "alice attends type systems"
//prints "alice attends programming"
//prints "bruce attends programming"
```

Relationships provide methods for adding and removing tuples. For common tuple sizes like pairs, methods are provided for adding and removing without explicitly creating tuples. For n-ary tuples, the programmer must create the tuple manually before passing it to the relationship to store. It might seem preferable to use a variable argument-length method but this would preclude static type checking. In fact in Java where generic types are erased we could not even perform runtime checking.

Relationships implement *Iterable* so that programmers can make use of Java's foreach loops and other collection operations. Because tuples and relations are independent, the library is able to support set operations like union, subset, and intersection for any relationships which use the same tuple types. For example, the programmer could take the intersection of an enrolments relationship (student, course) and a tutors relationship (student, course) to ensure that there are no students tutoring the courses that they are currently attending.

4.3 Roles provide context for Objects and Tuples

The library features that we have demonstrated allow programmers to create tuples and relationships. They also allow inheritance, polymorphism, and provide abstraction. One problem with the language features that we have so far demonstrated is that state added to tuples is accessible wherever they are used. This may expose relationship-specific state or behaviour on objects or tuples, breaking encapsulation. One language feature that has been proposed as a solution is the introduction of roles [1].

Roles in the literature encapsulate context specific behaviour, where the context is generally a relationship. In our model which has multiple relationship instances, roles augment the behaviour of objects and tuples when they are in a particular relationship instance. For example, the Student-Course tuple in the previous section might have a grade field which is specific to the enrolled relationship. Not only that, but it is specific to a particular instance of that relationship: the student may sit the course in multiple years. This means that the role type is dependent on a relationship instance. We discuss this further in section 5.2.

Our library allows roles for both objects and tuples in the context of a relationship:

```
class Enrolments extends
    BinaryRel<Student, Course, StudentCourse> {
    ...
    public StudentCourse add(Student s, Course c) {
        StudentCourse pair = super.add(s, c);
        attends.add(pair);
    }
}
```

```

}
public final Attends.AttendsType attends
    = new Attends.AttendsType(this);
static class Attends
    extends PairRole<Enrolments, StudentCourse, Attends> {
    public char mark;
    protected Attends(Enrolments e, StudentCourse sc,
        AttendsType type) {
        super(e, sc, type);
    }
    static class AttendsType<I extends Attends> extends
        PairRoleType<Attends, StudentCourse> {
        public AttendsType(? extends Enrolments e) {
            super(e);
        }
        protected I create(StudentCourse sc) {
            return new Attends(e, sc, this);
        }
    }
} ... }

```

The role declaration is similar to other elements of our system: the role is defined as a static inner class with a protected constructor which takes its type as an argument. The role has a static inner class which serves as type and factory.

Unlike tuples and relationships, which contain a static field providing access to their type, access to role types is provided by the relationship. The relationship has a public final field (not static) which provides access to that particular relationship's role type — the role type is dependent on the relationship instance. This means that role fields associated with tuples in one relationship instance are different from role fields associated with the same tuples in another instance:

```

Enrolments enrol07 = Enrolments.type.create();
Enrolments enrol08 = Enrolments.type.create();

enrol07.add(alice, programming);
enrol08.add(alice, programming);

enrol07.attends.get(alice, programming).mark = 'D';
enrol08.attends.get(alice, programming).mark = 'B';

Enrolments.Attends ap = enrol07.attends.get(alice, programming);
System.out.println(ap.student + " got a "
    + ap.mark + " for " + ap.course);
//prints "alice got a D for programming"

```

5. Implementation Challenges

Implementing a generic relationship library supporting our requirements in Java posed several challenges which affected design decisions.

5.1 Generic Tuples

One of the most challenging aspects of the design of the library was the correct typing of relationships and tuples. This was a particular problem in Java because generics are

not available at runtime, but even in a language with runtime generics the construction problem remains.

Tuples with the same values should be indistinguishable (tuple invariant). They are able to hold state, however, so it is important to ensure that either only one tuple exists with each combination of values, or the state is copied between duplicate tuples. The first solution is the only viable one, so we chose to prevent the creation of duplicate tuples by using a factory to create them.

Rather than replicating the creation logic in every tuple type, the root tuple type (Association) provides the creation logic in the get method, and requires that subtypes implement the specific creation logic as demonstrated in the various examples we have shown so far. This is a well-known pattern (abstract factory) which seems to be a good fit for this situation.

Another challenge for implementing generic tuples was support for arbitrary tuple lengths. Some languages provide arbitrary length generic arguments, but as Java does not we implemented n-ary tuples using a recursive list structure:

```
Tuple<Head<Student, Head<Course, Tail<Tutor>>>> tutors = ...
```

This format is very verbose, and not at all efficient in the general case so we provide explicit (direct reference), fixed length tuples for small tuple lengths. If this library were used as the basis for a new language syntax based on Java then it would certainly be possible to encode the generic parameters a type-safe way, as we have demonstrated, and the language could generate the fixed-length tuples as necessary.

5.2 Dependent Role Types

One aspect of the system which we were not able to implement in a type-safe way is dependent types for roles. The type of a role is dependent on the type of the element it extends and the instance of the relation which contains the element (object or tuple). In general it is not possible to type-check dependent types but with sufficient constraints on the use of roles this may be possible. For example, if roles may only be used by code which has a direct reference (local variable) to the relationship (context) then it would be possible to statically type check such dependent types in a conservative manner:

