The Role of Reuse in Introducing Software Engineering Principles in a Computer Science Second Course

PROJECT REPORT

Presented in Partial Fulfillment of the Requirements

for the Degree Master of Science in the

Department of Statistics and Computer Science

By

Jeffrey Gene Gray, B.S.

* * * * * * *

West Virginia University

May 1993

Project Committee:

Dr. Murali Sitaraman

Dr. Frances VanScoy

Dr. Douglas Harms

Approved by:

_____

Project Adviser

Department of
Statistics and Computer Science

ACKNOWLEDGMENTS

In the effort to complete this final degree requirement, I find myself indebted to my adviser, Murali Sitaraman. Over the past year, his willingness to assist me in the preparation of various papers, as well as this report, has been invaluable and very much appreciated. Apart from his guidance in these endeavors, I am also grateful for his introducing me to the many current topics in the area of software reuse.

My gratitude is also expressed toward Frances VanScoy and Doug Harms, members of my project committee. I consider it an honor to have them participate in this capacity.

PUBLICATIONS

Gray, Jeffrey 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.

Sitaraman, Murali, and Gray, Jeff, "Software Reuse: A Context for Introducing SE Principles in a Traditional CS Second Course," Department of Statistics and Computer Science, West Virginia University, Morgantown, WV, TR-93-2, March 1993, pp. 1-14.

Gray, Jeffrey G., "Ants Climbing Trees: Memory Management in Ada," *Embedded Systems Programming*, April 1990, pp. 23-26.

LIST OF FIGURES

TABLE OF CONTENTS

This page intentionally left blank.

CHAPTER 1

Introduction

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 totally displacing existing principles, is not easy. Software reuse provides an appropriate context for presenting principles of software design and specification, along with abstraction and encapsulation, in *traditional* freshmen computer science courses.

1.1 Why teach reuse in an introductory course?

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].

Introducing new principles, such as 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 readily applicable at the freshmen level. An answer to these problems lies in software reuse, which has been found to provide a most useful setting for reinforcing conventionally taught principles, such as abstraction and information hiding, and for introducing new software engineering principles.

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 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, as is the case with current approaches. 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 continually starting from scratch.

This report presents an outline of lectures and an example sequence of lab assignments for teaching the second course in computer science following a reuse-based approach. It includes our experience in adapting the approach to Ada for four semesters in the Department of Statistics and Computer Science at West Virginia University (Spring 92, Summer 92, Fall 92, and Spring 93). Some of the principles that are taught in the reuse-based second course at WVU include:

- The ability to understand abstract and formal specifications;
- Specification-based component reuse;
- Separation of the specification of a component from its implementation;
- Construction of new components by layering them on top of existing components; and
- Multiple implementations, with different efficiency characteristics, for a given specification.

1.2 Definition

Recent literature on software reuse contains several different definitions or classifications of the term [Krueger 92, Yourdon 92]. The definition of reuse used in this report is one which is component-based [Muralidharan 90b, Weide 91]. In this report, a reusable component is viewed as having two distinct elements: a formal specification and a certifiable implementation of that specification, possibly in the form of object code. All references to reuse discussed in this report are based only on the specification and performance characteristics of the implementation. The report concentrates on components which are *designed* for reuse since this is where the benefits are maximized [Hollingsworth 92b]. 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 in this report.

1.3 Benefits of a reuse-based approach

This section outlines 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 the 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 [Biggerstaff 89]. Such construction has the

potential to solve the most important problem facing software engineers today: how to produce high quality software on time. This futuristic view of software construction may come to resemble how other industries currently construct products. For example, in the same way that an individual can go to their local Radio Shack and purchase electronic components with specific characteristics, software engineers, or even computer hobbyists, may be able to go to a local software shop and purchase software components to their specifications. Thinking reuse early in the curriculum prepares students for this futuristic view of software construction. In fact, students will probably 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 new 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].

