SOFTWARE REUSE: A CONTEXT FOR INTRODUCING SOFTWARE ENGINEERING PRINCIPLES IN A TRADITIONAL COMPUTER SCIENCE SECOND COURSE[1]

Murali Sitaraman
murali@cs.wvu.edu

Jeff Gray[2]
jgg@cs.wvu.edu
jgray@vuse.vanderbilt.edu

Department of Statistics and Computer Science
West Virginia University
Morgantown, WV 26506-6330
(304)-293-3607

ABSTRACT

The demand for graduates well-trained in software engineering principles and practices is continuing to increase. Educators of undergraduate computer science curricula can no longer afford to wait until the senior year to motivate and instill software engineering principles in their students. However, introducing new principles in early courses, without displacing existing principles, is not easy. This paper presents software reuse as an appropriate context for discussing principles of software design and specification, along with abstraction and encapsulation, in traditional freshmen computer science courses. The paper includes an outline of lectures and an example sequence of lab assignments for the second course following the reuse-based approach. It describes our experience in adapting this approach in Ada for four semesters at the West Virginia University.

1 INTRODUCTION

Most current computer science curricula introduce software engineering principles at a junior/senior-level course, motivating the need for design and specification using group projects. This late exposure leaves students with little time to master and practice these principles before they graduate and join the industry. Moreover, having done most of their programming projects in previous courses without following sound principles, junior/senior-level students are less interested in correcting the bad habits they have practiced in this process. The need for learning key software engineering principles early in the curriculum is, therefore, quite clear. This need has also been stressed in recent literature [Reuse-Ed 92, SEI 92].

Introducing new principles, such as formal specification and design, in freshmen computer science courses is not easy. For one thing, new principles must be included without excluding important concepts conventionally taught in these courses. Additionally, it is important to motivate the need for learning these principles using an approach that is applicable to freshmen students. An answer to these problems lies in software reuse.

Concentrating on reuse brings into focus a number of key software engineering principles. Software engineering textbooks (e.g., [Pressman 90]) explain the role of abstraction in specifications and stress the need for designing the specification of a software product before it is implemented. Such books also argue for the need to modularize the construction of large software systems and for the need to restrict communication among different modules through only their interfaces. In addition, these textbooks often emphasize the importance of software quality and discuss the advantages of following these

[1] This research is funded in part by DARPA Grant DAAL03-92-G-0412; it has also benefited from research under NASA Grant 7629/229/0824 and NSF Grant CCR-99204461.

[2] Jeff Gray's present address is: Department of Computer Science, Box 1679B, Vanderbilt University, Nashville, TN 37235.

principles in software maintenance. All of these issues are equally important in the context of reusable software.

Focusing on software reuse makes specification and design central issues in problem solving and not as issues that are taught on the side. In the reuse-based approach, students appreciate the need for abstraction, specification, design, and quality by reusing components based entirely on their specifications, which are supplied by the lab instructor. Students also see software construction more as a process of assembling existing reusable software components rather than starting from scratch.

Recent literature on software reuse contains a variety of different definitions or classifications of the term [Krueger 92, Yourdon 92]. The definition of reuse used in this paper is one which is component-based [Murali 90b, Weide 91]. We view a reusable component as having two distinct elements: a (formal) specification and a (certifiable) implementation of that specification. In the reuse-based course, all reuse is based only on the specification and performance characteristics of the implementation. The focus is on components which are designed for reuse since this is where the benefits are maximized [Hollings 92]. Issues in reuse based on code scavenging, or other methods where the utilization of already existing software occurs by accident or serendipity, are not discussed.

The rest of the paper is organized into the following sections. Section 2 discusses the salient benefits of the reuse-based approach. Section 3 discusses the technical foundations of the approach. Section 4 then presents a course outline that has been used to teach the second course during the past four semesters at the West Virginia University. Included in this section is a sample sequence of laboratory assignments. Finally, section 5 reports our experience and conclusions.

2 THE BENEFITS OF A REUSE-BASED APPROACH

In this section, we outline several benefits of teaching the second course in computer science following a reuse-based approach.

1. Reuse provides an excellent context for presenting important computer science and software engineering principles.

The idea that a software component will be reused elsewhere permits students to readily see the importance of key software engineering principles. The realization that the developer of a component and the prospective client are likely to be different people leads students to new thinking. In particular, the importance and relevance of the following principles are made clear early in the curriculum:

- Separation of the specification and implementation details of a component

  This separation permits reuse to be based on the specification of a component; without it, reuse is impractical even when possible.

- Unambiguous and abstract expression of a specification

  The specification of a reusable component permits clients, as well as implementation developers of the component, to clearly understand how the component is supposed to behave. Such understanding in turn makes it easy to reason about other software that uses this component.

- Design

  If a reusable component is not well-designed, the scope for its reuse will be limited. Students learn to appreciate the role of design issues in software reuse and general software development. The design issues that students are exposed to include: adequate functionality, cohesion, coupling, composability, generality, and minimality.

- Certification

  If there is not sufficient confidence that the implementation of a component meets its specification correctly, it is likely that it will not be reused. Issues of verification and testing, seen as crucial for software development but seldom given much attention in undergraduate curricula, become prominent issues in the minds of freshmen students.

- Efficient implementations

  If a reusable component is not implemented efficiently, then users will prefer custom-built components. This fact in turn motivates students to appreciate the importance of efficiency, but not at the expense of correctness. The issue of designing efficient implementations also provides an excellent context for introducing topics in the analysis of algorithms.

- Maintenance

  Construction of software systems using existing reusable software components greatly enhances the

maintainability of these systems. The reuse-based approach provides students with an insight into maintenance issues early in the curriculum.

The fundamental software engineering principles mentioned above, when taught without "thinking" reuse, seem neither important nor interesting to students in introductory computer science courses. Traditionally, introductory courses often concentrate on the syntactic details of a particular programming language rather than specific principles. Absence of an early exposure to software engineering principles prevents students from applying and therefore understanding these ideas in a vast majority of their undergraduate courses. Since significant attention to syntactic details remains essential for beginners, the reuse-based second course continues to introduce language constructs to students. However, the approach also attempts to infuse various software engineering principles into the consciousness of freshmen students. This is done at an early stage of the curriculum so that students are afforded the opportunity to practice the principles throughout their undergraduate careers. Thus, by the time students are ready to enter the industry, they have developed a significant amount of confidence in designing and specifying software using sound principles.

2. Focus on reuse permits introduction to principles of specification and design early in the curriculum.

Principles of specification and design are usually introduced at a late point in most current curricula. When these principles are presented, they are taught as "other" ideas rather than central themes for software construction. In a junior/senior-level software engineering class, group projects are typically created to motivate the need for specification and design. One factor that complicates such late introduction to these principles is that students have already acquired certain "bad" software engineering practices (e.g., coding an implementation before designing the specification) that are difficult to change.

At the freshmen level, thinking reuse provides immediate motivation for applying software engineering principles while avoiding the need to form student teams. In the reuse-based courses, the laboratory instructor and students form a team. In some projects, students solve a problem using a reusable component which is implemented by the instructor. In others, they implement a component on their own. Acting as both developers and clients, students appreciate the role of design and specification in software construction at an early stage.

3. Computer scientists will now be equipped with a component-based software development mindset.

It is widely believed that sometime in the future new software products will be constructed largely by assembling existing components [Bigger 89]. Such construction has the potential to solve the most important problem facing software engineers today: how to produce high quality software on time. Mature engineering industries, such as the electronics industry, use a component assembly-based approach, and it is likely that a mature software industry will follow suit. Thinking reuse early in the curriculum prepares students for this futuristic view of software construction. In fact, students are expected to reuse some of the components developed in their earlier courses to complete software projects they encounter later in the curriculum.

