Efficient Object Querying for Java

Darren Willis, David J. Pearce, and James Noble

Computer Science, Victoria University of Wellington, NZ
{darren, djp, kjx}@mcs.vuw.ac.nz

Abstract. Modern programming languages have little or no support for querying objects and collections. Programmers are forced to hand code such queries using nested loops, which is both cumbersome and inefficient. We demonstrate that first-class queries over objects and collections improve program readability, provide good performance and are applicable to a large number of common programming problems. We have developed a prototype extension to Java which tracks all objects in a program using AspectJ and allows first-class queries over them in the program. Our experimental findings indicate that such queries can be significantly faster than common programming idioms and within reach of hand optimised queries.

1 Introduction

No object stands alone. The value and meaning of objects arise from their relationships with other objects. These relationships can be made explicit in programs through the use of pointers, collection objects, algebraic data types or other relationship constructs (e.g. [5,27,28,29]). This variety suggests that programming languages provide good support for relationships. We believe this is not entirely true — many relationships are *implicit* and, as such, are not amenable to standard relationship constructs. This problem arises from the great variety of ways in which relationships can manifest themselves. Consider a collection of student objects. Students can be related by name, or student ID — that is, distinct objects in our collection can have the same name or ID; or, they might be related by age bracket or street address. In short, the abundance of such implicit relationships is endless.

Most programming languages provide little support for such arbitrary and often dynamic relationships. Consider the problem of *querying* our collection to find all students with a given name. The simplest solution is to traverse the collection on demand to find matches. If the query is executed frequently, we can improve upon this by employing a hash map from names to students to get fast lookup. This is really a *view* of the original collection optimised for our query. Now, when students are added or removed from the main collection or have their names changed, the hash map must be updated accordingly.

The two design choices outlined above (traversal versus hash map) present a conundrum for the programmer: which should he/she choose? The answer, of course, depends upon the ratio of queries to updates — something the programmer is unlikely to know beforehand (indeed, even if it is known, it is likely to

D. Thomas (Ed.): ECOOP 2006, LNCS 4067, pp. 28–49, 2006.

change). Modern OO languages compound this problem further by making it difficult to move from one design to the other. For example, consider moving from using the traversal approach to using a hash map view. The easiest way to ensure both the collection and the hash map are always consistent is to encapsulate them together, so that changes to the collection can be intercepted. This also means providing a method which returns the set of all students with a given name by exploiting the hash map's fast lookup. The problem is that we must now replace the manual traversal code — which may be scattered throughout the program — with calls to this new method and this is a non-trivial task.

In this paper, we present an extension to Java that supports efficient querying of program objects. We allow queries to range over collection objects and also the set of all instantiated objects. In this way, manual code for querying implicit relationships can be replaced by simple query statements. This allows our query evaluator to optimise their execution, leading to greater efficiency. In doing this, we build upon a wealth of existing knowledge on query optimisation from the database community. Our focus differs somewhat, however, as we are interested in the value of querying as a programming construct in its own right. As such, we are not concerned with issues of persistence, atomic transactions, roll-back and I/O efficient algorithms upon which the database community expends much effort. Rather, our "database" is an object-oriented program that fits entirely within RAM.

This paper makes the following contributions:

- We develop an elegant extension to the Java language which permits object querying over the set of all program objects.
- We present experimental data looking at the performance of our query evaluator, compared with good and bad hand-coded implementations. The results demonstrate that our system is competitive with an optimised hand-coded implementation.
- We present a technique which uses AspectJ to efficiently track object extent sets.
- We present experimental data looking at the performance of our object tracking system on the SPECjvm98 benchmark suite. The results demonstrate that our approach is practical.

2 Object Querying

Figure 1 shows the almost generic diagram of students attending courses. Versions of this diagram are found in many publications on relationships [5,6,11,31]. Many students attend many courses; Courses have a course code, a title string and a teacher; students have IDs and (reluctantly, at least at our university's registry) names. A difficulty with this decomposition is representing students who are also teachers. One solution is to have separate `Student` and `Teacher` objects, which are related by name. The following code can then be used to identify students who are teachers: