Using Recursive Java Generics to Support Code Reuse in the CS2 Course

J. Andrew Holey and Lynn R. Ziegler
Department of Computer Science
Saint John's University and the College of Saint Benedict
Collegeville, Minnesota 56321
jholey@csbsju.edu and lziegler@csbsju.edu

Abstract

Showing students examples of code reuse in early Java programming courses is highly desirable, but one significant class of examples, linked data structures, is difficult to reuse and leads to frequent casting. We show a technique for using recursive generic parameters that significantly simplifies the inheritance of linked data structures, and we discuss the use of this technique in the CS2 course.

1 Introduction

Java is well-established as the most frequently-used language in the first two programming courses at colleges and universities (CS1 and CS2). While there is serious and well-founded debate about whether other languages may be more appropriate for an introductory programming sequence at the college level, many computer science faculty members have become comfortable with using Java in these courses and are able to teach programming and problem-solving very effectively using this language.

One of the advantages of using object-oriented languages like Java for teaching introductory programming is their support for the construction of abstract data types (ADTs). This support was enhanced in Java 5.0 with the introduction of generics. Libraries like the Java Collection Framework have reduced the emphasis on construction of ADTs in many CS2 courses, but most textbooks and instructors still choose to spend considerable time on the implementation of ADTs. It is now possible to guide students to produce simple and elegant implementations of the standards ADTs, including stacks, queues, various types of lists and trees, maps and even graphs.

These generic data types illustrate one type of code reuse, which is another advantage claimed for object-oriented languages. Unfortunately, another obvious candidate for code reuse, the underlying data structures on which the ADTs are built, is much more difficult to achieve. In particular, linked data structures are difficult to reuse. Linked implementations of ADTs are an essential component of the CS2 course, and they provide a natural way to build students' facility with recursion. It's precisely at the point where these concepts come together that code reuse becomes problematic.

This paper will propose a recursive generic template for linked structures that solves the problem described above. Section 2 will describe the problem in detail; in Section 3, we propose our solution in the context of one-way linked structures; Section 4 extends the solution to other linked structures; in Section 5, we discuss our experience with using this technique in our CS 2 our; finally, Section 6 suggests areas for further investigation. The code examples presented in this paper are all abbreviated for presentation purposes; the full code, including documentation, is available at www.users.csbsju.edu/~jholey in the ZHStructures link.

2 Initial Problem

We will begin with the relatively simple one-way linked structure. This structure is typically composed of linking components (often called nodes) that contain a single data

element and a link to the following component. A generic Java class for such a structure would be something like the following:

```
public class OneWayLinkedStructure<ElementType> {
   protected ElementType element;
   protected OneWayLinkedStructure<ElementType> next;
   // Constructors and accessor and mutator methods
}
```

There are a number of possible ways to use such a class. The ADT may directly extend the class, or it may have one or more instance variables that refer to the class. Depending on how we want to handle access to the instance variables of the class, we may need methods such as *getElement*(), *getNext*(), *setElement*() and *setNext*() or we may require the ADT to provide an inner class that extends the class or we may simply make the element and next variables public. We might also choose to provide methods for commonly needed operations like *isEmpty*(), *contains*() and *iterator*().

This class works perfectly well for implementing stacks, queues and other lists that don't require special forms of recursive traversal. We run into trouble, however, when we want to extend this class and attach methods to it in a recursive data structure. For example, we might want to implement a LISP-like list by extending the linked structure as follows:

```
public class LispyList<ElementType>
extends OneWayLinkedStructure<ElementType> {
   public LispyList() { super(); }
   public LispyList(ElementType car, LispyList<ElementType> cdr) {
      super(car, cdr);
   }
   public ElementType car() { return this.element; }
   public LispyList<ElementType> cdr() {
      return ((LispyList<ElementType>) this.next);
   }
}
```

The problem is the cast in the *cdr*() method. It's a fairly trivial problem here, but in more complex classes, many more casts may be necessary, and it may be difficult to determine whether the cast is appropriate in the given context. Casts are not only a nuisance; they can lead to runtime errors that may be difficult to detect with testing or to correct when they are finally discovered. Furthermore, such casts generate compiler warnings. We can tell students to ignore the warnings, but doing so sets a bad precedent since most compiler messages convey important information.