4. Non-computer science majors can acquire important insight into principles of software reuse and software engineering.

The reuse-centered approach greatly benefits non-computer science majors. It introduces them to important software engineering principles they would not otherwise have had the opportunity to learn and apply. These principles, which to the non-computer science student may not have any intrinsic value beyond the course, can carry over a particular mindset that could be applied to other disciplines as well. For example, English majors who happen to complete the reuse-based course will not only see the need for specifying and designing software before it is implemented, but also realize that the same philosophy applies in the construction of large term papers or literary works. In this sense, they acquire the realization that advanced preparation before the implementation of any type of work not only increases the quality of the work but also improves productivity, an impetus for applying the engineering metaphor to these other disciplines. This is certainly not a novel idea; the famous novelist/scientist C.P. Snow has often discussed the need for the two cultures, represented by the humanities and the sciences, to have a mutual understanding of the basic principles found in other disciplines [Snow 58].

3 TECHNICAL UNDERPINNINGS

The specification-based approach to software reuse, syntactically facilitated by Ada, offers tremendous advantages in the development and maintenance of large software systems. The reuse-based course strictly adheres to this philosophy, and all reusable parts developed in the course are reused based only on their specifications. Throughout the lectures, homework, and lab assignments, students always begin by reusing some component that is assumed to have already been implemented by the lab

instructor. They begin as users of components to solve some interesting applications, e.g., finding a solution to a "maze" problem using a stack. Later in the semester, they act as implementers of components. Some of these implementations are layered on top of existing components while others are built from scratch. This section discusses the technical foundations of the approach.

3.1 SPECIFICATION

The importance of specifications in the context of reusable software parts cannot be over-emphasized [Luckham 87, Meyer 85, Weide 91, Wing 90]. The specification of a reusable part serves as a contract between developers and clients of that part. Without the specification, the implementation developers will not know what should be implemented and clients will not know what is being used. Rigorous certification efforts also need a certain degree of formality in specifications. The specification must, therefore, be formal, yet understandable to a potential client and implementer; here, a beginning undergraduate student. The importance of an appropriate expression of a specification for a given audience is described in [Sitaraman 93a].

We have used the term "specification" to refer to both the syntactic and semantic interfaces of a reusable part. Figure 1 shows the specification of an Ada generic component, annotated using a variant of the RESOLVE specification language, which provides an abstract data type Set and operations on variables of this type. In the RESOLVE approach to specifications [Hollings 92, Sitaraman 93a, Weide 91], every abstract data type actually has an "abstract view" that is already familiar to the students. RESOLVE is unique in permitting the development of formal, yet understandable, specifications at the freshmen level. A comparison of RESOLVE and other specification approaches, such as those in [Wing 90], can be found in [Sitaraman 93a].

```
generic

  type Item is limited private;

  with procedure Item_Initialize(X : in out Item);
  --! ensures X = initial_Item

  with procedure Item_Finalize(X : in out Item);

  with procedure Item_Swap(X, Y : in out Item);
  --! ensures X = #Y and Y = #X

  with procedure Item_Compare(X, Y : in out Item;
                              Answer : out Boolean);
  --! ensures X = #X and Y = #Y and
  --!         Answer iff (X = Y)

package Set_Template is

  type Set is limited private;
  --! abstract view: Set is modeled by a
  --! mathematical Set of Items

-- standard operations

  procedure Initialize(S : in out Set);
  --! ensures S = {}

  procedure Finalize(S : in out Set);

  procedure Swap(S1, S2 : in out Set);
  --! ensures S1 = #S2 and S2 = #S1

-- set operations

  procedure Add_Item(S : in out Set;
                     X : in out Item);
  --! requires X ∉ S
  --! ensures S = #S ∪ {#X} and X = initial_Item

  procedure Remove_Item(S : in out Set;
                        X : in out Item);
  --! requires X ∈ S
  --! ensures S = #S - {#X}

  procedure Remove_Any_One_Item(S : in out Set;
                                X : in out Item);
  --! requires S ≠ {}
  --! ensures X ∈ #S and S = #S - {X}

  procedure Is_Member(S : in out Set;
                      X : in out Item;
                      Member : out Boolean);
  --! ensures S = #S and X = #X and
  --!         Member = (X ∈ S)

  procedure Is_Empty(S : in out Set;
                     Empty : out Boolean);
  --! ensures S = #S and Empty = (S = {})

private

      type Set_Rep;
      type Set is access Set_Rep;

end Set_Template;
```

The Specification of a Set Package
Figure 1

It is important to note that before students see the final version of Set_Template shown in Figure 1, they have viewed several intermediate versions which incrementally introduce them to various conventions used in RESOLVE. Other data abstractions, such as Stacks and Lists, are designed similarly. Therefore, only for the first example do the students have to learn the key principles (e.g., abstraction and information hiding) in specification.

