Application-based Modules using Apprentice Learning for CS 2 [*]

Owen Astrachan
Duke University
ola@cs.duke.edu

Robert Smith
North Carolina Central University
rfs@sci.nccu.edu

James Wilkes
Appalachian State University
jtw@shark.cs.appstate.edu

Abstract

A typical Data Structures (CS 2) course covers a wide variety of topics: elementary algorithm analysis; data structures including dynamic structures, trees, tables, graphs, *etc.*; large programming projects; and more advanced object-oriented concepts. Integrating these topics into assignments is a challenging task; educators often duplicate work done by others in re-inventing such assignments. At the same time, these assignments and large programs take time to develop and are often changed from semester to semester to preclude cheating. We report on a project that provides modules containing many kinds of programming and lab assignments which can be re-used across semesters with accessible and exciting application-oriented materials. Our project is a collaboration between a research and teaching oriented private university, a teaching oriented public university, and a teaching oriented historically black university. This helps ensure that the modules will be accessible to nearly all student populations. The modules developed are available electronically as hyper-text documents.

1 Introduction

In this paper we report on collaborative work to develop learning modules that support an application oriented, apprenticeship learning approach to the CS 2 course. Each module supports a methodology that is inverted from the traditional "learn a data structure and use it in a program" approach. Rather than studying data structures to use in applications, we use applications to study data structures. Each module supports an apprentice style of learning wherein students use and modify implementations of data structures and application programs before writing them from scratch. Modules supply code libraries and frameworks, explanatory material for students and instructors, and potential programming, lab, and homework exercises. The modules permit instructors to re-use related assignments from semester to semester with less worry that students will re-use work from previous semesters. Although face-to-face grading can alleviate this kind of "code re-use", not all institutions have the resources for grading assignments in this manner. As an alternative, we have developed modules that allow instructors to require implementation of different pieces, e.g., classes, functions, or code frameworks, in different semesters. Large and captivating assignments require significant resources to prepare. Our project attempts to address this issue in the context of CS 2. Similar projects address these issues in a Compilers Course [2] and an Operating Systems course [5].

Each module supplies a situated learning experience [12] that engages students from the outset by providing practical and illuminating examples for how and why data structures are used. Applications are graphical where appropriate, integrate several data structures into real-world programs, and provide a context for studying large-scale programs. Modules are developed to support object-oriented (using C++ and Java) design and programs, but each module will be developed as an Abstract Data Type (using C) to reach as wide an audience as possible.

1.1 Similar Approaches

The two key components of our project are (1) application-based learning and (2) apprenticeship learning [4, 3, 13]. Using applications as the context in which data structures are studied is related to what is termed a "top-down" approach [15, 10]. The top-down approach has students using data structures such as stacks and queues before implementing them. The data structures are studied and used from the specification of their interface. After students have developed programs using these data structures, different implementations are studied. Our project is related, but differs in that students are expected to study an initial application, then reason about different methods of implementation before developing a final application. The application domains used for modules as part of this project are more complex and use a more mainstream language than that developed and re-

[*]This work is supported in part by the National Science Foundation Grant #DUE-9554910

ported in [15].[1]

Situated and apprentice learning in the CS 1 course is described in [12]. Their approach uses an artificial-life microworld as a means of introducing students to programming. In our project, applications are from a broader domain and are based on exploiting concepts and topics covered in the CS 2 course. Similar work in the CS 1 course is discussed in [4].

Apprenticeship learning is related to the idea of a case study. Case studies have been the principle method of study for many years in business and law schools and are becoming more prevalent in Computer Science courses [7]. Defined there, case studies have the following attributes.

> A case study describes a programming problem, the process used by an expert to solve the problem, and one or more solutions to the problem. Case Studies emphasize the decisions encountered by the programmer and the criteria used to choose among alternatives.

Our materials can be used as the basis for case studies, but the intent of our project is to supply instructors with materials rather than to supply students with the explanations behind a full-fledged solution to a problem.

