Multiple Implementations for Component Based Software using Java Interfaces

Joan Krone
Denison University
Granville, Ohio 43023
krone@denison.edu

Abstract
Current software engineering research points out the benefits of reusing the most difficult part of every project – the specification [3,4,7]. Getting the specification right so that it meets the requirements is a major challenge. Once a correct specification has been written, there may be many ways to implement it, each with different performance characteristics, each fulfilling the specifications. C++ fails to support this idea of multiple implementations of single specifications. The closest we can come is to use a header file as the specification and then place the details in a C++ source file. However, it is not possible to use the same source file for different implementations, since whatever data structure is to be used must be revealed in the header file, forcing the implementation to use it. Here we will see how it is possible to achieve this goal of multiple implementations using Java interfaces. Additionally, we will see how the introduction of generics to Java would increase these benefits tremendously.

I. Introduction
During the past decade, software engineering research has introduced and promoted the idea of component based software, an approach that encourages reuse of carefully designed components with good performance characteristics [1, 8]. To get the good performance, it is often necessary to consider a variety of possible implementations before choosing one appropriate for a given situation. Although C++ does not support multiple implementations for single concepts, Java has taken a first step in that direction. Here we see what that first step is and we examine the possibilities for using this approach.

Anyone who uses Java soon notices the convenience of using abstract classes and interfaces from commonly available libraries. For example, everyone who writes an applet is likely to implement the ActionListener class, an abstract class allowing inputs such as selecting a button or a menu item to be processed. The idea is that the actionPerformed method conceptually is provided by the ActionListener class and every implementation of that class must include code for the actionPerformed method, but those implementations may vary widely, depending on the particular situation.

Certainly one popular reason for using interfaces from a Java library is simply that there exists some functionality that a programmer wants to use, and taking advantage of a library component class saves time by providing the programmer with names and descriptions of the methods to be implemented.

Nevertheless, most existing textbooks written to be used with Java have not noted another way to use interfaces, a way that is particularly helpful pedagogically for teaching about component performance. To illustrate the new approach advocated here, we look to sorting as a topic that
is covered in every student’s undergraduate career, possibly more than once. Certainly, this topic is frequently used as a good way to introduce algorithm analysis. There are so many ways to sort that students get to make comparisons among a variety of algorithms, thereby learning a lot about program complexity. But, what they don’t learn is that it should not be necessary to think of the different methods of sorting as different conceptualizations.

II. Sorting as an Example
A. Current approaches to algorithm analysis treat every sorting algorithm as a completely different idea. Yet, although the way in which these different sorts are performed is very different, and although each has different analysis results, they are all simply implementations of a common concept, that of rearranging data so that it is possible to enter some data and then to retrieve that same data in whatever order is desired.

Just as we tell our students not to start writing code until they have spent some time thinking about their problem, before we look at how we can use Java interfaces, we need to define the problem of putting some given data into a specified order. First, we think of what we need to do in order to achieve the goal of ordering data.

Abstractly, we might think of a sorting machine, into which we put what data we have. When we have entered all the data, we signal the machine that we want to change modes from entering data to the mode of getting it back. Of course, we want to be able to get the data back in a designated order.

With this concept in mind, we choose methods that enable us to do this. We need a method for adding new data elements, another to signal that we have no more data to insert, and one method that allows us to get back a smallest element. We may also need to check whether the sorting machine is in accepting mode, and we may want to know how many elements there are in the sorting machine. Finally, we want to be able to clear the machine so we can start a new sort when desired. Clearing may be associated with a finalize operation.

The following is a Java interface that captures these ideas:

```java
public interface Sorting_Machine
{
    // An Entry Keeper needs to provide a structure for
    // holding data and a boolean for determining whether or
    // not the sorting machine is in accepting mode.
    // Every implementation needs a constructor.

    public abstract void addEntry(int x);
    // Put a new entry, x, into the Sorting machine.

    public abstract void changeModes();
    // Set the boolean to the opposite mode indicating that
    // no more entries may be made, or to accepting mode if
    // more entries may be made.
}
```
public abstract void getSmallest (int smallest);
// Remove the currently smallest entry from the machine.

public abstract int entryCount ( ) ;
// Return the number of entries currently in the machine.

public abstract boolean isAccepting ( ) ;
// Returns true or false depending on whether or not the
// machine is ready to accept more entries or is in the
// mode for giving out the smallest value.
// Each call just reverses the current Boolean
// value of isAccepting.
}

Note that we have chosen to describe a sorting machine for integers, but
other types could be used by making appropriate changes. Here is where
generics would be useful, so that it would not be necessary to identify a
specific entry type until a declaration of a sorting machine is made.