Using RESOLVE, the program type Set is explained using mathematical notations from set theory, such as $\in$ and $\cup$. This specification of a set does not involve the idea of pointers or arrays, thus, freeing users from the necessity of understanding details of the private part. From an examination of Figure 2, one can note that two clauses are used in the specification of each operation; namely, a requires clause (pre-condition) and an ensures clause (post-condition). The requires clause states what must be true of the arguments passed to the operation. If the requires clause is true when an operation is called, the ensures clause will be true when it terminates, assuming a correct implementation of the operation. In the ensures clause, the notation "#X" for a parameter X denotes the incoming value of the parameter when the operation is called and "X" denotes its value when the operation returns. In the requires clause, the variables always denote the incoming parameter values. If either clause is omitted from the specification of an operation, the default is a true implication for that clause.

The specification of most operations provided in Set_Template should be easy to understand for those who have a basic comprehension of set theory. For example, the specifications capture the behavior that Remove_Item removes a specified item whereas Remove_Any_One_Item removes a random item from the set. This understanding is a result of applying two complementing principles, i.e., information hiding and abstraction. To demonstrate how information hiding is applied in this component, note that the program type Set is not only protected from the user, by making it a *limited private* type, but the details of the type representation are hidden in the package body. Using this approach, the user is prohibited from accessing and viewing the details of the type. The principle of abstraction is applied to Set_Template by modeling the program type Set using a related concept, i.e., mathematical sets. Rather than describing this type in terms of an array, or some other low level construct, the utilization of mathematical sets improves understanding and captures the true behavior of the component. These two principles aid in understanding since the client of the component does not need to sift through nonessential details in order to comprehend the true meaning. This example, presented to students early in the course, immediately introduces them to these important principles in a context which is often missing in other approaches for teaching the second course.

Specifications of several other interesting data abstractions, at a level of formality suitable for an undergraduate class, can be found in [Sitaraman 93b]. These specifications use familiar mathematical concepts such as strings, natural numbers, functions and relations. It is emphasized again that the goal is to make specifications reasonably formal, but allow them to be at a level appropriate for undergraduate students.

3.2 DESIGN

Recently, design issues have gained prominence in the literature on software engineering and reuse. For widespread reuse to occur, software parts must be designed to be reused. Set_Template is an example of a software part which was carefully designed for reuse following specific guidelines in [Hollings 92] and principles in [Harms 91, Reuse 92, SPC 89, Edwards 90].

As noted earlier, the specification of Set is abstract. It has also been designed without any implementation bias. Notice that the representation of the encapsulated type has been deferred to the body of the package. This is to permit the development of multiple implementations for this package by using different data representations and algorithms.

Set_Template provides only a *primary* set of operations. Other generic *secondary* operations (e.g., a Set union operation) can be constructed using layering. Standard operations such as Initialize, Finalize, and Swap are included in the specification of every data abstraction. This design approach permits the possibility of efficient data movement and storage management. Several subtle design ideas can be seen from a careful study of the Set_Template specification. Note that both the program type Set and the generic parameter type Item have been declared to be *limited private*. This permits construction of composable types such as a Set of a Set of Integers. Further explanation of these and other important design guidelines can be found in [Hollings 92, Gray 93b].

3.3 IMPLEMENTATION

Efficiency is an important characteristic of any software part. Unless reusable parts are as efficient as custom-made software parts, substantial reuse will not result. There is a common misconception that reusable parts are necessarily inefficient. On the contrary, it has been argued in [Harms 91], for example, that this need not be the case. Techniques for building efficient implementations of reusable parts, and analysis of their efficiency characteristics, are part of the reuse-based course. In this process, important programming techniques (e.g., recursion and backtracking) and methods for analysis of algorithms are discussed. The course also emphasizes the need for developing multiple reusable parts to satisfy the same specification.

