

First Class Relationships for OO Languages

Stephen Nelson, David J Pearce, and James Noble

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

Abstract. Relationships have been an essential component of OO design since the 90s and, although several groups have attempted to rectify this, mainstream OO languages still do not support *first-class* relationships. This requires programmers to implement relationships in an ad-hoc fashion which results in unnecessarily complex code. We have developed a new model for OO languages which presents relationships as the dominant mechanism for defining object-oriented behaviour. We believe that a language based on this model could bring the benefits of relationships to mainstream languages and allow better integration between OO systems and other paradigms such as relational databases.

1 Introduction

Object-oriented practitioners are frequently faced with a dilemma when they design and implement object-oriented systems: modelling languages have long described object systems as a collection of classes defining objects and collection of relationships and associations between them. Modern object-oriented languages, however, do not provide the same level of support for relationships, so practitioners must juggle the advantages of a clear decomposition in the modelling phase with the rather simplistic relationships available to them at implementation.

Relationships are not new-comers to the object-oriented paradigm: drawing on the ER model[4] Rumbaugh described an object-oriented language with relationship support in 1987 [13], Cunningham and Beck identified relationships as one of the distinguishing features of object-oriented design in 1989 [2], and relationships have been an essential part of the Unified Modelling Language (the dominant modelling language for the object-oriented methodology) since its infancy [14].

In spite of widespread recognition of the importance of relationships to the object-oriented paradigm there is still little support in mainstream languages, which has led to a recent resurgence of interest in providing support for relationships in object-oriented languages [10, 3, 12, 1, 15, 11].

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 for better paradigm integration between object-oriented programs and the relational databases prevalent in modern systems.

We intend to address the disconnect between object-oriented design and implementation by rethinking the way object-oriented languages are structured. Rather than adding relationships to existing language models we propose that existing language models should be re-factored to support relationships as a primary metaphor, rather than a late-life extension. We have developed a set of requirements with which to identify good relationship models [7], and used these requirements to develop a new, three-layered model for the object-oriented paradigm which focuses on relationships rather than objects for describing behaviour which we describe here.

2 Relationship Model

Traditional object-oriented systems consist of groups of interacting objects, generally modelled by classes. Typical relationship systems introduce relationships or associations at the modelling level which describe links, or groups of links, at the data level [13, 10, 3, 12, 1, 15, 11]. We believe that these systems do not adequately describe relationships because they either fail to consider groups of links at the data level or focus on groups of links (relationships) at the expense of the individual link at the modelling level. This introduces confusion in the literature because it is not clear whether a relationship is a link or a group of links, and whether the term can be used at the modelling or the data level, or both.

We believe that by modelling objects, links, and relationships (groups of links) separately the resulting system will be easier to understand and consequently easier for programmers to use. To clarify the distinctions between the modelling elements in our system we introduce a three tiered system for describing programs. This section discusses the three tiers, which are shown in Figure 1.

2.1 Object Tier

The object tier consists of objects (modelled by classes) and value-types such as integers etc.

Definition 1 (Objects and Classes). *Suppose X is a class of objects. An object x has type X iff $x \in X$.*

Objects may have associated state and behaviour. Object state may include mutable and immutable fields storing value types which may be exposed via the object's public fields. Objects may also have mutable and immutable fields storing other (mutable) system entities where there is strong composition between the two entities. This is enforced using ownership (owner as dominator): the composed entity cannot escape the scope of the object and it is tied to that object for its lifetime.

By enforcing these constraints the model ensures that references to mutable internal state cannot escape from objects, preventing inner aliasing. Inter-object

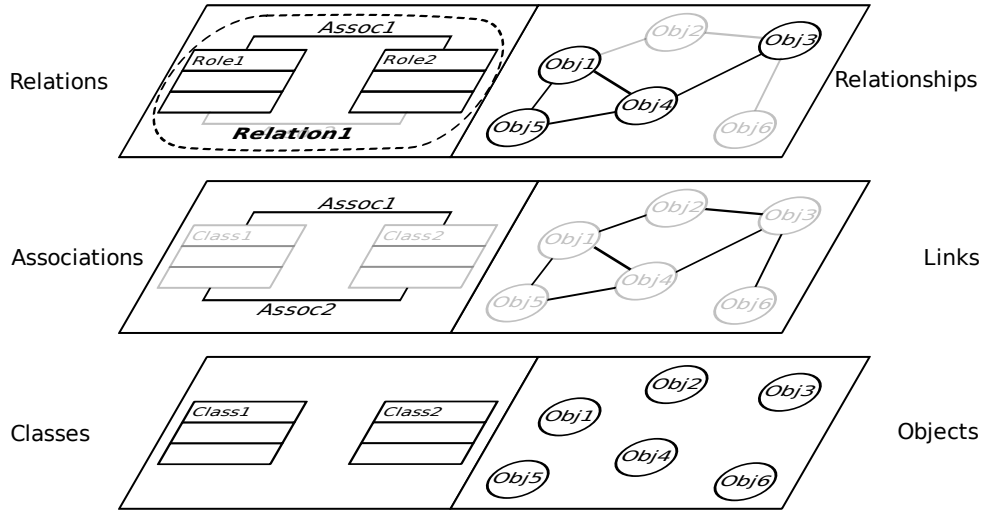


Fig. 1. The three tiers in our relationship system. The bottom tier (object tier) describes the objects in the system and the classes that describe them. The middle (link tier) adds the links between objects and associations which model them, and the top (relationship tier) describes properties of groups of objects and links, including roles, relationship constraints, and collections. These are modelled by relations. The three tiers are discussed in §2.

communication which is traditionally supported through aliasing should be handled at higher levels of the model.

Objects in our system are analogous to records in a relational database which have an automatically allocated unique primary key: they can be uniquely distinguished without reference to any state. The class of the object is the description of the record (its type).

2.2 Link Tier

The next tier of the model builds on the objects and classes (and value types) defined in the object tier to create composite types: links (tuples) defined between objects and modelled as associations between classes. The objects which the link associates are known as the link's *participants*, and are immutable.

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

Definition 2 (Links and Associations). Suppose we have classes $X_0 \dots X_n$, where an object x has type X iff $x \in X$. The association A between $X_0 \dots X_n$ is the cross-product of $X_0 \dots X_n$, thus $A = X_0 \times \dots \times X_n$. A link a is a tuple consisting of a pair of objects (x_0, \dots, x_n) where $x_0 \in X_0 \wedge \dots \wedge x_n \in X_n$. Thus, $a \in A$.

Links in our system are similar to the relation types proposed by Vaziri *et al.* [15]. There is an important difference, however: links in our system may not have state and behaviour. This prevents the possibility of two links existing with the same participants and different mutable state. Vaziri *et al.* do allow state and behaviour and their system prevents the creation of duplicate tuples, this solution is compared with our approach in §4.1.

A link is analogous to a record in a relational database which has multiple foreign keys: each ‘key’ corresponds to an object which the link relates and together they create a new uniquely distinguishable record.

2.3 Relationship Tier

A relationship is a group of links, modelled by a relation (relation is to relationship as class is to object):

Definition 3 (Relationships and Relations). *Suppose we have classes of objects X_0, \dots, X_n which are associated by an association A where $A = X_0 \times \dots \times X_n$. A Relationship r is a subset of A , and the Relation R is the set of all possible relationships r , thus $r \subseteq X_0 \times \dots \times X_n$ and $R = \mathcal{P}(X_0 \times \dots \times X_n)$.*

Objects, Links, and Relationships together form the basis for any possible data decomposition in our model, however for the model to be useful we need to have state and behaviour which is associated with relationships.

Two objects of type Person (“Joe”) and Event (“Comp103”) may be related to each other by a relationship of type Attends (“Victoria University”) but the relationship is not useful without additional state and behaviour which discusses their interactions: Victoria University may have logic for determining whether a person may attend an event, Joe may have a student id, Comp103 may have a list of courses which are prerequisites, and there may be a grade associated with both Joe and Comp103.

All of these properties depend on the relationship “Victoria University”, but they also depend on other entities. For this reason we introduce roles, which are extensions of system entities associated with relationships.

Roles are like mixins; they define state and behaviour for objects and links (and even other relationships) which is separate from the object’s main class but is added to the object at runtime in the context of the relationship. For example, when Joe is Attending Comp103 he is a student. This does not subsume Joe’s identity as a person—he may be in other relationships which are unrelated to Joe’s role as a student. Rather, Joe’s role as a student adds more functionality: in the context of being a student Joe can have a student id, enrolment details, and functionality for calculating fees. These details are not relevant outside of the student context and they do not affect his identity, so it does not make sense to include them in the general description of Joe as a person.

