

Using Interface Inheritance to Structure the Data Structures Course

Dr. Dale E. Parson and Dr. Daniel Spiegel

Kutztown University of Pennsylvania
P. O. Box 730
Kutztown, PA, 19530, USA

Abstract

Traditional instruction in the first data structures course treats object-oriented inheritance and polymorphism as isolated topics if at all. Course organization is an exploration of the plethora of commonly used implementation structures, along with analysis of their run-time costs and benefits. This paper examines an alternative organization of this course that uses inheritance and polymorphism as the primary means for organizing presentation and student projects in implementation, testing and application of data structures. Abstract sequence, map and set interfaces provide a conceptual and code-concrete framework within which students implement and extend arrays, linked lists, trees, hash tables, iterators and other structures. Reuse of student design and code modules fits seamlessly into this organization, saving time and effort and reinforcing the utility of reused classes. Achieving working exposure to type-generic container classes and class libraries is straightforward. The initial offering of a data structures course using this organization has been a success.

Keywords: data structures, generic class, inheritance, object orientation, polymorphism, reuse.

1. Introduction

Traditionally there have been two approaches to organizing the undergraduate data structures and algorithms course, considered by many computer science educators to be the keystone of undergraduate education in the field. The most popular is a *structural approach*, in which the implementation aspects of data structures and algorithms serve to organize course topics [1-4]. Each implementation structure such as a linked list, tree or hash table acts as a primary focus for study. Elaboration of implementation explores contributing language features such as array index calculation, dynamic storage management and pointer manipulation. Elaboration of application explores tradeoffs in applying data structures to standalone, monolithic problems. Given the fact that any reasonably small, course size programming project is

unlikely to include multiple data structures studied in the course, there is little opportunity for reuse among projects. Most projects constitute one-time, throwaway work.

The second traditional approach is the *analytical approach*, in which study concentrates on mathematical analysis of the performance properties of algorithms and associated data structures, particularly the properties of growth rates in execution time and memory space as functions of growth rates in the size of processed data [5-6]. Most computer science programs present the structural approach in a second year data structures course and the analytical approach in a subsequent course on analysis. The ACM curriculum guidelines recommend this two step process [7].

Regardless of approach, data structures courses have changed over the years because new topics such as objects, inheritance, and standard libraries have been added to the syllabus. While object-oriented design and prefabricated data structures may reside at some distance from the focus of traditional data structures courses, including these topics better serves students because introduction and reinforcement of proper design is important at the time most students take this course during their sophomore year.

With the inclusion of object-oriented programming and associated topics, traditional data structures topics such as graphs, trees (beyond binary), heaps and sets have been minimized or removed from the first data structures course, moving to a second data structures course offered as a junior or senior level elective. Analysis is often split between that second data structures course and an algorithm analysis course or moved entirely to the analysis course.

Many of the newer topics in the course do not require as much depth of coverage as was required for the subjects minimized or moved to advanced courses. These changes provide an opportunity to teach a big picture course to sophomores at a time when they are still gaining knowledge of the principles of design. Inheritance and polymorphism can be used to advantage to assist students in enhancing overall perceptions about design.

These enhancements create a third way to teach data structures. This paper presents an *abstraction approach* that uses object-oriented mechanisms of interface inheritance and polymorphism as organizing principles from the very start of the first data structures course. While textbooks and course offerings that take the structural approach usually teach abstraction as an abstract perspective on applying data structures, for example applying the concept of an abstract data type (ADT) as a veneer over a data structure such as a binary tree in order to hide certain implementation details [1], and while most curricula present or extend the concepts of object-oriented inheritance and polymorphism within this course, no popular approach to the course uses abstraction in the form of concrete interface inheritance – “concrete” in the sense of using object-oriented inheritance language mechanisms starting with the very first programming project – as a means for organizing the focus of study. Using interface inheritance and polymorphism to organize the content and order of delivery of course topics makes the concept of an ADT a concrete rather than an abstract concept for students. They build and extend software modules using ADT-implementing mechanisms as concrete actions from the start of the course. These language mechanisms provide a framework for comparing the common and distinguishing features of data structures and algorithms that implement codified abstract interfaces, where features include both structural features and run-time complexity characteristics. Finally, these language mechanisms provide opportunities for module reuse across course projects.