4 AN OUTLINE OF MATERIALS FOR THE SECOND COURSE

This section presents a course outline which has been used to teach the reuse-based course. The section also contains a sub-sequence of sample laboratory assignments used to teach the approach.

4.1 LECTURE OUTLINE

In [Sitaraman 93b], a complete set of materials for the second course, including home work assignments and lecture notes, is presented. Shown below is an outline of the content of the materials.

Week 1 *1. The Engineering Metaphor

Week 1 *2. Syntactic and Semantic Specification

Week 2 *3. An Introduction to Formal Specification

Week 3-4 4. Protected, Abstract, and Generic Data Types

- Key example: Sets

Week 4-6 5. Stacks: Design, Specification, and Implementation

- Specification of unbounded stacks
- Introduction to design issues in specification
  - -- Defensive/non-defensive specifications
  - -- Primary and secondary stack operations
  - -- Guidelines
- Problem solving using stacks
- Specification of bounded stacks
- An implementation of bounded stacks

Week 7 6. An introduction to certification of implementation correctness

- Key example: Certification of bounded stacks

Week 7-8 7. An introduction to efficiency analysis of implementations

- Key example: An analysis of a bounded stack

Week 8 8. Queues

- Specification of unbounded queues
- Specification of bounded queues
- Secondary queue operations
- Problem solving using queues
- An implementation of bounded queues

Week 9 9. Lists

- Specification of lists
- Secondary list operations
- Problem solving using lists
- List-based layered implementations (e.g., unbounded stacks and queues)

Week 10-11 10. Pointers as an implementation mechanism for unbounded ADT's

- An introduction to access types and variables
- An implementation of lists using pointers
- Direct implementation of unbounded stacks/queues

Week 11-12 *11. Recursion as a problem solving device

- Examples
- Implementation of secondary operations (e.g., Stacks, Queues, and Lists)

Week 12-13 12. Trees

- Specification
- Tree traversals
- An implementation of trees using pointers

Week 14 13. Searching

- Specification
- Multiple implementation techniques
  - -- Linear and ordered linear search
  - -- Binary search trees
  - -- Searching using hash tables

Week 15 14. Sorting

- Specification
- Multiple implementation techniques
  - -- Bubble Sort, Merge Sort, and Quick Sort

Week 15 15. Course summary

**Note:* Parts or all of chapters 1 (context setting), 2 and 3 (specification of procedures), and 11 (recursion) can be profitably covered in a first course; at West Virginia University, recursion is covered in the second course.

4.2 AN EXAMPLE LABORATORY SEQUENCE

Laboratory sections play an important role in the reuse-motivated second course. It is the forum where students actually gain practice in applying the software engineering principles that they have been taught in the lecture section

of the class. In all of the laboratory sections that have been offered over the past four semesters, there seems to be recurring themes in the descriptions of the assignments. Early in the semester, the students are simply users of a component. Next, they become implementers of a component using a layered approach. Finally, students implement their own components using "from-scratch" methods utilizing access types or arrays.

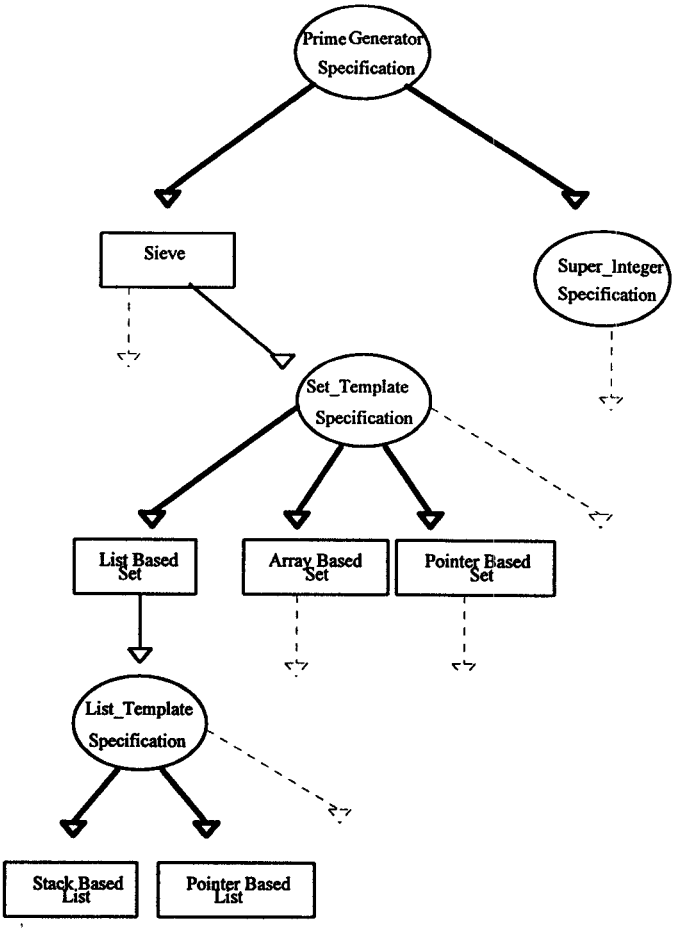
Examples of labs that have been used in the past include:

- Backtracking problems where the students are given a stack package and asked to solve some application (e.g., The Eight Queens problem, or helping a mouse find cheese in a maze);

- Manipulation of a Super Integer package which allows representation of integers larger than that provided by the standard integer type; and

- Incorporation of a Set package to solve graph problems.

This section provides a specific sequence of labs used in the Spring 1993 offering of the course. Each description of the lab follows an outline which consists of a statement of the problem, the items supplied by the lab instructor, the week in which the lab is assigned, and the principles taught. Figure 2 shows a pictorial view of the structure of this lab sequence following several of the conventions of the 3C Model used in [Tracz 89]. For a more detailed description of other labs, along with their solutions, see [Gray 93a, Gray 93b].



A Pictorial Representation of the Lab Sequence
Figure 2

Lab Assignment 1 — Introduction to Super Integers

Problem: Given the specification to a Super Integer component, construct secondary operations for this component (e.g., Print_Super_Int) and create a client program which performs various manipulations on variables of type Super_Int.

Items supplied by the Lab instructor: Listing of the Super Integer specification. Also, information about how to access the Super Integer object code for linking purposes. Student does not see any implementation of Super Integers.

Course Outline: Week 2-4

Principles Taught: Separation of the specification from its implementations, Specification-based reuse, Understanding of abstract specifications, Construction of secondary operations.

Lab Assignment 2 — Super Integer Prime Generator

Problem 1: Given the specification for a Set component, create a prime number generator for Super Integers using the Sieve of Eratosthenes algorithm.

Problem 2: Construct several secondary operations (e.g., Set union and intersection) for Sets.

Lab instructor supplied items: Listing of the Set specification. Also, information about how to access the Set object code for linking purposes. Student does not see any implementation of the Set component.

Course Outline: Week 4-6

Principles Taught: Problem solving using Sets, Defensive and non-defensive programming.

Lab Assignment 3 — Layered Set Implementation

Problem: Given the specification to a List component, implement the Set component from assignment two using a layered approach based on List. Re-link the Set created in this assignment with the client program from the previous assignment.

Lab instructor supplied items: Access to the List specification. Also, information about how to access the List object code for linking purposes. Student does not see any implementation of the List component.

Course Outline: Week 7-9

Principles Taught: The layered implementation approach toward component construction, Introduction to the List abstract data type, Multiple implementations for the same specification.

Lab Assignment 4 — "From-Scratch" Implementations of Lists

Problem: Given the specification to a List component, create a "from-scratch" implementation of this component using access types. Also, create secondary operations, such as Print_List, which are written recursively. Re-link this new implementation with the previous assignment to provide the prime generator.

Lab instructor supplied items: None

Course Outline: Week 10-12

Principles Taught: Recursion, Use of Ada access types, "From-scratch" implementations.

4.3 DISTINGUISHING FEATURES

This section identifies several distinguishing characteristics of the reuse-based approach which differ from traditional methods for teaching the second course. A more detailed comparison, which is beyond the scope of this paper, can be found in [Gray 93b].

