Ownership for Relationships (Position Paper)

Paley Li
Victoria University of
Wellington
lipale@ecs.vuw.ac.nz

Stephen Nelson
Victoria University of
Wellington
stephen@ecs.vuw.ac.nz

Alex Potanin
Victoria University of
Wellington
alex@ecs.vuw.ac.nz

ABSTRACT

Ownership systems express hierarchical data structures well but can have difficulty expressing relationships between peers. Relationships systems, on the other hand, do not address ownership issues. This paper considers the recent work on relationship systems in the context of ownership, and presents ideas for adding ownership to relationship systems.

1. INTRODUCTION

Ownership, the concept of one object controlling access to, or *owning* another has been proposed in various forms [1, 4, 6, 7, 8, 9, 12]. It identifies control hierarchies within object-oriented programs to address issues surrounding aliasing.

Independently, first class relationships have been proposed as a mechanism for addressing aliasing in specific cases. First class relationship systems address collaborations between objects, distinct from object associations such as composition, and inheritance, and add support to implementation languages to reduce aliasing and coupling [2, 3, 10, 11, 13].

Both of these fields address issues surrounding object interactions, and specifically, the representation of and use of various pointer structures, yet to our knowledge there has been little collaboration between the fields.

This work considers first class relationships in the context of ownership and discusses the problems encountered when combining the two concepts.

2. RELATIONSHIPS

Relationships are used in modelling languages to represent non-trivial interactions or dependencies between classes. There have been several proposals for adding relationships to programming languages as first class constructs [3, 11, 13]. Many of them use a simple example which we present in Figure 1. It describes a simple university enrollments system where Student objects attend Course objects and receive a grade for their efforts. This could be implemented by storing a list of courses each student attends and the mark they received, or the students attending each course, or both.

Relationship systems generally advocate extracting the relationship from both student and course and presenting it as a concrete system entity. Different relationship systems implement this in different ways.

Bierman and Wren [3] provide relationships analogous to classes: they are declared between the classes they relate and may have variables and methods of their own. Relationships may be instantiated by calling methods on the relationship type.

```
alice = new Student();
comp = new Course();
Attends.add(alice, comp);
```

Noble and Pearce [11] present relationships as a library, but use a similar syntax. They also support *dynamic* relationships, which can have multiple instances:

```
Attends attends09 = new Attends();
attends09.add(alice, comp);
attends09.get(alice, comp).mark = 'A';
```

Østerbye [10] demonstrates that relationships implemented in this style can be accessed in the participants as properties without affecting implementation decisions. This requires minimal changes to the participant classes.

3. OWNERSHIP

Ownership is an object encapsulation mechanism which restricts access to objects by assigning each object an owner. Any modification to objects must go through its owner. There are two main disciplines of ownership. Owner-as-dominator ensures that access from any reference to any object must be made through the owner of the object. Owner-as-modifier states that only the owner of an object may mutate it, but allows unrestrictive read only access to any object.
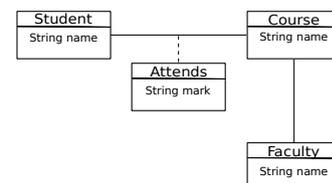

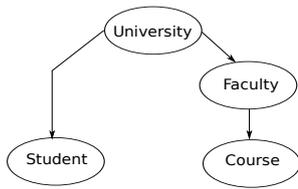Figure 1: Relationship example of an association class

Figure 2: Object ownership diagram of a basic University

Ownership Generic Java (OGJ) [12] provides owner-as-dominator discipline by introducing an ownership type parameter to the Java's generic type parameters. OGJ provides three ownership type parameters to express ownership: objects created with the `This` ownership parameter belong to the creating object, objects with the `World` parameter can be accessed by any object, and `Owner` corresponds to the owner of the current instance. OGJ enforces *deep* ownership: where any reference chain from the root of the object hierarchy to the object must go through the object's owner.

Generic Universe Types (GUT) [8] provides owner-as-modifier by adding read-only references to ownership. Like OGJ, GUT uses generic types. Objects in GUT can have one of three ownership modifiers: `peer`, `rep` or `any`. Objects with the `peer` parameter have the same owner as the current object. Objects with the `rep` parameter are owned by the current object. Finally, objects with the `any` parameter can have any object as their owner.

4. OWNERSHIP FOR RELATIONSHIPS

The example used to demonstrate relationships also demonstrates the ownership problem with relationship systems. Figure 1 shows that a student who attends a course receives a mark, but it doesn't specify who can change the mark. Obviously the student shouldn't be able to change their own mark, but it would be unfair if their mark could be changed indiscriminately by any third party with access to the course. Clearly, there is a need for some control over, at least, mutation of the mark field. It is also not clear who can modify the students enrolled in courses. We propose that ownership can be used for this purpose. Specifically, we will use owners as modifiers because it is the weaker of the two systems, although the observations we make could also apply to owners as dominators.