1.2 Project Plans

Early versions of some of our modules were used in the 1995-1996 year at Duke and NCCU. All three institutions in the project are using drafts of the modules during the Fall 1996 semester. In some cases, described below, the material in the modules has been used in previous semesters, and refined and packaged as modules as part of this project. Incremental improvements will be made for use of the materials in the Spring 1997 semester. All materials are available, and will continue to be available, via the world-wide web. We will refine and finalize the modules in the summer of 1997 based on comments from users at other institutions, our experiences, and comments from our students. We anticipate that the project will result in seven to ten modules with web-pages for both students and instructors. Answers to all assignments will be accessible only to instructors using a password mechanism. This means, for example, that header files will be available for classes and ADTs, but that implementations of each class will be available only to *bona fide* instructors.

In this paper we provide an overview of three of the modules currently accessible and a brief introduction to other modules and supporting materials. All materials are accessible via the web at `http://www.cs.duke.edu/~ola`. All modules are based on an application that uses data structures rather than on a specific data structure. In Section 2 we discuss automatic scoring of bowling, in Section 3 solitaire card games form the basis of the module, and in Section 4 data compression is the application. We discuss some of the other modules in Section 5.

2 Bowling

Most students are familiar with the game of bowling, and many bowling alleys have automated scoring. A computer keeps track of whose turn it is to bowl, the score for each ball, and a cumulative total score and displays this information on an overhead screen. Players bowl a ball only when the computer indicates it is their turn. For a detailed discussion of the different approaches to automatic bowling scoring see [6].

This seemingly simple task provides a wide range of difficulty levels as a late semester CS 1 or early semester CS 2 assignment. Different institutions cover topics at different times in CS 1 and CS 2. In addition, many institutions introduce C++ in CS 2 after a semester of programming in another language. As a result, the evolution from arrays of records to vectors of objects that might mirror a transition from Pascal to C++ is a topic for discussion in a CS 2 course. At Duke, for example, we offer a course for students with programming experience earned at other institutions, in high school, or as self study, that is a combination of the last part of our CS 1 course and our CS 2 course.[2] The first programming project in this combination course typically uses a vector of objects to introduce students to the object paradigm and C++. The bowling module provides ample material for this kind of project and for projects straddling the fuzzy border between CS 1 and CS 2.

The approach we use in this module requires one-dimensional arrays, with the option for two-dimensional arrays, or arrays of objects that include arrays, for multiple-player automatic scoring. By encapsulating the appropriate scoring functions as member functions (or functions to access ADTs in C) students can be asked to write different parts of the automatic scorer in different ways from semester to semester.

We are using this module at two institutions in the Fall of 1996, one using an ADT approach with C, and the other the aforementioned combined course using C++. We ask students to read scores from files and to generate scores on-the-fly using a skill-level in conjunction with a random number generator. We provide a high-level design for the program and students must determine how to compute cumulative scores in the framework of the design provided. The object-oriented version of the program uses several design patterns [9] that we believe will begin to play a more major role in introductory computer science courses [18]. We use the observer/observable pattern to show two graphical views of a bowling match, these are shown in Figure 1.

Other assignments may be based on requiring single or

[1] Of course using C++ is not without controversy, but many schools currently use C++ and C, especially in CS 2. The work done in [15] uses Eiffel. We are porting several of our modules from C++ to Java.

[2] The course has a closed lab providing more time for students to master the extra material. Our CS 2 course does not have a lab whereas our CS 1 course does.

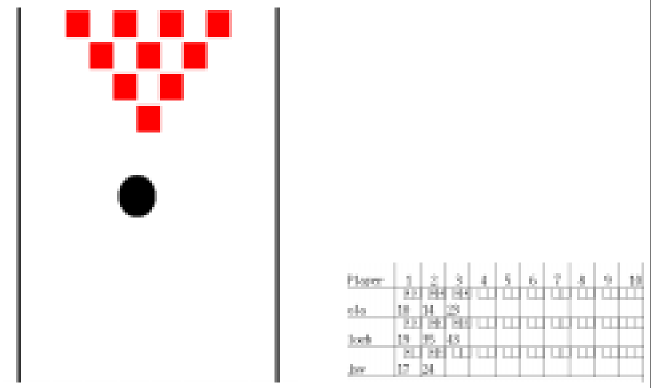

Figure 1: Bowling Observers

multiple-player scoring; providing a scoring solution and asking students to provide the I/O routines or the random ball-generating functions; simplifying or complicating the scoring rules e.g., changing the number of frames or the rules for balls thrown in the last frame.