The most obvious feature which distinguishes the approach from other methods is the introduction of formal, yet understandable, specifications in a freshmen level course. Such formal methods are fundamental for teaching software engineering principles. In the course, students reuse a package immediately after seeing its specification, but long before actual implementations are discussed. This emphasis is highlighted by the example sequence of lab assignments presented in the previous section. Most textbooks used for second courses do not concentrate on specifications, e.g., [Smith 87, Horowitz 78]. The few textbooks that do use specifications are often limited to syntactic specifications, e.g., [Booch 86]. The specifications that the students see in the reuse-based course are both syntactic and semantic. These specifications are also written in a manner that allows for multiple implementations, an idea which many authors seem to disregard.

The reuse-based course also emphasizes the need for designing an "appropriate" set of operations on abstract data types; the standardization and the small number of primary operations make abstract data types easy to understand. Efficiency issues also play an important role in the course; every implementation discussed in the course after week six is analyzed carefully. In the reuse-based approach, students are exposed to design issues that are not normally discussed in traditional freshmen courses.

In other details related to the outline presented in section 4.1, note that pointers are not taught until much later in the course. Lists are taught before pointers are even introduced, made possible only through abstract modeling. The specification of Lists, of course, do not involve pointers. Introducing access types after the layered implementation approach allows students to more fully appreciate the advantages of layered implementations and reuse in terms of ease of construction. Students quickly discover that the assembly of reusable components is considerably easier than the "from-scratch" method.

Finally, it is essential to note that most traditional principles taught in a second course are still taught in this course. This is critical because we do not want to displace principles taught in a conventional second course (e.g., efficiency analysis, pointers, and recursion) which are important for problem solving. Principles such as recursion, however, can be moved to the first course. The reuse-based approach, therefore, provides a feasible context for introducing software engineering principles in most schools.

5 EXPERIENCE AND CONCLUSIONS

The approach described in this paper has been used to teach the second course in computer science at the West Virginia University. The reuse-based approach instills software engineering principles without displacing traditional concepts taught in a second course, and thus it provides a practical approach for adaptation in most schools. It was developed to attack a common problem found in most

curricula, i.e., the introduction of fundamental principles of computer science void of any particular context. Early exposure to the principles presented in this paper will aid students in applying the ideas toward a vast majority of the programming projects that they will encounter throughout the remainder of their undergraduate careers. A few schools have already shown interest in adapting our approach, including Indiana University South East, Muskingum College and The Ohio State University.

One of the most significant findings of our research is the ability of freshmen to understand specifications when presented using our specification approach, demonstrating that specifications can be both reasonably formal and understandable. Feedback from students, obtained through confidential evaluations, has been very positive.

We are currently keeping track of previous students who have taken the course using the approach. We hope to report at a later date on how the approach, which students encounter early in the curriculum, affects other courses in which they are involved. We are also investigating principles that can be profitably taught in the first course. It is possible to "push" some of these principles down into the first course, such as from chapters on procedure specifications and recursion.

Finally, though our overall approach is language-independent, Ada has proved to be a most suitable language for teaching these principles to freshmen students. We are also beginning to explore how the approach could take advantage of the concepts in Ada 9X and languages such as C++.

6 ACKNOWLEDGMENTS

We are pleased to acknowledge the contributions of all members of the Reusable Software Research Group at The Ohio State University and members of the group at the West Virginia University towards this work. Our discussions with Tim Long and Bruce Weide at Ohio State, Doug Harms at Muskingum College, Jim Mooney and Frances Van Scoy at West Virginia University, and Jack Kramer at ARPA/SISTO have especially benefited this effort.

7 BIBLIOGRAPHY

[Bigger 89] Biggerstaff, T. and A. J. Perlis, *Software Reusability, Volume 1: Concepts and Models,. Volume 2: Applications and Experience*, Addison-Wesley, 1989.

[Booch 87] Booch, G., *Software Components with Ada*, Benjamin/Cummings, 1987.

[Edwards 90] Edwards, S., "An approach for constructing reusable software components in Ada," IDA Paper P-2378, Institute for Defense Analyses, Alexandria, VA, September 1990.

[Gray 93a] Gray, J. G., "Teaching the Second Computer Science Course in a Reuse-Based Setting: A Sequence of Laboratory Assignments in Ada," In *Proceedings of the Eleventh National Conference on Ada Technology*, March 1993, pp. 38-45.