1.4 Organization of the report

The report is organized into the following chapters. Chapter 2 summarizes several technical foundations involved in the approach. Chapter 3 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 chapter is a discussion of laboratory assignments that have been used to help teach the approach. Chapter 4 enumerates the topics which are assumed to be covered in the first course. A second section of this chapter compares the approach to other methods for teaching the second course. The final chapter provides a summary. It also includes our experience in teaching the approach and offers suggestions for possible future work. Two appendices provide supportive material used in demonstrating the approach.

This page intentionally left blank.

CHAPTER 2

Technical Underpinnings

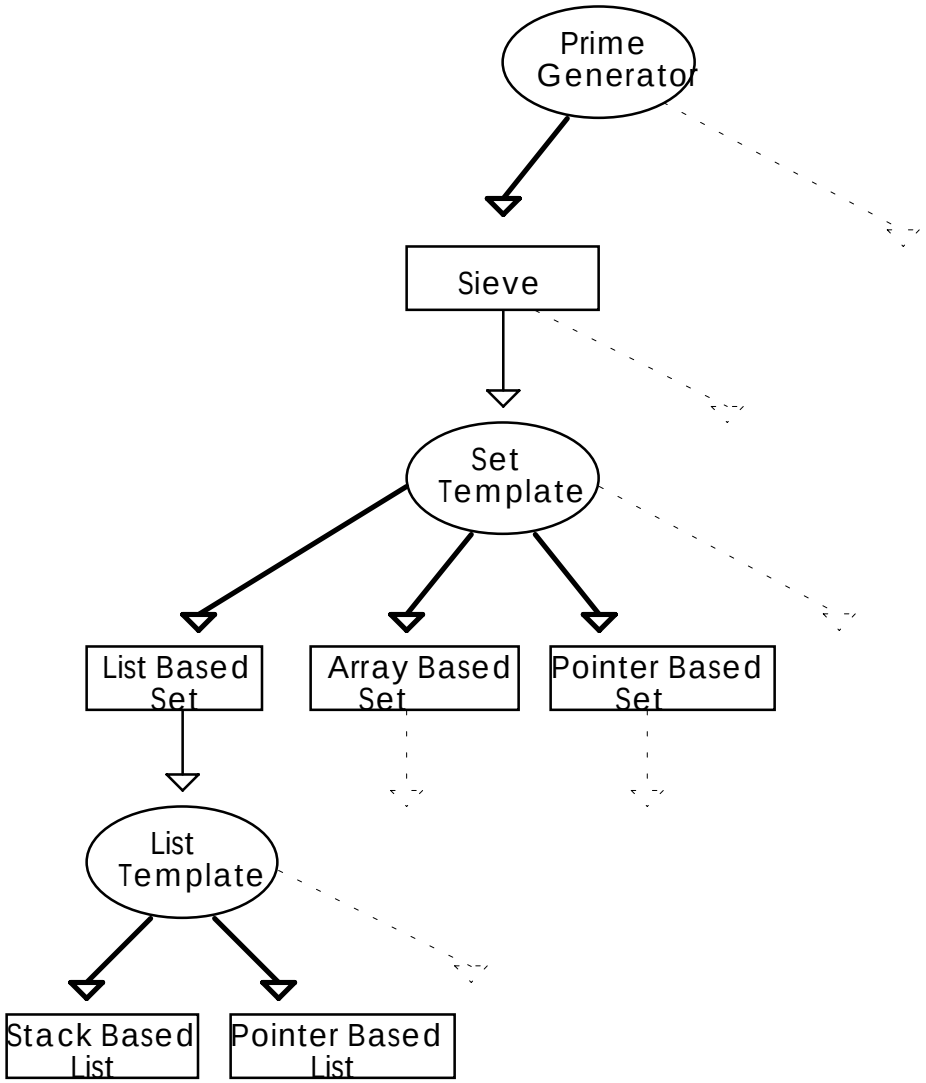
The specification-based approach to software reuse offers tremendous advantages in the development and maintenance of large software systems. The reuse-based course strictly adheres to this philosophy, and all reusable parts developed in the course are reused based only on their specifications. Throughout 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 chapter discusses the technical foundations of the approach.