In this example, the cast shouldn't really be necessary here since the dynamic type of the *next* variable will always be *LispyList*. We could solve this problem by eliminating the

inheritance and duplicating the linked structure code directly in the class itself, but with this solution we give up code reuse.

A similar problem arises when we move to a doubly-linked structure. The only difference between a one-way linked structure and a two-way linked structure is the addition of a *previous* reference in the latter. What could be more natural than to create the two-way structure as follows?

```
public class TwoWayLinkedStructure<ElementType>
extends OneWayLinkedStructure<ElementType> {
   protected TwoWayLinkedStructure<ElementType> previous;
   // Constructors and additional needed methods
}
```

This approach doesn't work very well, however. The inherited *next* variable has a static type of *OneWayLinkedStructure*, but its dynamic type should always be *TwoWayLinkedStructure*. It is actually possible to mix one-way and two-way components in a single structure, but doing so is almost certainly an error—an error we shouldn't give students the opportunity to make at the level of a CS2 course. If we persist with this approach, we need to cast any reference to *next* if the context requires a *TwoWayLinkedStructure*. If the dynamic type of the object turns out to be *OneWayLinkedStructure*, we get a runtime exception.

The usual solution to this problem is to give up on the inheritance and declare the two-way structure as a stand-alone class with any common code duplicated from the one-way structure class. In this example as well as the previous, we'd really like to make the static type of the *next* variable be the class extending *OneWayLinkedStructure*, but there's no easy way for the base class to anticipate the subclass. In the next section we show how to resolve this problem.

3 One-Way Linked Structures with Recursive Generics

It is possible to create linked-structure classes in which the static type of the links is that of the extending class rather than the base class. We accomplish this by making the link class a second generic parameter of the base class and constraining that parameter to extend the base class. For the one-way linked structure, the class definition is:

```
public class ZHOneWayLinkedStructure
<ElementType, StructureType extends
              ZHOneWayLinkedStructure<ElementType, StructureType>> {
   // "ZH" prefix for classes developed by Ziegler & Holey
   protected ElementType element;
```

```
   protected StructureType next;
   // Constructors and accessor and mutator methods
}
```

Now the class header here is ugly as sin, but it gets the job done. The *StructureType* generic parameter is the static type of the *next* variable. We don't expect students to understand the header completely, let alone to be able to construct it themselves, but they have little trouble learning to use it. The LispyList class now becomes:

```
public class LispyList<ElementType>
extends OneWayLinkedStructure<ElementType, LispyList<ElementType>> {
   // all code is unchanged except
   public LispyList<ElementType> cdr() {
      return this.next;
   }
}
```

The implementation here is quite straightforward. All the students need to know is that they must supply the name of the class they are declaring for the *StructureType* generic parameter. Since the static type of the *next* field matches its dynamic type, they no longer need to cast it, and their own code tends to be more readable and less error-prone.

Before we go on to examples of other linked structures, we give an example from a more complex class. Consider implementing a sorted list class using a one-way linked structure. The class header for such a list might be:

```
public class ZHLinkedSortedList
               <ElementType extends Comparable<ElementType>>
extends ZHOneWayLinkedStructure
               <ElementType, ZHLinkedSortedList<ElementType>> {
```

In this implementation, we have an additional *state* variable in the full implementation of the structure that indicates whether a component is empty or not, and we keep an empty component at the end of the list rather than a null pointer. We want to implement a recursive add method that inserts a new element in the correct position in the list, but only if that element is not already present. It returns true if the list was modified by the call to add. The code for the method is:

```
public boolean add(ElementType element) {
   if (this.isEmpty()) {
      this.element = element;
      this.next = new ZHLinkedSortedList<ElementType>();
      this.state = ZHComponentState.NOT_EMPTY;
      return true;
   }
```

```
      int comp = element.compareTo(this.element);
      if (comp < 0) {
         this.next = new ZHLinkedSortedList<ElementType>
                                    (this.element, this.next);
         this.element = element;
         return true;
      }
      else if (comp == 0) {
         return false;
      }
      else {
         return this.next.add(element);
      }
   }
```

