# First-Class Relationships in Object Oriented Programs

Stephen Nelson David J. Pearce James Noble
Computer Science, Victoria University of Wellington, NZ
{stephen,djp,jkx}@mcs.vuw.ac.nz

***Abstract*—Relationships have been an essential component of OO design since the 90s yet mainstream OO languages still do not support *first-class* relationships. Most programs implement relationships in an ad-hoc fashion which results in unnecessarily complex code. We examine requirements a good first-class relationship abstraction and compare this with existing work. A system which implements the requirements presented in this paper would allow programs to be less complex, easier to understand and more reusable.**

## I. Introduction

The object-oriented paradigm describes a collection of objects which interact by sending messages between each other [1]. Each object is composed of some state (data) and some behaviour (methods/functions). When objects communicate one will initiate 'conversation' by passing a message to another. The object which initiates conversation must have an address to send it to. This means that objects need to know about other objects. An object "knowing" about another object creates a relationship, or association, between the objects. These relationships are explicit in object-oriented design [2] but are relegated to second-class status in mainstream OO languages [3], [4], [5].

Armstrong identifies 8 "*Quarks*", or fundamental building blocks of object-oriented development: Abstraction, Class, Encapsulation, Inheritance, Object, Message Passing, Method and Polymorphism [6]. The 8 *Quarks* are the 8 concepts most commonly identified in object-oriented development literature between 1966 and 2005. Notably absent from this list is *Relationship* which takes a poor 13th position on a list of object-oriented development concepts. In fact, only 14% of the papers surveyed mention relationships, whereas each of the top 8 *Quarks* received mention in at least 50% of the papers surveyed. This demonstrates the neglect of relationships in OO literature.

Few languages provide support for explicit relationship, yet they are a fundamental concept in object-oriented paradigm: Rumbaugh, Jacobson and Booch claim that "associations are the 'glue' that ties the system together" [2]. Pearce and Noble quote John Donne, "no man is an island, entire of itself" to illustrate the requirement for relationships in object-oriented programs [5]. They identify the relationship disconnect between design and implementations as a severe limitation of current OO languages. Balzer, Eugster and Gross claim object *collaborations* (modeled by relationships) are key to understanding large object-oriented programs [7] and Bierman and Wren claim that "the programmer is poorly served when trying to represent many natural *relationships*" [4].

If relationships are present in every object-oriented program then why are they not in the top 8 "Quarks" of object-oriented development? Section II introduces the history of relationships in OO and describes the concept of a relationship in an OO language. Section III discusses the requirements for a language or system which claims to have first-class relationships, and Section IV discusses some of the recent literature related to relationships with reference to the requirements in Section III.

## II. Relationships in the OO literature

Relationships were identified early in the history of computer science: In 1976 Chen identified that viewing data as consisting of entities and relationships allows a more natural model for the real world [8]. Eleven years later Rumbaugh claimed that while OO languages support entities through Objects, support for relationships is lacking [3]. He proposed extensions to OO languages to support first-class relationships and with others constructed a language called DSM which implements his relationship extensions [9].

In the 1990s, the *Unified Modelling Language* (UML) was defined, drawing on work by Rumbaugh and others to include relationships as a fundamental component of OO design [2]. UML defines association as "the semantic relationship between two or more classifiers that involves connections among their instances" (This work equates relationship with association; the UML *relationship* is more general, including association, generalization, and other modelling tools).

We build on the definition of association in UML to define relationship as a semantic connection between $n$ classifiers (classes or relationships). The classifiers are called *participants*. A relationship instance is a set of $n$-tuples (*connections* in UML) between instances, where each element in the tuple corresponds to an instance of the model element in the same position in the relationship.

Figure 1 demonstrates a simple relationship (Attends) between a `Student` class and a `Course` class. This relationship has been frequently used in the literature for demonstrating relationships [4], [5], [7]. The solid line between Student and Course indicates that the relationship is an association relationship. The same relationship could also be represented as a power set of the cartesian product of the participants:

$$\textit{Attends} \subseteq \mathcal{P}(\texttt{Student} \times \texttt{Course})$$

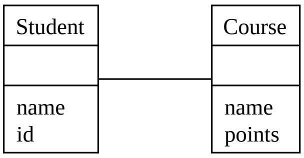


Fig. 1. A simple relationship shown in a UML class diagram

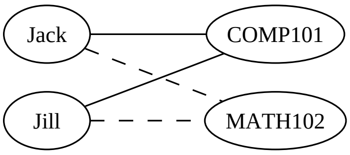


Fig. 2. Two instances of the *Attends* relationship

This representation expresses the relationship as the set of all possible sets of `Student-Course` tuples. The order imposed in by the tuples allows same class to be present twice in the expression without ambiguity.