This paper reports on the presentation of the data structures course using this abstraction approach in the Fall, 2008 semester at Kutztown University of Pennsylvania [8]. The programming language for presentation and programming projects was C++; this approach applies equally well when using Java or another language that supports interface inheritance. The course also made use of template classes (a.k.a. generics) in a subset of container projects, it required reuse of code across several projects, and it used the C++ Standard Template Library (STL) [9] in several projects. This organization of the course did not crowd out any traditional topics in the departmentally approved course syllabus. The applicability of interface inheritance and polymorphism to the presentation of traditional course material compensated for the cost of its insertion at the start of the semester.

2. Interface-driven Pedagogy

2.1 Historical influences

The three interfaces that guided this offering of the data structures course are *Sequence*, *Map* and *Set*. The “List” ADT in Aho, Hopcroft and Ullman [3] serves as the

starting point for *Sequence*, while `java.util` interfaces [10] serve as inspirations for *Map* and *Set*. The instructor introduced data structures as embodiments of these interfaces, using interfaces coded by the instructor as abstract C++ classes in lectures and student projects. All interface method declarations take the form of pure virtual C++ member functions, which are function declarations devoid of implementation. Pure virtual C++ functions are equivalent to operations in a Java interface, and a C++ class composed strictly of pure virtual functions and constant data field declarations is roughly equivalent to a Java interface. The term *C++ interface* as used throughout the remainder of this paper refers to a pure, abstract C++ class used similarly to a Java interface.

One additional historical influence is the software development history of the instructor using object-oriented analysis and design techniques within industrial projects. Mechanisms that help to focus design efforts and to structure implementation work turn out to be useful mechanisms for structuring course delivery as well.

2.2 Introduction to interface inheritance

The course commenced with an introduction to interface inheritance via a code example. The instructor first illustrated a concrete C++ class with one member function that prints a string parameter to a file. This class did not use inheritance. The instructor then presented a refactoring of this code as a class hierarchy with a C++ interface in a header file that specifies operations, and a concrete C++ class declared in a header file and implemented in a source file that implements the operations of the interface.

This initial body of code supports introduction of several essential concepts and mechanisms. First, the practice of refactoring a body of code makes an early entry into the course. Second, using a C++ interface as a concrete barrier for separating implementation concerns in concrete subclasses from application concerns in client classes makes an early appearance. The concepts of module decoupling and information hiding first appear in concrete form. Interface lecture topics also include preconditions on client-supplied input parameters and postconditions on implementation-supplied output parameters, return values and object state.

The first project also provides an opportunity to introduce makefile-driven compilation, linking and regression testing [11]. Students will have had some introduction to separation compilation and linking. The form of course projects as distinct C++ interfaces, concrete implementation classes, and client test driver modules mandates that students have a thorough grasp of these processes, along with tools that automate their execution. Standards on file naming conventions minimize the need for students to become experts in creating makefiles, although addition of student tests to makefile-driven testing is a requirement for every project.

2.3 The Sequence interface

Figure 1 shows the Unified Modeling Language (UML) class diagram for *sequence_interface* and its two initial concrete derived classes. The interface declares operations that *insert*, *get* and *remove* string elements according to position, and a *size* operation that returns the number of elements contained in a sequence object. Boolean return values indicate success or failure of an operation. This initial class hierarchy, supplied by the instructor, limits itself to sequences of strings in order to defer template class topics until later in the course. The course introduces Sequence before other interface abstractions because incoming students are already familiar with positional arrays.

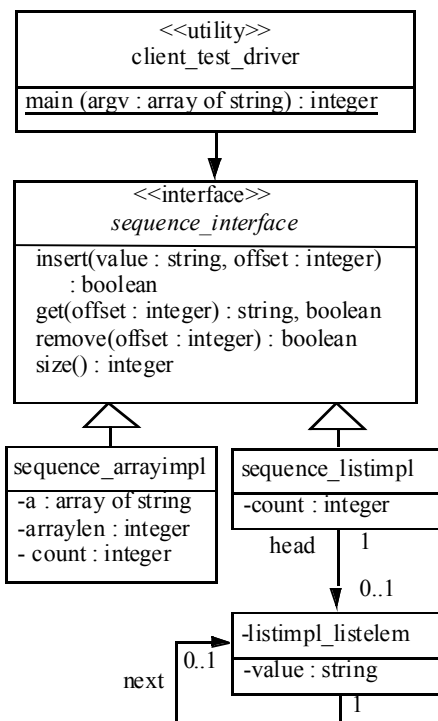


Figure 1. Initial class diagram for *sequence_interface*.

The two concrete classes of Figure 1 provide array-based and singly-linked list-based implementations of the interface. Both classes reinforce the practice of using private access visibility for data members as part of encapsulation. The list class also adds a private helper structure type, *listimpl_listelem*, that is the content-bearing linked data structure that constitutes the traditional, structural presentation of linked lists without regard to encapsulation. With two such disparate implementation classes, separation of an ADT's interface from specific data structures becomes clear. Finally, the presence of the *count* field within the linked list class, while not strictly necessary, makes it possible to test for illegal offset parameters without traversing a list in order to count it.

This small optimization provides an opening topic for discussion of the benefits of encapsulation for optimizing fields and functions that are not normally part of the raw underlying data structure.

Figure 2 shows a student-supplied doubly-linked list class *sequence_dlistimpl*. Students create their doubly-linked class by migrating the singly-linked class of Figure 1. In addition to adding a tail pointer to the class and a previous pointer to the helper structure, students add a mid pointer to the class that tracks the most recently accessed element in the list. The mid pointer provides an optimization when client code iterates over a list in single steps, since access to the element at mid's offset requires no additional traversal through the list. Implementation of mid is the first step in the introduction of iterators, because mid is an iterator-like optimization for sequential traversal of a list.

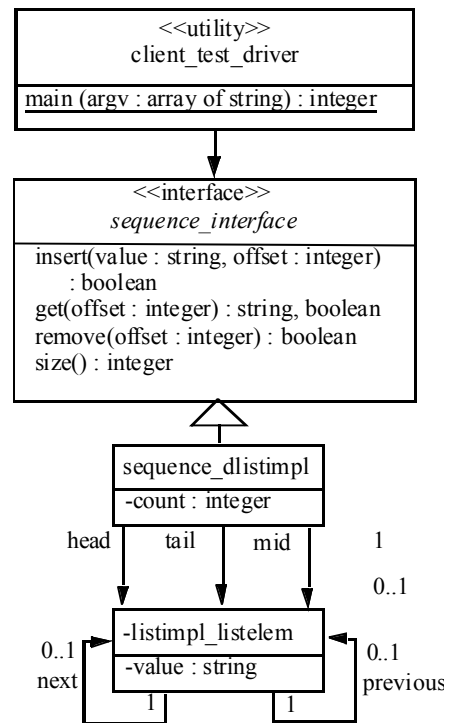


Figure 2. Student doubly-linked list assignment.

In addition to introducing doubly-linked lists and iterators, this assignment reinforces refactoring by having students migrate an existing, encapsulated class. It adds optimization opportunities via the mid pointer that are not present in raw, unencapsulated doubly linked lists. In addition to interface reuse and source code reuse in migrating the singly-linked class, students obtain the benefits of test case reuse. All makefile-driven tests supplied by the instructor for the concrete classes of Figure 1 also apply immediately to the student concrete class of Figure 2 thanks to interface inheritance. Each student has a

means for testing the assignment class as soon as it compiles. Once these tests pass, students must add tests for possible failure modes of doubly-linked lists.

2.4 STL and implementation inheritance

One additional instructor-supplied implementation class for `sequence_interface` is `sequence_stl_vectorimpl` that uses the STL vector class. This example introduces template classes, STL container classes and STL iterators in a now-familiar context.

All of the concrete classes to this point include element `count` fields, `size` functions that return count, and identical verification code for `offset` parameters that determines whether offset parameters are within range of sequence size, and that convert negative offsets from the end of a sequence to nonnegative offsets from the start. All three positional functions – insert, get and remove – in the implementation classes use identical copies of this offset verification code, resulting in twelve identical copies of verification code for four classes. At this point the instructor has a concrete example for discussing the maintenance problems of proliferation of multiple copies of originally identical code. The traditional solution for procedural programming would be to consolidate a single copy of the verification code in a helper function called by all three operations. In the current example, however, this idea gives rise to the question, “In what class should we place the helper function?” Given the encapsulated nature of the classes of Figures 1 and 2, the solution to the problem is to use implementation inheritance to create abstract class `sequence_abstract_baseclass` of Figure 3 that resides between `sequence_interface` and the concrete classes of Figures 1 and 2 in the inheritance hierarchy. This abstract class provides implementation inheritance of the protected `count` field and the public `size` method. It also provides protected helper function `getRealOffset` that validates offset parameters and converts negative offsets to nonnegative values, thereby eliminating redundant validation code. This field and these operations are useful for all implementation classes of this interface, and so they provide a non-contrived means for initiating students to applications of implementation inheritance.

Several of the students in the data structures course were also taking a Java programming course given by the instructor, presenting another opportunity for design reuse. The instructor ported this interface and implementation class hierarchy to Java, thereby giving students enrolled in both classes some language-independent reinforcement of these concepts.

2.5 Template containers and client-side reuse

Work with the design and use of template classes is necessary in the data structures course because many of the data structures and their encapsulating classes are containers, and container classes achieve maximum reuse

via parameterization of the contained object type. In the initial step of designing template classes the instructor recoded the class hierarchy of Figure 3 as a template class hierarchy, replacing the string element type with template element type `ElemType`. This new hierarchy provides an incremental means for introducing template class design to students.

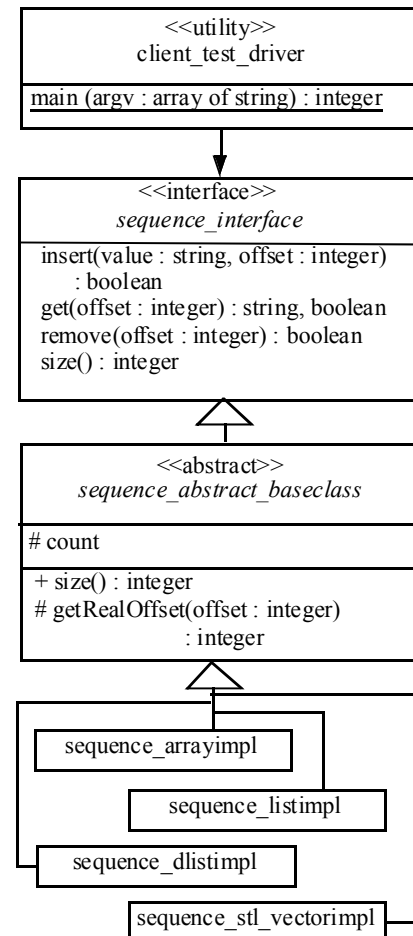


Figure 3. Implementation inheritance and STL added.

The next student assignment was the design and implementation of concrete template classes `deq_seq` for double ended queues, `stack_seq` for stacks, and `queue_seq` for queues as shown in Figure 4, along with construction of a test driver and test data. The constructor for a `deq_seq` `<ElemType>` object takes a `sequence_interface` `<ElemType>` object as a constructor parameter, and uses this object for `storage` of the double ended queue. The constructors for `stack_seq` `<ElemType>` and `queue_seq` `<ElemType>` objects take a `deq_seq` `<ElemType>` object as a constructor parameter, using subsets of the `deq_seq` operations to implement stack and queue operations respectively. Client test driver code first constructs appropriate `sequence_interface` `<ElemType>` objects for

storage, and then constructs the type-bound `deq_seq`, `stack_seq` and `queue_seq` objects for testing. A student design session concluded that `sequence_dlistimpl<ElemType>` is appropriate for the storage class, since it allows the double ended queue to use both ends of the storage object in $O(1)$ time.