3 Solitaire

One of the richer domains we use for modules is solitaire card games. There are hundreds of different versions of the same simple idea [14], each providing a new programming problem using the same set of primitives. Solitaire, in some form, is familiar to almost every student. Given some degree of common background, new forms of solitaire can easily be explained. Furthermore, there are versions of solitaire that employ lists, stacks, trees, and graphs. Students can be asked to design and implement a game using classes supplied with the module or to implement one or more of the classes needed as determined by the goals of the instructor. Finally, when the student finishes a project, the game they've created is fun to play.[3] In related work, a case study describing solitaire and programs for playing solitaire well is included in [8]. A module for the card game War is described in [3].

3.1 Development of the Module

The concepts and original programs that form the basis of this module were originally written in Turbo Pascal, without objects, as part of a collection of tools used to motivate CS 2 students [16]. The code was updated to use classes and objects in Turbo Pascal the following year. Students were excited enough about writing games to request the next assignment before it was time to be given, an extraordinary request. Comprehension of the topics covered by the assignment was good.

The programs have now been redesigned and implemented using C++ as part of this project and will be used in class in the Fall of 1996. The graphical components of the software are currently text-based, but encapsulated in a few methods. The classes have been designed to be platform independent and will be used by all institutions that are part of the project in the Spring of 1997. The container classes (described below) are implemented using standard stack and vector classes that are part of the materials available for use in Advanced Placement Computer Science courses and publicly available [1].

3.2 Class Design

The basic class used in all solitaire games is *PlayingCard*. The container classes include *DisplayStack* and *SpreadList*.

A *PlayingCard* has instance variables that describe its location, suit and value, and whether it is face up or face down. Methods for *PlayingCard* include functions for moving, displaying, flipping, and operators for equality, predecessor, and successor.

The class *DisplayStack* uses a stack for storing objects; the class is used in the solitaire application to represent a deck of cards, a discard pile, or the face-down stacks of cards that sometimes occur in solitaire. It has instance variables to hold its location and methods to display the top card of the stack as well as standard stack operations. The class is templated so that any object that has location and display capabilities can be stored in a *DisplayStack*.

The class *SpreadList* uses a vector[4] for storing objects; the class is used wherever cards are "fanned out". For example, in the tableau of a solitaire game the Ten of Diamonds is placed on the Jack of Clubs which is on the Queen of Hearts, so that the top of each card is visible. The class has instance variables to represent its location, and its orientation (whether the contained objects are spread out to the right, left, up, or down). Methods include display, move, append new elements, append a second *SpreadList*, retrieve an element, retrieve a sub *SpreadList*, and determine if an element is a member of a *SpreadList*. This container class also uses templates in the implementation so that any object that can be displayed and moved can be stored.

Two complete implementations of solitaire games are provided with the module: *Up and Down the River* and *Klondike*. These two games exercise all the capabilities of the classes described above. Instructions for these games are included with the module. Although these two implementations use only augmented stacks and vectors, solitaire games exist that map nicely to trees or graphs.

4 Data Compression

In Duke's CS 2 course CPS 100, the culminating project for the past two and half years has been a data compression program using Huffman coding [11]. Although some CS 2 texts cover Huffman coding (e.g., see [19]) we have not found supporting materials for reading and writing bits-at-a-time which is required for data compression. We provide students with two stream classes: *ibstream* and *obstream*, for

[3]Given the addictive nature of games, some instructors might see this as a disadvantage rather than an asset.