Viewing the example in its context, we hypothesise that there must be a faculty member responsible for the course, and that both the student and the faculty member are under the control of the university. Figure 2 shows an ownership diagram for the objects involved. A Faculty member who owns the course that they teach, but they do not own the student. Both the faculty and the student are owned by the university.

Assuming that we are happy with the ownership hierarchy in Figure 2 lets consider where the relationship 'Attends' fits in. The relationship is clearly a property of both the Student and the Course because changing the relationship affects both: the student is affected by the courses that they are expected to attend, and the course must ensure that it has spaces for any students enrolled. Multiple owners of the relationship (student *and* course) would not be correct

either, even if they were supported [5], because this would allow either object to make changes independently. Giving ownership of the relationship to the faculty member is not correct either because they have no ownership relationship with the student. The only logical solution is to give ownership of the relationship to the University. This leads us to our first principle for relationship ownership:

Principle 1 *A relationship is owned by the lowest common owner of the participants involved.*

Unfortunately, there are still problems with the system we have described. It is reasonable for the university to handle the enrolment of students into courses, but it is unnecessarily restrictive to force a faculty member to go through university bureaucracy to assign a mark to a student in their course. This is not a problem with the general system: we do not want the faculty member to arbitrarily add or remove students from courses, even courses they teach, but the university should be able to delegate control over the properties of the individual records of enrolment: when a student is enrolled in a course we would like to assign an owner for that individual enrolment record so that the faculty can control it.

Pearce and Noble [11] make a distinction between relationship tuples and relationship extents. A relationship tuple is the record of particular instance participation in a relationship; a particular student attending a particular course. A relation extent is the mutable collection of tuples representing the currently active participants. We can exploit this distinction to address separate ownership of relationship membership (extent) and relationship details such as the mark (tuple).

Suppose we give the ownership of the relationship tuples (attendance records) to faculty members, and the relationship extent (attends relationship) to the university. The faculty member can create an attendance record for any student and any course, and mutate the mark at will, but the university does not need to acknowledge that the record exists because it is not in the universities extent. If the faculty member wants the university to recognise the attendance record then they must give it to the university. The university can then add the attendance record (tuple) to the relationship (extent).

Figure 3 shows an ownership diagram describing this system. The university owns the relationship extent (attends) but it does not own the attendance record directly. The record is owned instead by the faculty member that created it for their course. It is up to the university to ensure that it only adds records owned by the faculty members that teach the course, and that they check with the student before doing so.

The following is the second principle for relationship ownership that we have distilled:

Principle 2 *Relationship owners can allow other objects to submit relationship records to be added to the relationship. It is up to the relationship owner whether or not to add the records to the relationship.*

5. CONCLUSION

Current relationship systems do not directly support the system we have described: one system already supports re-
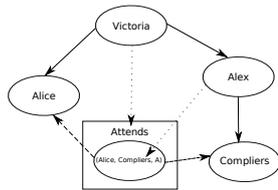
Figure 3: Ownership diagram with relationships

lationship extents as objects (hence own-able) [11] but most would have to add this support. We believe that this is crucial for providing ownership guarantees for relationships. Relationship systems should also support the external creation of relationship. This is already somewhat supported by Pearce and Noble, but could be improved upon. If these changes are made, then systems such as the one we have described here can be created using existing ownership systems.

6. REFERENCES

[1] Jonathan Aldrich and Craig Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *ECOOP*, volume 3086, pages 1–25, Oslo, Norway, June 2004. Springer-Verlag.

[2] Stephanie Balzer, Thomas R. Gross, and Patrick Eugster. A relational model of object collaborations and its use in reasoning about relationships. In *ECOOP*, LNCS, pages 323–346. Springer, 2007.

[3] Gavin M. Bierman and Alisdair Wren. First-class relationships in an object-oriented language. In *ECOOP*, pages 262–286, 2005.

[4] Chandrasekhar Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, EECS, MIT, February 2004.

[5] Nicholas Cameron, Sophia Drossopoulou, James Noble, and Matthew Smith. Multiple ownership. In *OOPSLA*, 2007.

[6] Dave Clarke. *Object Ownership and Containment*. PhD thesis, School of CSE, UNSW, Australia, 2002.

[7] David Clarke, John Potter, and James Noble. Ownership Types for Flexible Alias Protection. In *OOPSLA*, pages 48–64. ACM, 1998.

[8] Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic universe types. In *ECOOP*, 2007.

[9] James Noble, Jan Vitek, and John Potter. Flexible Alias Protection. In *ECOOP*, (LNCS), pages 158–185. Springer-Verlag, July 1998.

[10] Kasper Østerbye. Design of a class library for association relationships. In *LCSD*, 2007.

[11] David J. Pearce and James Noble. Relationship aspects. In *AOSD '06*, pages 75–86, New York, NY, USA, 2006. ACM Press.

[12] Alex Potanin. *Generic Ownership*. PhD thesis, SMSCS, Victoria University of Wellington, 2007.

[13] James Rumbaugh. Relations as semantic constructs in an object-oriented language. In *OOPSLA*, pages 466–481. ACM Press, 1987.