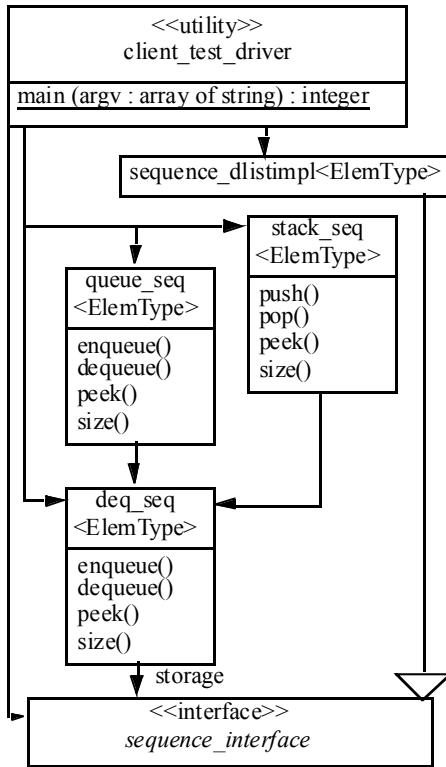


Figure 4. Students template classes and project reuse.

Students were surprised and delighted to find that there was very little code to write in order to implement the operations of their three container classes of Figure 4. The most difficult conceptual part was comprehending that they did not need to do much work! The double ended queue class uses a `sequence_interface` object to do most of its work, and the stack and queue classes rename some operations of the latter class while hiding others. Building on previous projects was an ideal working example of reuse. In addition to saving time and effort, it reinforced concepts that went into designing the `sequence_interface` template class hierarchy from a client perspective. The most difficult part of implementing this project was deciphering the obscure error messages coming from the `g++` compiler [12] for errors in template class implementation and use. The compiler also produced object code for template classes that would not work with debuggers. The solution for the first problem was to find another compiler with somewhat better error diagnostics. The students wound up using `print` statements for debugging; fortunately, the container classes are not

complicated, so this approach was sufficient. Because of the time involved in using the available tools for debugging interdependent template classes, the class went back to building non-template classes for subsequent projects. We did continue to use STL without excessive trouble from the compiler.

2.6 Sorting algorithms, data views and reuse

Coverage of sorting algorithms took a break from designing class hierarchies, but it did not take a break from concrete mechanisms for abstraction. We used the following non-contiguous C++ statements in support of parameterized sort functions.

```
typedef void (*sortfunction)(int iarray[], int left, int right);
```

```
sortfunction sortfunc = NULL;
```

`sortfunc` = address of a sort function based on a command line argument.

```
(*sortfunc)(intarray,0,size-1); // call to the sort function
```

A `sortfunction` sorts elements within `iarray` from index `left` through `right` inclusive. Student test driver code selected from among a standard set of sort functions as determined by a command line argument, invoked the parameterized function and printed the resulting array of integers.

Students were able to reuse code from earlier exercises by constructing and manipulating `queue_seq<int>` objects that use `sequence_arrayimpl<int>` objects as overlays on integer arrays to be sorted by an iterative version of mergesort. Students constructed a `queue_seq<int>` object as a *view* of an integer array to be sorted, and then passed this queue as a parameter to mergesort. Mergesort constructed two additional temporary `queue_seq<int>` objects, initially empty, to use in its split phase, and it merged sorted subsequences from these split queues back into its merge queue parameter. Mergesort works entirely with a first-in first-out (FIFO) queue abstraction of an array. It does not need to manipulate elements in other positions in an array being sorted. Requiring students to construct mergesort based on a queue `<int>` view of an integer array reinforced earlier mechanisms via reuse, and introduced the concept of a client *view* of the application of a data structure. It also gave us an opportunity to discuss an alternative design of a queue that uses a sequential file implementation in support of external sorting.