When a relationship is instantiated there are two interesting components to consider. The first are tuples created by the cross product of the participants which consist of references to instances:

$$(\text{“Jack”}, \text{“COMP101”})$$

The second interesting component of a relationship is the set of tuples (and element in the relationship set). This is known as the *extent set* of a relationship and corresponds to a particular instantiation of the relationship in a running program. This paper refers to the extent set as *relationship instance*. Generally there is only one relationship instance for each relationship in a program, but it is interesting to consider multiple instances in the same program, such as enrolments for multiple years. Figure 2 shows a the objects in a program with two instances of *Attends* represented by solid and dashed lines respectively. The same object graph could be represented as two sets of tuples from the relationship (instances):

$$\{(\text{“Jack”}, \text{“COMP101”}), (\text{“Jill”}, \text{“COMP101”})\}$$
$$\{(\text{“Jack”}, \text{“MATH102”}), (\text{“Jill”}, \text{“MATH102”})\}$$

The literature is not consistent when referring to relationships: UML for example calls relationship tuples connections, and many works do not consider relationship instances, instead considering a relationship to be any subset of the cross-product of its participants.

## III. Requirements for a Relationship Abstraction

First-class relationships in OO languages have clear benefits to programmers and designers: traceability between design and implementation is improved [10], coupling between relationships and participants is reduced [5], and complexity and inconsistency is reduced [4]. A good implementation of first-class relationships would provide all of these benefits and more.

Unfortunately there is no accepted benchmark for relationship support for evaluating systems which claim to support relationships. Noble identifies a possible set of requirements [11] based on concepts which are commonly regarded as important to OO languages [6]. The section briefly discusses each requirement.

### A. Abstraction

Abstraction is an essential tool for coping with complexity in programs. Abstraction is particularly important to the OO paradigm where it is used to hide implementation details of object behaviour behind a shared interface [12]. A good relationship abstraction should allow relationship implementation to be encapsulated within the relationship and provide a shared interface for interaction. This would improve complexity handling by removing relationship implementation details from code which uses it.

### B. Polymorphism

Polymorphism in the OO paradigm allows different implementations for a common interface [6]. A good relationship abstraction should allow relationship polymorphism. This would allow different relationship implementations to be switched without modification of participants and clients.

### C. Reusability

The OO paradigm uses abstraction to improve reusability: a class which was written for one system can be reused in the same system or another because it is not statically coupled with the rest of the system. In the same way, a relationship abstraction would allow relationships to be reused: for example a relationship might be instantiated multiple times in the same system. This cannot be done in traditional OO systems because the relationship is statically woven into the system.

### D. Composition

Composition is an important concept of the OO paradigm: components of the system can be composed together to allow implementation to be shared, and to reduce complexity. In the same way a good relationship abstraction would allow relationships to be composed.

### E. Separation of Concerns

The Aspect Oriented methodology [13] claims that separating different program concerns procudes programs which are more traceable, less tightly coupled, and more reusable. A good relationship abstraction would separate the relationship concern from participants, providing the same advantages. This would have the added advantage of supporting reusability by allowing the participants to be reused without the relationship, and without each other.

### F. Additional Requirements

In addition to the five requirements above, there are secondary requirements and constraints which need to be addressed such as: what is the optimal syntax for expressing relationships; should relationships be explicit or implicit, how should roles (relationship context of participants) be addressed? Any work which implements first-class relationships should address these problems.

## IV. Survey of Recent Work on Relationships

This section contains a brief survey of recent work on relationships. These go some way towards a relationship abstraction which satisfies the requirements in the previous section but none provide all of the requirements.

### A. Relationship Java

Bierman and Wren define a language called *RelJ* which includes a small, functional subset of Java with extensions providing first-class support for relationships[4]. RelJ allows programmers to define relationships between objects, specify attributes and methods on the relationships, and create relationship hierarchies. The authors provide a complete formalism for RelJ with a type system and small-step operational semantics for the system.