2.1 The 3C model

A useful model for discussing some of the topics mentioned in this chapter can be found in the 3C model [Tracz 89]. This model, when applied to the underlying structure of a software system, can be used to illustrate the relationship among concepts (specifications), contents (implementations), and the context (environment) in which they occur. Conventions of the model state that concepts are to be represented by circles while contents are indicated by rectangles. Contexts, although useful in other descriptions, will not be utilized in the example from this section. As an illustration, the model will be used to describe the structure of a sample laboratory assignment.

In Figure 1, the model is used to represent the structure of a lab assignment whose purpose is to construct a prime number generator. Detailed descriptions of this assignment can be found in Chapter 3 and Appendix A. In this section, the assignment is used only to illustrate how the 3C model is applied. In the figure, an arrow from a concept to a content indicates that the content implements the concept. An arrow from a content to a concept means that the content uses the facilities provided by the concept. Dotted arrows which point to empty spaces indicate additional concepts/contents which were not included in order to keep the figure as simple as possible. For example, the content List_Based_Set can be used to implement the concept Set_Template. Likewise, the concept List_Template is used by the content List_Based_Set. It is obvious from the illustration that the model allows a concept to be realized by multiple contents, a useful attribute that is needed in the description of reusable software.

Although recent offerings of the second course at WVU have not utilized this model, a revised version of [Sitaraman 93b] plans to incorporate the model as a basis for introducing many of the principles presented in this report. The model is extremely useful in visually describing many of the key aspects of the principles taught in the course. The reader of this report may find the model useful in understanding the structure of the various lab assignments discussed in the next chapter.



The 3C Model Representation of a Prime Number Generator Using Sets
Figure 1

2.2 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].

In this report, the term "specification" will refer to both the syntactic and semantic interfaces of a reusable part. Figure 2 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 [Hollingsworth 92b, Sitaraman 93a, Weide 91], every abstract data type actually has an "abstract view" that is already familiar to the students. A comparison of RESOLVE and other specification approaches, such as those in [Wing 90], can be found in [Sitaraman 93a]. Other approaches, such as Larch and Z, may also be used if they can be presented in a way understandable to freshmen.

```
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_Comparator(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 mathematical Set of Items
```

The Specification of a Set Package
Figure 2

```
-- standard operations
```

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

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

  procedure Finalize(S : in out Set);

-- 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 2 (cont.)

It is important to note that before the students see the final version of Set_Template shown in Figure 2, 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 ∈ and ∪. 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. On a related note, discussions on the introduction of formal methods in graduate level courses can be found in [Garlan 92, Lutz 92].

2.3 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 [Hollingsworth 92b] and principles in [Harms 91, Reuse 92, SPC 89, Edwards 90].

The principles discussed so far in this chapter often synergetically combine to increase the reusability of software. For example, a by-product of specifying a software part using sound principles is that the design process becomes more manageable. This can be demonstrated by once again examining Set_Template. In enforcing information hiding on this component, by deferring the actual type representation until the package body, design issues concerned with multiple implementations for the same specification become more amenable.

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, unlike the copying approach [Harms 91]. It is possible to initialize and finalize storage for a Set in constant time using amortization techniques. These techniques are not taught as part of the course but descriptions of the method can be found in [Harms 89, Weide 91].

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*. Following the design guidelines of [Hollingsworth 92b], all imported and exported types are designed in this manner. Exported types, such as Set, are declared as *limited private* to allow clients to confidently reason about variables of the exported type. Exported types declared as being *private*, rather than *limited private*, would allow the client to access implicit operations, such as the assignment operator and tests for equality, which can be an obstacle to reasoning about clients that use the component. An additional guideline states that all exported types should be represented as pointers to some other representation type which is deferred until the implementation.