One might argue that the accepting flag might be hidden, yielding an even more
abstract description of sorting. In that case, users could ask for a smallest
element at any time. The reason for keeping the flag in this particular example
is that it allows for more efficient implementations. Without the mode setting,
every time a new element is inserted, it would be necessary to sort the
collection in anticipation that the smallest element might be called for. From a
software engineering point of view, it would be preferable to have a “static”
sorting machine (the one shown here), and a “dynamic” sorting machine which
does permit one to ask for the smallest at any time. In fact, if the dynamic
sorting machine were given, a static sorting machine could be built from it. So,
one might view this machine as built from the dynamic one, thereby allowing
for a greater variety of efficient implementations.

B. A Variety of Implementations
We can use the single interface described for several implementations of sorting.
Here we examine a few, and leave others for the reader to do.

First, we consider what is frequently called “selection sort.” In this approach to
sorting, an array is used to hold the data. After the data has been inserted into the
array, each element of the array is considered starting with the first element. All
elements following the one under consideration are compared with that one, and if
one of those elements should precede the currently chosen one, the index of the new
one is stored. At the end of the loop through the array elements following the
chosen element, whatever element should precede the chosen one is exchanged with
the chosen one. After each element has been treated in this way, the elements are in
order.
To use this implementation with the above interface, we use the addEntry to enter our data. When we call changeModes, the isAccepting flag is reversed and a selection sort is called. Then a loop is used to repeatedly call getSmallest until the array has been traversed and the items have been returned in order.