The roles we introduce in this system are similar in concept and intent to the ‘member interposition’ fields proposed by Balzer *et al.* [1]. Balzer proposes that relationships should be able to add fields to their participants within the

declaration of the relationship. We extend this to support extensions for links as well, and encapsulate the extensions new (dependent) types which can be accessed and extended by derivative relationships.

A relationship is analogous to a table in a database: the table is a group of records, however the contents of the table does not distinguish the table from other tables. A role is a record which is distinguished by a foreign key from another table (a link or object in a relationship).

3 Implementing the Relationship System

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

We propose that the three tiers identified in the previous section should be explicitly implemented in the language: the first layer describes objects and classes, the second describes links and associations, and the third describes the behaviour of the system in terms of relationships and roles, expressed as relationships. Each layer consists of M0- and M1-layer components in UML parlance, with classes defining objects, associations defining links, and relations defining both relationships and roles [8].

The rest of this section presents the core components of our language in a partial grammar similar to that used in Featherweight Java [5] and RelJ [3]. The grammar is not intended to be complete, and we do not provide formal semantics for the language: instead we give an informal description of their intended semantics.

3.1 Objects

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

$$c \in \text{Class} ::= \text{class } c \text{ [extends } c'] \{ \\ \quad \text{Field*} \\ \quad \text{Method*} \\ \quad \}$$

Unlike standard Java fields, fields in our system represent composition relations; this means that an entity (object or relationship) stored in a field of an object is tied to that object for its lifetime. In addition, we use ownership to enforce that references to the composed entity (or its components) cannot escape the object. We believe that this will encourage programmers to create good composition hierarchies and preserve a strong hierarchical structure for the system as a whole. If a programmer wants a more relaxed or symmetric interaction than composition then that should be reified as a relationship.

In addition to mutable system entities, fields may store (shallow)-immutable entities such as basic types and tuples. Restrictions on these are far more lenient:

the fields may be mutable or immutable and the entities may refer to other (mutable) types, so long as the entity the field references is immutable.

A definition of a person class in our system would look like this:

```
class Person {  
    const string name;  
    Address home;  
}
```

3.2 Links

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

```
 $a \in \text{Association} ::= \text{association } a \{$ 
```

(participant t p)+

```
}
```

This definition of associations is similar to Vaziri *et al.*'s *relation type* declarations [15] except that the term *key* is replaced with *participant*, and associations may not define state and behaviour (this is added at a later stage).

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

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

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

The association construct in our system is intended to embody the associations discussed in UML [9]: a semantic connector between objects representing interaction or association between them.

Unlike UML associations, a link has no identity independent of the participants it associates. We feel that independent link identity weakens the link abstraction, making links more like objects. Links are valuable precisely because their identity is entirely dependent on the objects they associate.

We do not allow associations to have state or behaviour associated with them, and the participants of a link are immutable, like final fields in Java. This means that two different associations declared with the same participants are equivalent and can exchange links; the name of the association is merely a convenience for the programmer.

For the same reasons, there is no value in having association inheritance (without modification of their participants) as there can be no difference between different association declarations with the same types. However, there is subtyping between associations: associations are covariant on the types of their participants.

3.3 Relationships and Roles

Relationships are collections of links. A relationship may extend another relationship, and it nominates the type of the links which it contains, either by reference to an association or by direct reference to participant types (or type parameters).

Relationships are intended to represent the extent set of an association. The closest analogy to this in a traditional object-oriented system would be access to the set of objects which implement a class, however, there can only be one instance of this set in traditional systems whereas there can be multiple relationships which use a given association and have different sets of links.

Relationships might also be described as a link factory: the relationship is responsible for creating and maintaining links between objects. It can provide access to them and add state and behaviour through roles. The distinction here would be that the relationships share the links which they create with the same participating objects.

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

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

where T is a type name for a role.

The relation has a name which corresponds to the type name of its relationships. Unlike associations, the relation's type name is significant (like that of an class); two relationships which are otherwise identical will be distinct if they have different type names.

We define inheritance for relationships as a sub-typing relation rather than a set operation: if a subtype relationship contains a link then its super-type relationship must also and vice-versa. This is possible because inherited relations must be declared on the same association type; there is no variance allowed between derived relations. An alternative would be to allow relationships and associations to be covariant, and require that the super-type is a super-set of the subtype. However, this would introduce well-known variance issues. Instead, we suggest that if such a system is required, two instances of the relationship are created and an additional relationship is added to maintain the consistency of the sets.