To discern the importance of declaring exported types as *limited private* instead of *private*, consider the following example: Suppose two variables, S1 and S2, are of type Set where S1 contains the values {1, 2, 3, 4, 5} while S2 is empty. In this example, assume that Set just happens to be declared as *private* to allow access to the assignment operator. Also, assume that the representation of Set is explicitly defined in the private part of the specification. Now, suppose that the following portion of code appears somewhere in the client and is executed with the above values for S1 and S2:

...

```
1)      Print_Set(S1);
2)      Print_Set(S2);
3)      S2 := S1;
4)      Integer_Set.Remove_Item(S1, 2);
5)      Print_Set(S1);
6)      Print_Set(S2);
        ...
```

Without following the guidelines just mentioned, what can a client confidently state about the behavior of the above code? As will be shown by simulated execution, the client can not assuredly reason about the behavior of this code since the assumptions from the example violate the previously stated guidelines. To illustrate this point, suppose the type Set is represented in a bounded form containing a record structure involving an array. Line 3 of the above code would then make a complete copy of S1 before assigning it to S2. The effect of line 4, then, would have no effect on the instance of S2. The output of lines 5-6 using this representation would be:

S1 => { 1, 3, 4, 5}, S2 => {1, 2, 3, 4, 5}

This simulated execution seems to capture the behavior that one would normally expect of a set package. However, suppose that instead of a bounded representation, a particular implementation of Set_Template utilized access types to represent the exported type Set. In this scenario, line 3 would then simply make S2 point to the same abstract value pointed to by S1. As is often the case when using pointers, this example introduces problems concerned with *aliasing*. Using this unbounded form, the statement found in line 4 would also have the side effect of altering the contents of S2. The final output of this representation would be:

S1 => {1, 3, 4, 5}, S2 => {1, 3, 4, 5}

In this case, S1 = S2, which is perhaps different from the behavior that one would intuitively expect of Set_Template. Inherent in the above discussion are differences between “deep” and “shallow” copying. In one case, a complete copy of a particular instance of the type is made while in the other case only pointers to the representation of the value are copied. As illustrated, this can introduce problems when trying to reason about the client of a component. Enforcing the use of *limited private* types removes this problem by eliminating the availability of the assignment operator. With *limited private* types, clients must use the standard data movement operation called Swap rather than the assignment operator. This not only corrects the aforementioned problems but also improves efficiency. For example, line 3 of the above code, when using the bounded form of a set, could be viewed as a concern for efficiency if, for instance, the sets contained complex elements rather than simple Integers.

Once the choice has been made to declare all exported types as *limited private*, imported types are also declared in this manner. Aside from maintaining consistency between imported and exported types, there is a very important reason for this design guideline:

composability. If imported types were allowed to be *private*, rather than *limited private*, then any implementation of the component would be allowed to use the assignment operator associated with the imported type. If the imported *private* type happens to be the exported *limited private* type of another component, then the Ada compiler will reject the composition of these components. Set_Template provides an example of the advantages of this guideline. By enforcing that the imported type Item be *limited private* in the Set_Template specification, new structures such as a Set of a Set of Integers are possible. Such compositions would otherwise be a violation of the semantics of Ada *private* and *limited private* types if these guidelines were not followed. Thus, by declaring both imported and exported types as *limited private*, one can more assuredly reason about the local behavior of a client of the component while being certain that all components are composable with each other. These important guidelines are often overlooked by some authors, thus, limiting the scope of reuse for their components.

An additional guideline that can be noticed from a study of Set_Template is that in addition to exporting the standard operations Initialize, Finalize, and Swap, the respective routines are also imported corresponding to the imported type. This is needed to allow storage management and data movement of the imported type from within the component. All temporary variables of the imported type are initialized and finalized. Rather than using the assignment operator to copy variables of the imported type, the Swap operation associated with Item is used instead. For example, the code to implement Add_Item would need to make use of Item_Swap in order to move the data value represented by X into the set. Since the value represented by X is swapped, and not copied, the specifications must indicate the returning value of X. In the case of Add_Item, the value of X returned from the procedure is some initial value obtained from the underlying type representation.