[4]The standard class-based implementation of arrays in C++ is called a vector.

input and output bit streams, respectively, to isolate I/O of single and multiple bits. In some semesters we have students write the complete program from scratch except for the I/O classes. In 1994 the program was implemented in C using ADTs as a means of encapsulation. In the past four semesters it has been implemented using an object-oriented approach. Students are required to write re-usable C++ classes to implement the various pieces of the program. In addition, classes used throughout the semester are re-used in implementing the compression program. These classes include: a templated priority queue class, a binary tree class, a list class, and a histogram class. As one use of the module approach, we have given students a naïve implementation of a priority queue (an unsorted array) and asked them to re-implement the class using heaps. Using this kind of modular/class approach allows students to focus debugging attention on specific parts of the program. Class diagrams from one student solution written from scratch are shown in Figure 2.

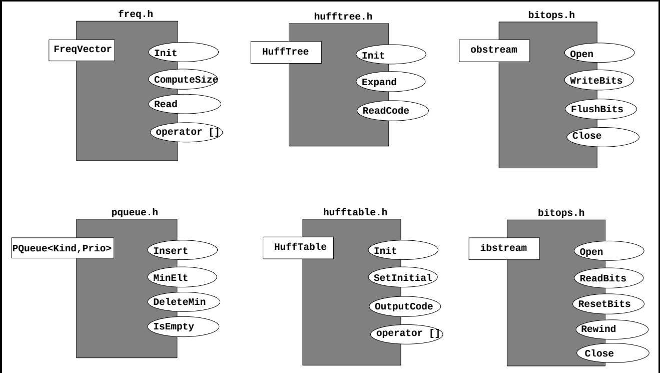

Figure 2: Class Diagrams for Huffman Coding

Student enthusiasm for this project is extremely high. The resulting program can compress any file: both text and binary (executable) programs. Students know about data compression and are amazed that they have written a complete program that compresses reasonably well. End of semester evaluations consistently praise this assignment and students do well on test material associated with the assignment, e.g., construction and traversal of binary trees.

5 Additional Modules

The modules we have described have been tested in previous semesters and are now available with supporting materials in module form. We have also developed several new modules that we will test in the Fall of 1996. Materials for these modules are also available on the web including all code, but with less written supporting material than the other modules.

5.1 Navigator

Several car models can be purchased with on-board navigation systems that rely on the Global Positioning System (GPS) for location and mapping. We have written an assignment, based on a class assignment developed at Carnegie Mellon, that can be used to navigate pedestrian and vehicle travel. Three data files are used for each navigable region processed by the program. In addition to simple location queries, e.g., "Where is the restaurant Neo-China?", travel queries can also be answered by providing directional clues based on locational information maintained by the program. below (output from the program)

```
Go for 370 feet, turn right onto Oak Drive
Go for 250 feet, and stop at number 299
Total distance traveled:  620 feet
```

Classes used in this module include queues, priority queues, hash tables, and graphs. Classes and algorithm support are provided for breadth-first search and for Dijkstra's shortest path algorithm. Data files available from Carnegie Mellon include thousands of locations in Pittsburgh. A plot of the locations, generated by an animator we have developed for this project[5] is reproduced in Figure 3.

Figure 3: City Map of Pittsburgh (realized in Lambada)

5.2 Interactive Graphics

We have developed several detailed modules that support animations and interactive graphics for use in CS 2. We provide a brief synopsis of these, detailed descriptions may be found on our web pages.

One module is for discrete-event simulations from queuing theory. We include support for traditional single/multiple server queues that show customer arrivals and departures graphically. We also include a detailed simulation of multiple elevators that provides a fertile framework for experimenting with control strategies and for using inheritance in many classes.

[5]Our animator is called *Lambada* and is written in Java. It supports a superset of the commands available in the Xtango and Samba animators [17].

We have a module for the game of TETRIS, that we call MINT (Mint Is Not Tetris). This module uses one- and two-dimensional arrays, and inheritance.

We also have a module for simulation that has the flavor of artificial-life games. This simulation, called *Darwin*, is modeled after an assignment given at Stanford University. Creatures roam a rectangular world behaving according to a species-specific program written in a BASIC-like language. We used this module successfully in the summer of 1996 with a group of summer students participating in a workshop for women and minorities. We anticipate that this module will be quite successful with our classes because of the graphics, the biology connection, and the meta-programming nature of the assignment wherein students in essence write a small interpreter.