Without the recursive generics, casts would be required in two places: in the second constructor call and in the final return statement. In the return statement two extra sets of parentheses would be required to make it work:

```
return ((ZHLinkedSortedList<ElementType>) (this.next)).add(element);
```

We doubt whether most students would be able to figure out how to make that cast work. In our solution, no casting is required.

4 Other Linked Structures with Recursive Generics

Our recursive generic technique works equally well with other types of linked data structures. In Section 2, we showed how it is difficult to extend a one-way linked structure to a two-way linked structure using simple inheritance. With recursive generics, it becomes natural to define a two-way linked structure by extending the one-way structure. The new code is:

```
public class ZHTwoWayLinkedStructure
<ElementType, StructureType
              extends ZHTwoWayLinkedStructure<ElementType,
                                              StructureType>>
extends ZHOneWayLinkedStructure<ElementType, StructureType> {
   protected StructureType previous;
   // Constructors and additional needed methods
}
```

Once again, the class header is rather daunting, but it's almost exactly parallel to the one-way structure declaration. The internal code and the use of the class are both straightforward, with most of the instance variables and methods inherited without change from the parent class. It might, however, provide a *ListIterator* in place of a one-

way *Iterator* if the instructor chose to attach an *iterator*() method to these two classes. Using the two-way structure, it is possible to implement queue, deque and various list classes with simple, elegant code.

As the underlying data structures and the classes that use them grow more complex, the benefits of using recursive generics become greater. Binary search trees and their underlying binary tree structures provide a good example. We can implement a linked binary tree structure in a way equivalent to the linear structures:

```
public class ZHBinaryTreeStructure
<ElementType, StructureType extends
              ZHBinaryTreeStructure<ElementType, StructureType>> {
   protected ElementType element;
   protected StructureType leftChild, rightChild;
   // Constructors and accessor and mutator methods
}
```

This class is structurally identical to the two-way linked structure, but it is used differently, and the links have different names. For this reason, we chose not to implement this class by extending an existing linked structure. In our own implementation, we have included methods for pre-order, in-order and post-order iterators as well as protected utility methods like *find*() and *findLeftmost*() that return references to components within the tree.

The binary search tree class builds directly on the binary tree structure. The first stage is an abstract binary search tree that retains the StructureType generic parameter:

```
public class ZHLinkedBinarySearchTree
               <ElementType extends Comparable<ElementType>>
extends ZHBinaryTreeStructure
               <ElementType,ZHLinkedBinarySearchTree<ElementType>> {
   // Constructors and appropriate methods
}
```

The methods here follow directly from the recursive definitions of the binary search tree methods. For example, the *add*() method, which is very similar to the add method for the sorted list, is:

```
public boolean add(ElementType element) {
   if (this.isEmpty()) {
      this.element = element;
      this.leftChild = new ZHBinarySearchTree<ElementType>();
      this.rightChild = new ZHBinarySearchTree<ElementType>();
      this.state = ZHComponentState.NOT_EMPTY;
```

```
            return true;
        }
        int comp = element.compareTo(this.element);
        if (comp < 0) {
            return this.leftChild.add(element);
        }
        else if (comp == 0) {
            return false;
        }
        else {
            return this.rightChild.add(element);
        }
    }
```

Again, note that there are no casts. Without the recursive generics, both recursive calls would require casts with complex parentheses:

```
return ((ZHBinarySearchTree<ElementType>)
          (this.leftChild)).add(element);
return ((ZHBinarySearchTree<ElementType>)
          (this.rightChild)).add(element);
```

We can implement a red/black tree using a similar method, although we do not choose to cover balanced binary search trees in our CS2 course. It is actually possible to implement the red/black tree by extending a regular binary search tree if we implement the binary search tree in two stages. The first stage is an abstract binary search tree that preserves the *StructureType* generic parameter; the second stage extends the abstract binary search tree and eliminates the second generic parameter by passing itself; the concrete class contains only constructors. The red/black tree can then be implemented by extending the abstract binary search tree. Again, we do not recommend such a complicated inheritance structure for the CS2 course, but we present it here to illustrate the power of the recursive generic technique.