2.4 Implementation

Efficiency is an important characteristic of a reusable 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. Although more advanced techniques, such as constant initialization and finalization, are postponed until later in the curriculum, students still gain an appreciation for the importance of efficient implementations. For more information on designing efficient implementations, see [Bentley 82], which describes various techniques for increasing the efficiency of software, in general.

CHAPTER 3

Description of Materials for the Second Course

This chapter presents a course outline which has been used to teach the reuse-based course. The chapter also contains a sequence and description of sample laboratory assignments. The assignments chosen to be discussed in these sections are only a subset of the total collection of labs that we have developed but are those which best exemplify our overall goals.

3.1 Lecture outline

In [Sitaraman 93b], a complete set of materials, including home work assignments and lecture notes, is presented. The following outline illustrates the order in which materials are introduced in the course. This outline will be referenced throughout the report.

*1. The Engineering Metaphor — Week 1

*2. Syntactic and Semantic Specification — Week 1

*3. An Introduction to Formal Specification — Week 2

4. Protected, Abstract, and Generic Data Types — Week 3-4
    - Key example: Sets

5. Stacks: Design, Specification, and Implementation — Week 4-6
    - Specification of unbounded stacks
    - An introduction to design issues in specification
        - Defensive and non-defensive specifications
        - Primary and secondary stack operations
        - Guidelines
    - Problem solving using stacks
    - Specification of bounded stacks
    - An implementation of bounded stacks

6. An introduction to certification of correctness of implementations — Week 7
    - Key example: Certification of bounded stacks

7. An introduction to efficiency analysis of implementations — Week 7-8
    - Key example: An analysis of a bounded stack implementation

8. Queues — Week 8

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

9. Lists Week 9
Specification of lists
Secondary list operations
Problem solving using lists
List-based layered implementations of unbounded stacks and queues

10. Pointers as an implementation mechanism for unbounded ADT'sWeek 10-11
An introduction to access types and variables
An implementation of lists using pointers
Direct implementation of unbounded stacks and queues

*11. Recursion as a problem solving vehicle Week 11-12
Examples
Implementation of Stack, Queue, and List secondary operations

12. Trees Week 12-13
Specification
Tree traversals
An implementation of trees using pointers

13. Searching Week 14
Specification
Multiple implementation techniques
Linear and ordered linear search
Binary search trees
Searching using hash tables

14. Sorting Week 15
Specification
Multiple implementation techniques
Bubble Sort, Merge Sort, and Quick Sort

15. Course summary Week 15

**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.

3.2 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.

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 specification and implementations, Specification-based reuse, Understanding of abstract specifications, Construction of secondary operations.

Lab Assignment 2 — Postfix Evaluation of Super Integers

Problem: Given the specification for an unbounded Stack component, create a postfix evaluator for Super Integers.

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

Course Outline: Week 4-6

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

Lab Assignment 3 — Layered Stack Implementation

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

Lab instructor supplied items: Listing of 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 postfix evaluator.

Lab instructor supplied items: None

Course Outline: Week 10-12

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

3.3 Laboratory descriptions

This section provides a more detailed description of how three recurring lab themes are implemented. Each description contains three sub-sections which provide a list of the goals that are desired, a summary of the assignment, and possible variations. Provided in Appendix A are actual assignment descriptions which are given to the students. They provide specific examples of how the following themes are adapted in the course. Appendix B contains source code listings which offer solutions to the assignments from Appendix A.

Sample Lab Assignment 1
Student as client of a reusable component

Goals

To teach the following principles:

- The ability to understand formal and abstract expressions of a specification;
- Specification-based component reuse;
- The need for separating the specification of a component from its implementation;
- Acquaint each student with the notation of a specification language; and
- Construction of secondary operations.