Here is one such implementation:

```java
// The "SelectSorter" class.
// This class implements the Sorting_Machine interface with
// a selection sort with worst case complexity quadratic in the number
// of integers being sorted.
import hsa.Console; // for doing I/O

public class SelectSorter implements Sorting_Machine
{
    static Console c = new Console();
    int[] keeper;
    boolean accepting;
    int total;
    static int current = 0;

    public SelectSorter()
    {
        c.println("How many integers do you want to sort?");
        total = c.readInt();
        keeper = new int[total];
        accepting = true;
    }

    public void addEntry(int x)
    {
        keeper[current] = x;
        current++;
    }

    public void changeModes()
    {
        Select smember = new Select();
        if (isAccepting)
        {
            smember.selectSort(keeper, total);
            accepting = false;
        }
        isAccepting = true;
    }

    public void getSmallest(int smallest)
    {
        smallest = keeper[current];
    }
}
public int entryCount()
{
    return current;
}

public boolean isAccepting()
{
    return accepting;
}

public static void main (String[] args)
{
    SelectSorter srt = new SelectSorter();
    for (int i=0; i<srt.total; i++)
    {
        c.println("next?");
        srt.addEntry(c.readInt());
    }
}

class Select
{
    static int getMinIndex (int[] v, int first, int last)
    {
        int minInd = 1;
        int i = first;
        while (i < last)
        {
            if (v[i] < v[minInd])
            {
                minInd = i;
                i = i + 1;
            }
        }
        return minInd;
    }

    void selectSort (int[] v, int n)
    {
        int i = 0;
        int temp = 0;
        int minI = 0;

        for (i = 1; i <= n; i++)
        {

In this implementation, a main method has been included along with the implementation of the sorting machine. One could separate out the main method and put it in a class that imports the implementation of the sorting machine. Such a separation leads to a still better software engineering solution, since one could then put the Sorting Machine and the SelectSorter in a library to be used as a programmer might choose.

Since quicksort is considered the best choice for random data, we consider it next. The only difference between the implementation using selection sort and one using quicksort is that in the changeModes method, instead of calling the selection sort, one would call the quicksort.

To see a very different possibility still using the same interface, we turn to a heap sort implementation. Here the appropriate structure is a heap. In this case, the addEntry operation will need to build the heap, so as each new entry is entered, steps are taken to maintain the heap property. This differs from both the quicksort and the selection sort implementations, since in both of those cases, the program uses an array as the structure and the addEntry simply inserts the values into the array.

The changeModes method will simply reverse the accepting boolean. In both of the previous examples, all of the sorting took place in this method, so this is another major difference.

The getSmallest method will need to remove the root of the heap and then reheapify the elements, still another difference from previously described implementations in which the getSmallest simply returned an element.

What is important is that even though these implementations are very different from each other both with regard to what is done and the performance characteristics of the various methods, they all follow the same conceptualization described in a single interface.

Another implementation, different from any of the those shown so far, is an insertsort implementation, one which puts the elements in order as they are entered, so that by the time all the elements have been entered, they are already in order. The usual implementation for insertsort is a linked structure. In this approach, the addEntry method will find the appropriate place in the list for each new integer and insert the item at that position. changeModes will simply reverse the boolean, and getSmallest just exports the next element on the list.
You can now choose your favorite sorting technique, or any other, and figure out how you can implement the given interface using that technique, possibly introducing still another set of performance characteristics.

### III. What Generics Can Do

These examples show how a single Java interface can serve as the specification for many sorting algorithms, each of which can be written as an implementation of the Sorting_Machine interface. Since Java does not have generics, we are forced with two choices for writing the interface, neither of which is a good one.

One choice is what we have done above, namely, fixing the type of elements to be sorted in the interface. This is not a good choice because if we want to sort some other type of entry, such as real numbers, another interface must be written.

The second choice is to write the interface in terms of Objects. In this case the entry type is Object. However, this choice also fails to be optimal, because many of the types that one might want to sort are not Objects. None of the primitive types, such as integer, float, and double, are Objects. Each is associated with a class that does have member Objects, but these Objects differ from the primitives. For example, the primitive integer type, `int`, is associated with the class called `Integer`. Objects that are members of the Integer class may call a variety of methods such as one that makes it possible to convert from a string to an integer, but Integer Objects cannot do simple integer arithmetic and comparison and hence cannot be sorted unless first converted back to primitive integers.

If Java were to introduce generics, then one could write the interface in terms of some generic class, thereby making it possible to use one interface for sorting any entries that have a comparison to use with them. Of course, here is still another problem for making the interface completely generic. Some items we might want to sort cannot be compared with a simple “≤.” For example, one might want to sort some records and the comparisons would be made on some particular field or fields in the record. Unless the fields to be used happen to be primitives, it might be necessary to have some other comparison. For such a general sorting situation, the interface could include a method for comparing two entries. Then an implementation would provide whatever comparison action would be appropriate.

### IV. Teaching

Where might this idea of single interfaces with multiple implementations be introduced in the undergraduate curriculum? Of course the answer depends on what language students use in their introductory course(s) as well as when students learn to analyze and compare sorting algorithms. For example, if students use Java in their introduction to programming course, they will have some idea of what an interface is, even though they may not write their own interfaces in CS1.

The Decker and Hirshfield text, *Programming.java* [2], explains interfaces to beginning students so they can implement some when they write applets. With such a background students could benefit from seeing a sorting interface in their CS2 course when they are
studying a variety of sorting algorithms. This approach encourages them to make comparisons and to see how different performance can be achieved for the same concept.

If students have not studied Java in their early courses, this idea might be introduced in a theory of programming languages course, since they are likely to examine Java in such a course. There, interfaces would be a natural topic for exploring how Java differs from whatever language the students have used previously, probably C++. Here interesting comparisons between C++ header files and Java interfaces would be instructive.

Still another course in which these ideas might be important is software engineering. Texts such as Pressman [5] and Sommerville [6], emphasize the importance of specifications. One could include an assignment in which students needed to consider a variety of sorting implementations depending on application domain. For example, one might prefer a merge sort for managing database records, but choose a quicksort for random data.

V. More Than Just Sorting
Although sorting was chosen to illustrate the idea of multiple implementations for a single specification, this software engineering principle is applicable in every situation where there are multiple ways to achieve a single goal. Some additional examples include a single stack interface with one implementation for small stacks using an array and top index and another implementation using a linked list structure.

A single interface describing a queue may have one implementation using a circular array and another using a linked list.

A coalescable equivalence relation (often called union – find) specification allowing elements to be made equivalent and a check to see if two elements are equivalent may have at least four implementations, each appropriate for a particular situation. For cases where one builds a single equivalence relation and then repeatedly check for equivalence of pairs of elements, a two dimensional array would be a good choice of implementation.

On the other hand, if one may be doing a lot of making pairs of items equivalent, as well as checking them for equivalence, one might use an array of root pointers as an implementation. Here, each equivalence class can be thought of as a tree with a representative as the root.

One could also use lists of equivalent items or an array of parent pointers.

This is just a small collection of single ideas that have multiple interfaces, an approach to doing design and implementation that follows well known principles of software engineering and promotes the importance of performance comparisons between and among various implementations.
VI. A Vision of Future Software

We can now look toward a future in which software engineering, like other engineering disciplines, will have some standard off the shelf components that are well designed and that have a variety of performance options to choose from. An interface (not necessarily a Java one) will be what the user sees to describe the “what” of the software. Performance information from each component will be given, so that an appropriate choice can be made by the client.

Java allows us to take one step toward that goal and to acquaint our students with this idea.
VII. References