```

Enrolled enrolments = new Enrolled();
enrolments.add(alice, programming);
enrolments.Attends alprog
    = enrolments.attends.get(alice, programming);

```

In our existing implementation in the above example the alprog variable would be typed with the static type of the role: Enrolled.Attends. This is potentially unsafe as references to roles for other relationship instances could be stored in this variable, leading to unexpected aliasing effects.

5.3 Role Method Dispatch

Roles implemented in our system are intended to extend the behaviour and state of an object or tuple in a particular

context. When the object is not used in that context the role is not relevant and should not be accessible. We implement this using the decorator pattern: the role is represented as a separate object which refers to the object it extends. It provides the same interface (via extension) and delegates non-overwritten method calls to the original object, as well as providing its own methods and state.

The method overloading must be implemented by hand, which can be a substantial burden. An alternative would be to use Java's dynamic proxy to handle calls by default. The other significant problem, as is usual in the decorator pattern, is that calls delegated to the inner object cannot make their way back to the role through virtual methods as the objects do not share a "this" pointer (the *self* problem [18]). It would be possible to implement typical virtual method behaviour by passing a self pointer to every method, however this would break compatibility with existing Java classes, and it is not yet clear what type of dispatch is reasonable for dependent role objects.

This type of extension, encapsulation and delegation is also used by Bierman and Wren in their relationship language [4]. They do not discuss the change in semantics from Java however.

6. Conclusion

We have demonstrated that it is possible to implement our model for relationships in OO languages as a library in a type-safe manner (apart from the roles issue discussed) without modifying existing type systems. We have identified and discussed several problematic areas, particularly where the behaviour we would like requires substantial boilerplate code. It would be interesting to see how much of this could be removed with some syntactic 'sugaring'. We believe that our model is a viable and attractive solution for adding relationships to OO languages, and plan to continue refining it to fit better with existing metaphors.

6.1 Future Work

We believe that this model is a powerful way of introducing relationships to OO languages. In particular, representing relationships as independent, mutable sets of tuples with their own identity has potential for powerful interaction with other developing ideas such as queries [20, 8]. We are working on some basic language sugaring to remove the boilerplate code from user code. The next step is to introduce queries over relationships and set operations on relationships (join, union, intersection).

References

- [1] BALDONI, M., BOELLA, G., AND VAN DER TORRE, L. Relationships meet their roles in object oriented programming. In *International Symposium on Fundamentals of Software Engineering* (2007).
- [2] BALZER, S., GROSS, T. R., AND EUGSTER, P. A relational model of object collaborations and its use in reasoning about relationships. In *ECOOP* (2007).
- [3] BECK, K., AND CUNNINGHAM, W. A laboratory for teaching object oriented thinking. In *OOPSLA* (1989).
- [4] BIERMAN, G. M., AND WREN, A. First-class relationships in an object-oriented language. In *ECOOP* (2005).
- [5] CHEN, P. P. The entity-relationship model - toward a unified view of data. *ACM Trans. Database Syst.* 1, 1 (March 1976), 9–36.
- [6] GAMMA, E., HELM, R., JOHNSON, R. E., AND VLISSIDES, J. M. Design patterns: Abstraction and reuse of object-oriented design. In *ECOOP* (1993), pp. 406–431.
- [7] IGARASHI, A., PIERCE, B., AND WADLER, P. Featherweight Java: A minimal core calculus for Java and GJ. In *OOPSLA* (N. Y., 1999), pp. 132–146.
- [8] MEIJER, E., BECKMAN, B., AND BIERMAN, G. Linq: reconciling object, relations and xml in the .net framework. In *SIGMOD* (2006), ACM, pp. 706–706.
- [9] NELSON, S., NOBLE, J., AND PEARCE, D. First class relationships for oo languages. In *6th International Workshop on Multiparadigm Programming with Object-Oriented Languages* (2008).
- [10] ØSTERBYE, K. Associations as a language construct. In *TOOLS* (1999).
- [11] ØSTERBYE, K. Design of a class library for association relationships. In *ACM SIGPLAN Symposium on Library-Centric Software Design LCSD'07* (2007).
- [12] PEARCE, D. J., AND NOBLE, J. Relationship aspects. In *AOSD* (2006).
- [13] RUMBAUGH, J. Relations as semantic constructs in an object-oriented language. In *OOPSLA* (1987).
- [14] RUMBAUGH, J., JACOBSON, I., AND BOOCH, G. *The Unified Modelling Language Reference Manual*. Addison-Wesley, 1999.
- [15] SOUKUP, J. Implementing patterns. *Pattern Languages of Program Design* (1994).
- [16] SOUKUP, J. *Taming C++, Pattern Classes and Persistence for Large Projects*. Addison-Wesley, 1994.
- [17] SOUKUP, M., AND SOUKUP, J. Reusable associations. *Dr. Dobbs Journal November* (2007), 51–56.
- [18] UNGAR, D., AND SMITH, R. B. Self: The power of simplicity. In *OOPSLA* (1987), pp. 227–242.
- [19] VAZIRI, M., TIP, F., FINK, S., AND DOLBY, J. Declarative object identity using relation types. In *ECOOP* (2007).
- [20] WILLIS, D., PEARCE, D. J., AND NOBLE, J. Efficient object querying for java. In *ECOOP* (2006), pp. 28–49.