Summary

Traditionally, the first assignment has always focused on solving some backtracking problem using a stack package provided by the instructor. Several different backtracking problems have been introduced to the students. Examples of problems used in the past include:

- The Eight Queens problem, whereby the students must find all possible combinations of placing eight queens on a chess board so that no queen can be attacked by another;
- Helping a mouse find a piece of cheese by moving through a maze which contains dead-ends; and
- Assisting a squirrel in climbing to the top of a tree, filled with many empty branches, to find an acorn.

This section, however, will describe an assignment which uses the Set_Template from Chapter 2. A description of several backtracking labs can be found in [Gray 93].

Sets can often be used as an aid toward solving graph problems. Additionally, they can be utilized to help solve other mathematical problems such as prime number generation. This sub-section provides a description of how students are first introduced to Set_Template by asking them to implement a solution of the Sieve of Eratosthenes algorithm. The students are given a copy of the Set_Template specification and told how to access the object code version of the body to allow for proper linking. They must construct a client program which utilizes Set_Template to generate a list of prime numbers.

When the students are given the Set_Template component, they are asked to view the specification as a contract between themselves and the implementer of the package, i.e., the lab instructor. This reinforces the notion that the developer and user of a component are often different people. They are assured that the Set_Template operations will work correctly provided they follow the specification. They must surmise on their own, by reading the specification, the syntax and meaning of each operation. Thus, the students get an early example of the importance of providing specifications which are unambiguous. As noted earlier, to add semantic information to Ada package specifications, we use a close dialect of the RESOLVE specification language [Hollingsworth 92b, Sitaraman 93a, Weide 91]. RESOLVE specifications are formal, but yet succinct and understandable by freshmen who have been briefly exposed to topics covered in discrete mathematics.

A final requirement of the assignment is to construct secondary operations for the component. The assignment directs the students in assembling six secondary operations, as shown in Figure 3. Several of these operations are needed in the driver program while others are included for completeness. Notice that the secondary operations require access to a package called Int_Sets. This is simply a pre-instantiation of Set_Template based on Integers. This component is also supplied by the lab instructor. The reliance on Int_Sets, however, forces the component to sacrifice genericity for the sake of simplicity. An alternative to providing Int_Sets would be to fully parameterize the needed set operations as generic parameters. This would allow for truly generic secondary operations. For an illustration of fully parameterized secondary operations, see [Gray 93].

```
with Int_Sets;
package Secondary_Set_Ops is

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

  procedure Clear_Set(S : in out Int_Sets.Set);
  --! ensures S = {}
```

```
  procedure Print_Set(S : in out Int_Sets.Set);
  --! ensurese S = #S and output = S

  procedure Union(S1, S2, S3 : in out Int_Sets.Set);
  --! ensures S1 = #S1 and S2 = #S2 and
  --          for all X : Integer, X is_an_element_of S3 iff
  --             X is_an_element_of S1 or X is_an_element_of S2

  procedure Intersection(S1, S2, S3 : in out Int_Sets.Set);
  --! ensures S1 = #S1 and S2 = #S2 and
  --          for all X : Integer, X is_an_element_of S3 iff
  --             X is_an_element_of S1 and X is_an_element_of S2

  procedure Difference(S1, S2, S3 : in out Int_Sets.Set);
  --! ensures S1 = #S1 and S2 = #S2 and
  --          for all X : Integer, X is_an_element_of S3 iff
  --             X is_an_element_of S1 and X is_NOT_an_element_of S2

end Secondary_Set_Ops;
```

The Specification of Secondary Set Operations
Figure 3

Variations

As stated above, backtracking problems have been traditionally used as the first laboratory assignment. Similar labs that make use of abstract data types other than a set or stack could easily be developed. For example, a lab instructor might give the students a queue package and ask them to write a client program that uses the component. They might be asked to use the queue to simulate a message passing system where requests to send and receive messages are handled and placed on a queue. Alternatively, they might use the queue to simulate a row of tellers at a bank where each teller has a queue of customers with individual requests to be serviced.