There is currently no implementation of the RelJ language and there are some aspects of the language which could cause issues in implementation. In particular, the specification for inheritance in RelJ is quite different to Java: polymorphism is handled by delegation for both objects and relationships. It is not clear whether this could be adapted to match Java as the relationship hierarchy depends on this behaviour.

The authors do not provide runtime access to extent sets in their language and separation of concerns is limited because there is strong coupling between the relationship and its participants.

### B. A Relational Model of Object Collaborations

Balzer et al use mathematical relations to model relationships; they define classes as sets of objects and relationships as a subset of the cartesian product of the participants. They then outline a language for expressing consistency constraints on the relationships using mathematical notation.

The language support for relationships in their system is a loose extension of RelJ with added support for constraints. They do not provide a full specification or an implementation.

The authors discuss a concept they call *member interposition*: adding fields to an object which participates in a relationship which are accessible to code in the relationship. This is particularly interesting as it improves separation of concerns while providing some notion of relationship roles: state associated with both the relationship and the participant.

### C. Relationship Aspects

Aspect Oriented Programming [13] allows programmers to define separate *cross-cutting concerns* by writing *Aspects* which encapsulate a single ‘aspect’ of the system in a modular manner. Pearce and Noble identify that relationships are not part of their participants [5]. Consider the `Attends` relationship in Figure 1: it is separate from both `Student` and `Course` so the participants could be reused in another context where an attends relationship is not relevant. As relationships are not essential to the class they are cross-cutting concerns.

The authors present a library of relationship aspects, the *RAL*, implemented in AspectJ [14] which defines several common types of relationships, for example *MultiRelationship*s which allow duplicate tuples and *UniRelationship*s which are optimised for one-way access. The RAL provides relationship abstraction, polymorphism, and separation of concerns as well as some reusability. Composition support is more limited.

This paper goes some way towards providing a good relationship abstraction. Most of the primary concerns are addressed, but it is not first class, it does not have very elegant syntax (use of generics clutters the interface) and the authors do not address relationship roles.

## V. Conclusion

Relationships are an important part of OO design which are neglected in OO languages. By supporting relationships as first-class participants in OO programs programmers can benefit from reduced coupling between relationships and their participants, improved traceability between design and implementation, and stronger relationship encapsulation.

Unfortunately it is not clear how relationships should be expressed in OO languages. Several recent works have proposed different solutions but the community needs a clear specification of the relationship abstraction for implementation languages, just as the UML specification of association provides for design.

In future work we plan to further develop the requirements detailed in §III and provide a relationship abstraction which satisfies them. In addition, we plan to provide a reference language specification and implementation to demonstrate the advantages of relationships in OO languages.

## References


[1] A. Goldberg and D. Robson, *Smalltalk-80: the language and its implementation*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1983.

[2] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modelling Language Reference Manual*. Addison-Wesley, 1999.

[3] J. Rumbaugh, “Relations as semantic constructs in an object-oriented language,” in *OOPSLA*, 1987.

[4] G. M. Bierman and A. Wren, “First-class relationships in an object-oriented language.,” in *ECOOP*, pp. 262–286, 2005.

[5] D. J. Pearce and J. Noble, “Relationship aspects,” in *AOSD*, 2006.

[6] D. J. Armstrong, “The quarks of object-oriented development,” *Commun. ACM*, vol. 49, no. 2, pp. 123–128, 2006.

[7] S. Balzer, T. R. Gross, and P. Eugster, “A relational model of object collaborations and its use in reasoning about relationships,” in *ECOOP*, 2007.

[8] P. P. Chen, “The entity-relationship model - toward a unified view of data,” *ACM Trans. Database Syst.*, vol. 1, pp. 9–36, March 1976.

[9] A. Shah, J. Rumbaugh, J. Hamel, and R. Borsari, “Dsm: An object-relationship modelling language,” in *OOPSLA*, 89.

[10] J. Noble and J. Grundy, “Explicit relationships in object oriented development,” in *TOOLS*, 1995.

[11] J. Noble, “Roles and relationships.” Keynote at Roles and Relationships Workshop (ECOOP07), July 2007.

[12] G. Booch, *Object-Oriented Analysis and Design with Applications*. The Benjamin/Cummings Publishing Company, Inc., second edition ed., 1994.

[13] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, “Aspect-oriented programming,” in *ECOOP*, 1997.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, “An overview of AspectJ,” *LNCS*, pp. 327–355, 2001.