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

The body of a relation declaration is similar to that of a class: it defines the fields and methods which will be present in the relationships the relation defines. In addition, the relation may define roles for its participants and its association. These roles add fields and methods to any objects or links which are contained in a relationship so that the relationship can access them. They will also be externally accessible in some cases, although we have not finalised how this works.

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

```

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

```

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

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

In addition to the roles defined, the relation defines a method called *getUnpaidStudents()* which returns a set of students who have not paid their fees. Note that within the context of the relation the attendees have type *Student*, whereas they are actually of type *Person*. For the fields of *Student* to be accessed the context must be aware that the student role is present, and it must know which relationship the role is for.

Connections between Objects, Links, and Roles It is interesting to consider the connections between objects, links, and roles. Links are defined in the tier above objects, but objects have independent identity whereas links do not. Roles and objects have fields and methods whereas links do not. In fact, we

believe that objects are actually roles for special links: if we create a link with one participant: a memory address, we have essence of an object. From there, we can define a relationship called "Object" with a role for that link which adds the state and behaviour we expect from objects. In this way, the object as an independent entity becomes redundant: merely sugar for an underlying mechanism dependent on links and roles.

If we extend this decomposition we find that relationships are in fact a subset of object (ignoring the ownership requirements), and thus composed of links and roles. In this way, the entire system can be decomposed into (immutable) value types, and roles for instances of those types which define state and behaviour. This decomposition is probably not as useful for understanding program intentions however, so we prefer to discuss object systems composed of three levels of data: objects and basic types, links, and relationships (groups of links).

4 Related Work

There has been a great deal of recent work which discusses adding relationships of object-oriented languages [10, 3, 12, 1, 15, 11]. These can be divided into language extensions and libraries, but the general approach is the same: take an object oriented language and extend it with additional constructs, similar to objects, known as relationships or associations. These generally have two or more participant objects which are declared as a part of the signature of the relationship, either as a tuple [3], as generic parameters to the relationship [12, 11], or as annotated fields [15, 11]. This mingles the definitions of links and relationships which leads to confusion when state is added: it is not clear whether the state is associated with the link or the relationship, and regardless of which is chosen, support for the other becomes challenging.

By separating the definitions of associations and relations and adding state with roles, it becomes clear which properties belong to links, which to objects and which to relationships. Declaring association participants within the association the declaration can be broken into several lines, avoiding the tendency for declarations to run on which has become the hallmark of extensions to Java. In addition, the participants are given relevant role names (attended, attendee) which correspond to the role names in UML and can be referred to within the context of the relationship and the association's roles. This avoids possible confusion caused by using role names like *from*, *to* to refer to participants where it may not be clear which participant is being referred to [12, 11]. It also simplifies the syntax for relationships of greater arity than two.

There are some feature of recent work which are particularly relevant to our model which we discuss in the remainder of this section.

4.1 Relation Types

Vaziri *et al.*'s relation types are a particularly interesting approach to the problem of relationship representation. They use minimal language modifications to

implement a system with first-class tuples, a language feature that seems essential to adding relationships to object-oriented languages. Their relation types nominate a few 'key' fields which are immutable and determine the identity of the type instance, though it is not clear that overriding the 'equals' method is the best approach to this problem.

Relation types are very similar to our tuples, with the key difference that they allow state in the relation types, whereas our tuples do not. By allowing state they are forced to ensure that only one instance of each key combination is constructed, which they do using a constructor method which returns existing types if they occur. This mitigates the problem, but limits the potential applications of their system. For example, the database field has many useful operations such as join, union and intersect which operate across tables (relationships) to extract records with a particular criteria. Suppose we have one relationship called "Customer" between people and companies, and another called "Employee", also between people and companies. If we take the intersection of the two we can find the customers who are also employees, and by inspecting each role we can extract information from them, such as the sum of sales to employees. In Vaziri *et al.*'s system however it is not clear whether this operation can occur, certainly it would not be supported by their type system. However, even if it could occur, it would not be clear what should be returned; customer or employee.

4.2 Member Interposition

Balzer et al. describe a technique they call member interposition, essentially a mechanism for adding extra fields to objects in the context of a relationship [1]. These fields can be accessed by the relationship to store information (for example, student id on a person) which is particular to an object in the context of a relationship. This is somewhat similar to *aspect weaving* described in Aspect Oriented Programming, where fields can be inserted into classes without modification of the class [6].