2.7 Maps, sets and the design of iterators

Figure 5 gives the final class hierarchy used in this offering of the data structures course. Interface `map_interface` specifies mapping operations `put`, `get`, `remove` and `size` for a string `key` : integer `value` mapping, and interface `map_iterator` specifies operations of an iterator for a `map_interface` object. Each `map_interface` object can construct zero or more `map_iterator` objects as

directed by client code. These two interfaces allow the introduction of the object-oriented concept of class *covariance*. A concrete class that implements `map_interface` must pair with a concrete class that implements `map_iterator`, because an iterator must be capable of traversing its map object's data structures. This pairing in class derivation is a form of covariance. Covariance also provides a realistic example of using the *friend class* construct in C++ or *package level protection* in Java, since a concrete `map_iterator` class must have access to data structure definitions within its covariant `map_interface` class.

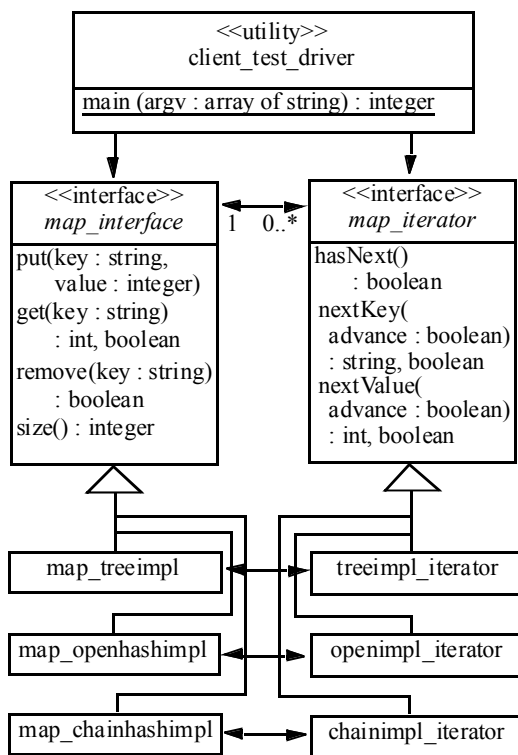


Figure 5. Covariance in Map classes and Iterators.

The instructor presented an equivalent Set class hierarchy as a blackboard example of a Map with only keys and no values. There was no coding on behalf of Set.

The instructor supplied the design of Figure 5, along with much of the code, and students supplied selected functions and test data. For the binary tree classes `map_treeimpl` and `treeimpl_iterator`, students wrote all public and private functions of class `treeimpl_iterator` in support of incremental traversal of tree nodes, one per call to `nextKey` or `nextValue`. Students worked from instructor-supplied pseudocode in building a left-to-right, inorder iterator class. Several students successfully tackled bonus project options for traversing the tree in right-to-left, preorder and postorder traversals. While these traversals are easy to implement for a monolithic scan of an entire

binary tree, appearing in every data structures textbook, implementing them in an iterator class that must use state variables and a stack to keep tracks of its current and upcoming tree node locations is a challenging exercise. Students used an STL stack to keep track of the iterator's progress through the binary tree.

Instructor-supplied classes `map_openhashimpl` and `openimpl_iterator` introduced open address hash tables, where `key : value` mappings are stored directly within hash buckets in an array. Example code provided a framework for discussion hashing functions, collisions, and other typical hash table structures and operations. The supplied iterator class skipped over unoccupied buckets in its traversal of its table array. An optimization appearing on the final exam had students construct a secondary, index array for occupied buckets in order to improve iterator performance for sparse tables.

Students coded the `remove` function for the chained hash table class `map_chainhashimpl` that manipulates an STL *list* constituting a bucket's chain of `key : value` mappings. They also wrote a demonstrably improved (via testing) hash function using C++ bitwise and shift operators.