5 Experience in the CS2 Course

We have been using recursive generics in our CS2 course for two semesters as of this writing. We introduce the technique very shortly after we cover the basics of Java generics and we use it throughout the course, whenever we implement ADTs built on linked data structures. Students in our labs use the recursive generic structures successfully.

Initially, students don't understand what's going on with the recursive generics—they barely understand generics at all—but we think that's OK. What is important is that students understand the way we use a few underlying data structures to implement a variety of abstract data types and then use those data types to build interesting software. The code that students write is virtually identical to the code they would write if they created a new inner class for each ADT with a linked implementation, but instead of using a potentially error-prone copy and paste, they use inheritance. As we have shown above, without the recursive generics, inheritance would increase the complexity of their code through the need for complex casts. So students get the advantage of code reuse through inheritance along with the code that closely matches the conceptual algorithms.

The use of inheritance here is not just a nice side-benefit; it is practical for the students. They see that a lot of the code that they need for the class they are implementing is already there in the underlying structure. Sometimes the method they need to implement is directly inherited with no need for modification; sometimes they just need to wrap the existing code in a new method; sometimes they need to override a method completely. This approach reinforces their understanding of inheritance in a natural way.

As the course progresses and students do programming projects that use the ADTs they have developed, they come to a fuller understanding of the power of generics and they begin to develop an intuition of what is going on with the recursive generics. We do not ask the students to develop any recursive generic structures, because getting the class declaration right is still beyond their programming skills; rather, we supply all the structures that they need and expect them to learn how to use those structures. This approach pretty much eliminates basic pointer manipulation and instead focuses on building comprehension of the relationships between higher level structures.

We have decided to use our own ADTs rather than the Java Collection Framework in our CS2 course, but we have modeled our interfaces on Java's as much as practicable. We find the Java interfaces are too complex for students to implement, and they often introduce issues which obscure the concepts we want students to learn. Our classes are powerful enough for students' needs in the CS2 programming projects. We expect that students will have little difficulty making the transition to the standard classes in their later coursework.

6 Problems for Further Study

We are beginning to prepare complete course materials, including textbooks, for both the CS1 and CS2 courses. This is a large undertaking and involves the development of many

software components. The linked structures and ADTs we have described here are a major part of this project.

We would like to extend the recursive generic technique to the full range of standard CS2 linked data structures and ADTs as well as to structures that would be covered in more advanced courses. We believe that the technique could be useful beyond its pedagogical benefits. For example, we have done research on phylogeny trees in bioinformatics and expect that this technique will make tree representation much easier. We expect that we can apply this technique with C++ templates, but we have not attempted to do so yet.

We are also investigating array-based data structures in Java. Here, Java generics present a related problem. When we present array-based implementations of ADTs, we would like to use arrays directly, but Java generics don't handle generic arrays. The only practical way to use arrays directly in generic ADTs is to use *Object*[], but this solution requires even more casts than we find with the linked structures. Using an *ArrayList* instead of an array fits better with the generic ADT implementation, but the *ArrayList* already implements most of the methods we want the students to handle for themselves. We are therefore trying to develop a simplified *ArrayStructure* class that will be as much like plain arrays as possible but will handle generics appropriately. We hope to present results of this work in the next year.

We welcome comments or questions from readers. We would particularly like to hear from those who use the recursive generic technique in their own courses or software development. Once again, the complete versions of the classes discussed in this paper, including full documentation are located at www.users.csbsju.edu/~jholey.