Sample Lab Assignment 2
Student as an implementer of a layered component

Goals

This lab instills the following principles, in addition to those already named:

- Construction of new components by layering them on top of existing components; and
- Multiple implementations, with different efficiency characteristics, for a given specification.

Summary

This sub-section describes an assignment that is along the same idea as the last assignment but offers somewhat of a change in the implementation of Set_Template. In this assignment, the students are given the specification to a list component shown in Figure 4. Implementation details about this component are hidden but access to the object code is provided to allow linking. They are then asked to use this component to actually implement the operations of Set_Template which they have already seen and used. They must implement the set operations solely by making calls to the operations of Figure 4 and are not allowed to use any form of pointers or array constructs. Thus, Set_Template is implemented by layering it on top of another component. The lab described in the previous sub-section is reused in this case by re-linking it with the new set implementation. The assignment should assist the students in beginning to think about how multiple implementations for the same specification are constructed [Sitaraman 92]. Also, the ease with which this lab can be completed should reinforce the idea of reuse. Students learn that it is often advantageous to make use of pre-existing standard components rather than "re-inventing the wheel".

The concept used to represent the list component in Figure 4 is different from the typical list concept presented in textbooks like [Booch 87]. In particular, the abstract idea of lists is presented without discussing pointers or access types. In the figure, a type called List is modeled as two strings of some other type named Item. These two strings are called, appropriately, "left" and "right". This view can be better understood if one envisions a conceptual cursor that separates the two strings. The package provides operations to move this cursor around the list as well as the ability to perform insertions and deletions. To illustrate this notion of a cursor, as it would apply to a list, examine the following instance of a list variable called L:

```
    |
3 4 | 7 2 6 3
    |
```

The value of L.Left would contain the two elements 3 and 4 while the value of L.Right would contain the four values 7, 2, 6, and 3. All insertions and deletions are performed to the right of the cursor. The Move_To_Left_End and Advance operations are used to traverse through the list. Using the above values of list L, a call to the Move_To_Left_End operation, followed by a call to Remove_Right, would result in L now resembling the following:

```
|
| 4 7 2 6 3
|
```

As a design principle, functions needed to check the requires clause of all operations are also included in the specification, e.g., At_Right_End. This illustrates the design principle of adequate functionality. The operation Swap_Rights will not be used in this assignment.

It has been provided for future assignments that may implement secondary operations. It has been found useful in constructing efficient implementations of a Copy_List operation [Weide 91].

The students have often found that this assignment can be completed within several hours. Almost all of the required set operations that they must write can be implemented with a small amount of code. For example, code to implement the Add_Item set operation would simply entail making the proper call to a corresponding list operation, i.e., Add_Right. A student only needs to understand the specification of the list component well enough to discern what calls correspond to similar notions within the set operations. This reinforces the concept of specification-based component reuse.

```
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

package List_Template is

  type List is limited private;
  --! abstract model: a pair of mathematical strings of Items,
  --                  named Left and Right

-- standard operations

  procedure Initialize(L : in out List);
  --! ensures L.Left = empty_string and L.Right = empty_string

  procedure Finalize(L : in out List);

  procedure Swap(L1, L2 : in out List);
  --! ensures L1 = #L2 and L2 = #L1
```