3. Results and Conclusions

The instructor has taught the data structures course a number of times using a more traditional organization. This paper describes the first offering using this object-oriented approach. The motivation in using this approach was not one of turning the data structures course into a course on object-oriented design. Rather, the motivation was one of finding a more effective perspective and framework for presenting traditional material, while introducing inter-object data structuring and programming language mechanisms for inheritance and polymorphism to an appropriate degree.

If student satisfaction is a fair indicator, the course was a success. 100% of attending students rated the organization of material as very good or excellent, and 100% rated the instructor as excellent.

Retention rate and student success rate in the course were typical. However, these results are notable in view of the fact that the course introduced significant conceptual material not present in a traditional, structural organization. The only traditional topic for which this class did not have time was an introduction to graph algorithms and structures, and the computer science department had already decided to move this topic to a more advanced course. Significantly, the inclusion of object-oriented concepts, constructs and assignments did not crowd any of the desired data structures topics out of the course offering.

The conclusion is that inclusion of this object-oriented material did not displace conventional data structures topics because this organization helps with the

presentation and student comprehension of traditional material. Interface inheritance provides a useful perspective for teaching separation of implementation of data structures from their use. This separation allows the class to examine the tradeoffs in using one structure versus another in an explicit, modular manner. Polymorphism is similarly part of this perspective on alternative implementations of a single interface.

Furthermore, these two object-oriented constructs allow students to reuse class hierarchy code and client test drivers across projects. Code reuse both within inheritance hierarchies and across classes such as the deque – stack – queue example helps students save time and reinforce concepts. Students enjoy learning to write code that they can reuse rather than throw away after an assignment, adding an affective dimension to the course. Reuse also saves instructor time and effort in creating presentations.

Incremental refactoring of a code base is an important practice to learn by working on a series of class hierarchy transformations. Reading another person's code – in this case the instructor's examples – determining portions to change, making changes and verifying their effectiveness are all important skills for computer science students to acquire.

Given the fact that it was possible to keep all planned traditional data structures material in the course while adding this object-oriented material in a manner that catalyzes student comprehension rather than taxing it, there is much to recommend this approach. This approach does require more planning than a traditional structural approach in order to ensure the fit of interrelated object-oriented components. Much of that planning is reusable. Student programming assignments must vary from semester to semester, but the same statement is true of the traditional structural approach. Upcoming plans include future offerings of introductory and advanced data structures courses using this approach, possibly with a textbook to follow.

References

- [1] F. Carrano, *Data Abstraction and Problem Solving with C++*, Fifth Edition, Pearson / Addison-Wesley, 2007.
- [2] M. Weiss, *Data Structures and Algorithm Analysis in C++*, Third Edition, Pearson / Addison-Wesley, 2006.
- [3] Aho, Hopcroft and Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983.
- [4] N. Wirth, *Algorithms + Data Structures = Programs*, Prentice Hall, 1976.
- [5] C. Shaffer, *A Practical Introduction to Data Structures and Algorithm Analysis*, Second Edition, Prentice Hall, 2001.
- [6] P. Purdom and C. Brown, *The Analysis of Algorithms*, Holt, Rinehart and Winston, 1985.
- [7] Association for Computing Machinery, *Computer Science Curriculum 2008, An Interim Revision of CS 2001*, November 15, 2008, <http://www.acm.org/education/curricula/ComputerScienceCurriculumUpdate2008.pdf>.
- [8] Kutztown University, Department of Computer Science, <http://cs.kutztown.edu/>, February, 2009.
- [9] Silicon Graphics, Inc., *Standard Template Library Programmer's Guide*, <http://www.sgi.com/tech/stl/>, 1993-2006.
- [10] Sun Microsystems, *Java Platform, Standard Edition 6 API Specification*, package java.util, 2008, <http://java.sun.com/javase/6/docs/api/index.html>.
- [11] Free Software Foundation, *GNU Make Manual*, <http://www.gnu.org/software/make/manual/>, April 1, 2006.
- [12] Free Software Foundation, *GCC Online Documentation*, <http://gcc.gnu.org/onlinedocs/>, 2008.