[Gray 93b] Gray, J. G., "The Role of Reuse in Introducing Software Engineering Principles in a Computer Science Second Course," MS Project Report, West Virginia University, Morgantown, WV, May 1993, pp. 1-66.

[Harms 89] Harms, D.E., and B. W. Weide, "Efficient Initialization and Finalization of Data Structures: Why and How," Technical Report, Department of Computer and Information Science, The Ohio State University, OSU-CISRC-3/89-TR11, March 1989.

[Harms 91] Harms, D. E., and B. W. Weide, "Copying and swapping: Influences on the design of reusable software components," *IEEE Trans. Soft. Eng.*, 17(5): 424-435, 1991.

[Hollings 92] Hollingsworth, J., *Software Component Design-for-Reuse: A Language-Independent Discipline Applied to Ada*, Ph.D. thesis, The Ohio State University, 1992. Available by anonymous FTP from archive.cis.ohio-state.edu in directory pub/tech-report/TR1-1993.

[Horowitz 78] Horowitz, E., *Fundamentals of Data Structures*, Computer Science Press, 1978.

[Krueger 92] Krueger, C. W., "Software Reuse", *ACM Computing Surveys*, Vol. 24, No. 2, June 1992, pp. 131-184.

[Luckham 87] Luckham, D., and F. W. von Henke, B. Krieg-Bruckner, and O. Owe, *ANNA: A Language for Annotating Ada Programs*, Springer-Verlag, 1987.

[Meyer 85] Meyer, B., "On Formalism in Specifications," *IEEE Software 2*, no. 1, pp. 6-26.

[Murali 90a] Muralidharan, S., and Weide, B. W., "Should Data Abstraction Be Violated to Enhance Software Reuse?" *Proceedings of the Eighth National Conference on Ada Technology*, Atlanta, GA, March 1990, pp. 515-524.

[Murali 90b] Muralidharan, S. and Weide, B. W., "Reusable Software Components = Formal Specifications + Object Code: Some Implications," *Third Annual Workshop: Methods and Tools for Reuse*, Syracuse, NY, June 1990.

[Pressman 90] Pressman, R. S., *Software Engineering: A Practitioners Approach*, McGraw-Hill, Inc., 1990.

[Reuse 92] *Proceedings of WISR '92 Fifth Annual Workshop on Software Reuse*, San Francisco, CA, 1992.

[Reuse-Ed 92] *Proceedings of the Reuse Education Workshop*, Morgantown, WV, 1992.

[Sitaraman 92] Sitaraman, M., "A Class of Programming Language Mechanisms to Facilitate Multiple Implementations of a Specification," *Proceedings of the 1992 IEEE International Conference on Computer Languages*, San Francisco, CA, April 1992.

[Sitaraman 93a] Sitaraman, M., L. Welch, and D. Harms, "On Specification of Reusable Software Components," *International Journal of Software Engineering and Knowledge Engineering 3*, 2, June 1993.

[Sitaraman 93b] Sitaraman, M., *Course Notes for the Second Course in Computer Science*, Department of Computer Science, West Virginia University, 1993.

[Smith 87] Smith, H., *Data Structures: Form and Function*, HBJ Publishers, 1987.

[Snow 58] Snow, C.P., *The Search*, Charles Scribner and Sons, 1958.

[SPC 89] Software Productivity Consortium, *Ada Quality and Style: Guidelines for Professional Programmers*, van Nostrand Reinhold, 1989.

[Tracz 89] Tracz, W. J., and S. Edwards, "Implementation Working Group Report," *Reuse in Practice Workshop*, Pittsburgh, PA, 1989.

[Weide 91] Weide, B. W., W. F. Ogden, and S. H. Zweben, "Reusable Software Components," In *Advances in Computers*, volume 33, Ed. M. C. Yovits, pp. 1-65, Academic Press, 1991.

[Wing 90] Wing, J. M., "A Specifiers introduction to formal methods," *IEEE Computer*, 23(9): pp. 8-24, 1990.

[Yourdon 92] Yourdon, E., *Decline and Fall of the American Programmer*, Prentice Hall, 1992.