

First Class Relationships for OO Languages

Stephen Nelson, David J Pearce (Advisor), and James Noble (Advisor)

{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 relationships in OO which provides a better abstraction than existing models provide. We believe that a language based on this model could bring the benefits of relationships to mainstream languages.

1 Problem Description

Object-oriented practitioners are frequently faced with a dilemma when they design and implement object-oriented systems: modelling languages describe object systems as a graphs of objects connected by relationships [2, 9], but most object-oriented languages have no explicit support for relationships. This results in a trade-off between high-level models which are de-coupled from their implementations and low-level models which are confusing to use as they contain irrelevant detail.

There have been many proposals for adding relationship support to object-oriented languages. Rumbaugh proposed a language with relationship support in 1987 and there has been a recent resurgence interest with proposals for language extensions [1, 3, 10] and library support [6, 7].

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, relationship (function) operations, and program structure constraints.

In spite of the clear advantages of relationship support none of the solutions proposed so far have achieved widespread use, and while this may be due to limited time and exposure we believe that existing solutions fail to capture the intent of object-oriented models and so fail to realise those advantages.

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 [1, 3, 6, 7, 10] we propose that existing language models should be re-factored to support relationships as a primary metaphor.

2 Goal Statement

Our goal is to address the disconnect between object-oriented design and implementation by rethinking the way object-oriented languages are structured.

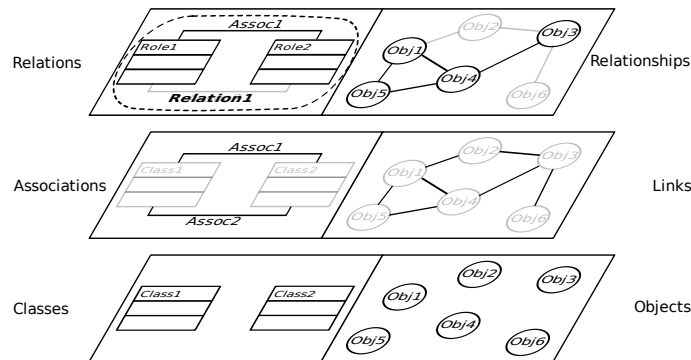


Fig. 1. The three tiers in our relationship system. The bottom tier (object tier) describes the objects in the system. The middle (link tier) adds the links between objects, and the top (relationship tier) describes properties of groups of objects and links, including roles, relationship constraints, and collections.

This will allow us to realise the advantages long promised by relationships and introduce new high-level language features such as relationship queries, relationship (function) traversal operations, and program structure constraints. We have developed a new model for the object-oriented paradigm which focuses on relationships rather than objects.

3 Relationship Model

Typical relationship systems introduce relationships or associations (class level) which describe links, or groups of links in implementation (object level)[1, 3, 7, 6, 8, 10]. We believe that these systems do not adequately describe relationships because they either fail to consider groups of links at the implementation 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 implementation level, or both.

We believe that by modelling objects, links, and relationships separately as classes, associations and relations, the resulting system will be easier to understand and consequently easier for programmers to use. To clarify the distinctions between the model elements we introduce a three tiered system for describing programs which is shown in Figure 1.

Object Tier The object tier consists of objects (modelled by classes), similar to the objects defined in traditional object-oriented models. Objects may have state and behaviour, but they may not communicate with other objects on this tier unless there is a strong composition relationship between them. This ensures that more complex relationships are moved up into the relationship tier.

Link Tier The next tier of our relationship system builds on the objects and classes defined in the object tier by adding connections (links) between objects

described as associations between classes. An association between classes in this system indicates that the classes are related in some way. For example, an association between the classes Person and Course indicates that people may be linked with courses.

Links are immutable tuples containing one instance (object) from every class in their association, similar to Vaziri *et al.*'s *relations* [10]. Unlike Vaziri *et al.*'s relations, our links may not have any state or behaviour associated with them. This removes a lot of the complexity of associations in UML and other systems which makes them easier to describe and understand. For example, there is no distinction between sub-typing and subsetting of associations in this system: an association between subtypes is naturally a subset of the related association between super-types because all possible tuples in the subtype association are also tuples in the super-type association. Instead of having complex associations, we introduce an additional modelling element.

Relationship Tier Relationships, introduced at the third tier of our system, are sets of links which have type and identity. For example, we can describe the notion of people attending courses as a subset of people and a subset of courses which are linked by some subset of the possible links between the set of people and the set of courses. Relationships exist as runtime entities modelled by 'relations'.

Unlike links and associations, which have the same type if they have the same type parameters (participants), relationships have a type defined by their relation in the same way as objects have a type defined by their class. For example, there may be two relationships between People and Courses, one of type "Attends" and the other of type "Teaches". These can even contain the same links but are not equivalent because they have incompatible type. There may even be two instances of type "Attends" which contain the same links and are not equivalent: relationships have independent identity in the same way as objects.

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

The three levels of abstraction which we identify are typically collapsed into a single level in the object-oriented paradigm. This causes interwoven, fragile code which is confusing to create and to maintain. The dependence of some object state and behaviour on relationships is seldom recognised because object-oriented programming languages obscure relationship concerns by describing their implementation rather than their intention. Even languages which do address relationships as separate entities fail to separate the behaviour associated with relationships from the classes [8, 3], or lose the association between the state and behaviour, and the objects with which it is associated [5, 7]. We believe that our relationship system simplifies the categorisation of behaviour as object, link, or relationship (role) behaviour and provides a more natural way to describe the related behaviour than Aspect-Oriented programming [4].

4 Validation

To validate our model we plan to use it to develop a language. We will present a grammar for the language, provide typing rules and semantics, and we will provide proofs of type safety, soundness and correctness. We will also provide an implementation of the language. This will highlight any practical problems with the language which are not apparent from the theoretical model. The implementation will consist of a compiler and a library system which will allow programs to be written in our language. Finally, we will conduct various case studies; implementing different systems in our language and others so that we can make comparisons between them.

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* (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. KICZALES, G., LAMPING, J., MENHDHEKAR, A., MAEDA, C., LOPES, C., LONGTIER, J.-M., AND IRWIN, J. Aspect-oriented programming. In *ECOOP*. 1997.
5. NOBLE, J., AND GRUNDY, J. Explicit relationships in object oriented development. In *TOOLS* (1995), Prentice-Hall.
6. ØSTERBYE, K. Design of a class library for association relationships. In *LCSD* (2007).
7. PEARCE, D. J., AND NOBLE, J. Relationship aspects. In *AOSD* (2006), ACM Press, pp. 75–86.
8. RUMBAUGH, J. Relations as semantic constructs in an object-oriented language. In *OOPSLA* (1987), ACM Press, pp. 466–481.
9. RUMBAUGH, J., JACOBSON, I., AND BOOCH, G. *The Unified Modelling Language Reference Manual*. Addison-Wesley, 1999.
10. VAZIRI, M., TIP, F., FINK, S., AND DOLBY, J. Declarative object identity using relation types. In *ECOOP* (2007), pp. 54–78.