6 Support and Summary

The modules we have developed will be refined next summer by a cadre of undergraduates. These students were of immeasurable help developing initial versions of the modules. All the modules support apprentice-style learning using applications. We have tried to provide materials so that educators can make effective use of the modules over several semesters. We hope that as the modules are used, feedback from students and instructors will help improve the modules and add more material to them. Information on how to help with the project is accessible from our web pages.

References

[1] ADVANCED PLACEMENT COMPUTER SCIENCE DEVELOPMENT COMMITTEE. Classes for the ap computer science course. http://www.cs.duke.edu/∼ola/ap.html, 1996.

[2] AIKEN, A. Cool: A portable project for teaching compiler construction. *SIGPLAN Notices 31*, 7 (July 1996), 19–24.

[3] ASTRACHAN, O., AND BONO, C. Using simulation in an objects-early approach to cs1 and cs2. In *OOPSLA: Object Oriented Programming Systems, Languages, and Applications: Educator's Symposium* (October 1994), pp. 1–8. Portland, Oregon.

[4] ASTRACHAN, O., AND REED, D. AAA and CS-1: The applied apprenticeship approach to CS 1. In *The Papers of the Twenty-Sixth SIGCSE Technical Symposium on Computer Science Education* (March 1995), ACM Press, pp. 1–5. SIGCSE Bulletin V. 27 N 1.

[5] CHRISTOPHER, W., PROCTER, S., AND ANDERSON, T. The Nachos instructional operating system. *1993 Winter USENIX Conference* (January 1993), 479–488.

[6] CLANCY, M., AND LINN, M. Bowling scores. Used in College Board training classes, 1989.

[7] CLANCY, M. J., AND LINN, M. C. The case for case studies of programming problems. *Communications of the CACM 35*, 3 (1992), 121–132.

[8] CLANCY, M. J., AND LINN, M. C. *Designing Pascal Solutions: Case Studies with Data Structures*. W.H. Freeman and Company, 1996.

[9] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[10] HILBURN, T. A top-down approach to teaching an introductory computer science course. In *The Papers of the Twenty-Fourth SIGCSE Technical Symposium on Computer Science Education* (March 1993), ACM Press, pp. 58–62. SIGCSE Bulletin V. 25 N 1.

[11] HUFFMAN, D. A method for the contruction of minimum redundancy codes. In *Proceedings IRE 40* (1951), pp. 1098–1101.

[12] METER, G., AND MILLER, P. Engaging students and teaching modern concepts: Literate, situated, object-oriented programming. In *The Papers of the Twenty-Fifth SIGCSE Technical Symposium on Computer Science Education* (March 1994), ACM Press, pp. 329–333. SIGCSE Bulletin V. 26 N 1.

[13] MEYER, B. Toward an object-oriented curriculum. *Journal of Object Oriented Programming* (May 1993), 76–81.

[14] MOREHEAD, A. H., AND MOTT-SMITH, G. *The Complete Book of Solitaire and Patience Games*. Bantam Books, 1977.

[15] REEK, M. M. A top-down approach to teaching programming. In *The Papers of the Twenty-Sixth SIGCSE Technical Symposium on Computer Science Education* (March 1995), ACM Press, pp. 6–9. SIGCSE Bulletin V. 27 N 1.

[16] SMITH, R. Video games challenge and motivate data structure students. In *Proceedings of 30th Annual Southeast Conference of the ACM* (1992), pp. 11–14.

[17] STASKO, J., BADRE, A., AND LEWIS, C. Do algorithm animations assist learning? An empirical study and analysis. In *INTERCHI 93 Conference Proceedings: Human Factors in Computing Systems* (April 1993), ACM Press, pp. 61–66.

[18] WALLINGFORD, E. Toward a first course based on object-oriented patterns. In *The Papers of the Twenty-Seventh SIGCSE Technical Symposium on Computer Science Education* (1996), ACM Press, pp. 27–31.

[19] WEISS, M. A. *Algorithms, Data Structures, and Problem Solving With C++*. Addison Wesley, 1996.