We have extended this feature in our model to create roles, a term which is prevalent in the literature, but not generally associated with relationships as we use it here. We believe that roles are a more natural way of describing these object extensions than other approaches because they define a new, explicit type which programmers can refer to and systems can use for static type checking. The use of roles in programs should be familiar to programmers as it mimics the behaviour of object-oriented extension, while at the same time introducing a powerful new language feature. The use of dynamic role introductions should allow more flexible and efficient programming, without losing language features like static type checking. Of course, this remains to be proven through implementation and formal analysis, however we are confident that this can be achieved and look forward to presenting results in this area in the future.

4.3 Language Extensions and Libraries

There have been several attempts to implement relationships with libraries rather than language extensions [12, 11]. Library implementations have an advantage over language extensions because the library can be added to existing code without modification of code which does not require the features, and library development can be performed independently from language development. Existing work suggests that many relationship features can be implemented within the confines of existing languages (as libraries), but there are also limitations apparent which can be resolved with language extensions. We believe that there are probably certain key features of relationships which are best implemented with language extensions. It may be that these already exist, for example Vaziri *et al.*'s relations dependent identity (see §4.1) and Balzer *et al.*'s member interposition. Regardless, it is not clear at present which features will ultimately prove to be necessary so we have added the language features we deem necessary to implement the functionality which we would like, but plan use libraries where it is clear that they are sufficient. If it becomes clear with experience in relationship system development that some features are better implemented as libraries, then we will attempt to do so.

5 Conclusion

Relationships are an essential part of the object-oriented paradigm which are generally not well supported by existing programming languages. This paper describes our proposal for a new model for object-oriented programming which draws on ideas from the relational database paradigm and existing work on implementing relationships in object-oriented languages. We group objects into links (immutable tuples akin to database rows) and then group the links into relationships (akin to tables). We are developing a language based on Java which is class-based with relation definitions describing relationships and roles, associations defining links, and classes defining objects.

There are no doubt many issues with the language which we have proposed. We believe, however, that the ideas presented are novel and interesting; worthy of further consideration and discussion.

We are proceeding to implement this language by translation to Java bytecode, and will no doubt revisit both the language definition and the underlying model in the light of that experience.

References

1. BALZER, S., GROSS, T. R., AND EUGSTER, P. A relational model of object collaborations and its use in reasoning about relationships. In *ECOOP* (2007).
2. BECK, K., AND CUNNINGHAM, W. A laboratory for teaching object oriented thinking. In *OOPSLA* (New York, NY, USA, 1989), ACM, pp. 1–6.
3. BIERMAN, G. M., AND WREN, A. First-class relationships in an object-oriented language. In *ECOOP* (2005), pp. 262–286.

4. CHEN, P. P. The entity-relationship model - toward a unified view of data. *ACM Trans. Database Syst.* 1, 1 (March 1976), 9–36.
5. IGARASHI, A., PIERCE, B., AND WADLER, P. Featherweight Java: A minimal core calculus for Java and GJ. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '99)* (N. Y., 1999), L. Meissner, Ed., vol. 34(10), pp. 132–146.
6. KICZALES, G., LAMPING, J., MENHDHEKAR, A., MAEDA, C., LOPES, C., LONGTIER, J.-M., AND IRWIN, J. Aspect-oriented programming. In *ECOOP*. Springer-Verlag, 1997.
7. NELSON, S. First-class relationships for object-oriented languages. Tech. rep., Victoria University of Wellington, 2008.
8. THE OBJECT MANAGEMENT GROUP. *Meta Object Facility (MOFTM)*, 2.0 ed. 140 Kendrick Street, Building A Suite 300, Needham, MA 02494, U.S.A.
9. OBJECT MANAGEMENT GROUP. *Unified Modelling Language (UML) 2.1.1*, 02 2007.
10. ØSTERBYE, K. Associations as a language construct. In *TOOLS* (1999).
11. ØSTERBYE, K. Design of a class library for association relationships. In *LCSD* (2007).
12. PEARCE, D. J., AND NOBLE, J. Relationship aspects. In *AOSD* (New York, NY, USA, 2006), ACM Press, pp. 75–86.
13. RUMBAUGH, J. Relations as semantic constructs in an object-oriented language. In *OOPSLA* (New York, NY, USA, 1987), ACM Press, pp. 466–481.
14. RUMBAUGH, J., JACOBSON, I., AND BOOCH, G. *The Unified Modelling Language Reference Manual*. Addison-Wesley, 1999.
15. VAZIRI, M., TIP, F., FINK, S., AND DOLBY, J. Declarative object identity using relation types. In *ECOOP* (2007), pp. 54–78.