The Specification of a List Component
Figure 4

```
-- List-specific operations

  procedure Move_To_Left_End(L : in out List);
  --! ensures L.Left = empty_string and L.Right = #L.Left * #L.Right

  procedure Advance(L : in out List);
  --! requires L.Right /= empty_string
  --! ensures L.Left * L.Right = #L.Left * #L.Right and
  --!         thereExists X : Item, s.t., L.Left = #L.Left * X

  procedure Add_Right(L : in out List;
                      X : in out Item);
```

```
  --! ensures L.Left = #L.Left and L.Right = X * #L.Right and
  --!         X = initial_Item

  procedure Remove_Right(L : in out List;
                         X : in out Item);
  --! requires L.Right /= empty_string
  --! ensures L.Left = #L.Left and #L.Right = X * L.Right

  procedure Swap_Rights(L1, L2: in out List);
  --! ensures L1.Left = #L1.Left and L2.Left = #L2.Left and
  --!       L1.Right = #L2.Right and L2.Right = #L1.Right

  procedure At_Right_End(L : in out List;
                         At_End : in out Boolean);
  --! ensures (L = #L) and (At_End = true iff L.Right = empty_string)

  procedure At_Left_End(L : in out List;
                        At_End : in out Boolean);
  --! ensures (L = #L) and (At_End = true iff L.Left = empty_string)

  private

      type List_Rep;
      type List is access List_Rep;

end List_Template;
```

The Specification of a List Component
Figure 4 (cont.)

Variations

Although the above description layers a set package on top of a pre-existing list component, it is certainly plausible that one could also use alternative abstract data types. For instance, the students might be asked to implement a set layered upon a deque or a standard FIFO queue rather than a list. They also could be asked to analyze the efficiency of each operation in comparison to other strategies. With this in mind, the students will come to realize the need for efficient implementations since the client will probably decide to rewrite the component themselves if the component does not meet their performance requirements.

Sample Lab Assignment 3
Student as an implementer of a reusable component built from-scratch

Goals

In addition to the principles already named, this lab introduces the following new concepts:

- Use of access types to efficiently implement components from-scratch; and

- Introduction to the use of recursion.

Summary

This sub-section describes variations to a laboratory assignment that is often presented toward the end of the semester. It tends to focus more on specific details of implementing components, e.g., using pointers. It builds upon the previous two discussions by requiring the students to finally write lower level implementations of the list component. The set package will still be layered on top of the list but in this case the students acquire a feel for using access types to represent unbounded components.

Variations

Several possible variations could be suggested toward implementing the list in ways other than pointers. The list itself could be layered upon an already assembled component or the implementation details might opt to focus on an array based approach. Additionally, rather than concentrating on using a list to construct the set as done in the previous layered implementation, the idea of pointers could be used to implement the set directly, which would allow one to eliminate the need for implementing lists altogether. Also, secondary operations for lists could be requested similar to those described in the first assignment. Students might be asked to implement a secondary operation which performs a Copy_List, using the primary Swap_Rights operation. Correspondingly, the students may be asked to write secondary operations for the list package to provide the facilities for printing and reversing lists. At this time, the students may be required to write the secondary operations recursively.

This page intentionally left blank.

CHAPTER 4

Related Work

This chapter provides a discussion of work related to the reuse-based approach. The first section describes certain assumptions expected of a student upon entering the reuse-based course. A second section compares the reuse-based approach with other methods offered by either textbooks or course outlines obtained from instructors of related courses at other institutions.

4.1 Topics to be covered in the first course

Upon entering the reuse-based second course, several assumptions are made concerning certain topics that the student should be familiarized with from the first course. In our implementation of the approach at West Virginia University, students are first taught the basic constructs of the Ada language in the first course. The textbook used to teach these constructs in the first course has often been [Feldman 92a]. In order to ensure successful performance in the reuse-based second course, students should have been exposed to at least chapters one through nine from this text. Specifically, students are assumed to have a knowledge of the following material, with the corresponding chapters from [Feldman 92a] indicated:

- Fundamental concepts of computer systems (Chapter 1);
- Basic problem solving techniques (Chapter 3);
- Introduction to top-down and structured design (Chapter 3);
- Control constructs, e.g., 'if' and 'case' statements (Chapters 4 and 7);
- Iterative constructs, e.g., various 'loop' constructs (Chapters 5 and 6);
- Data structures using scalar and composite types (Chapters 7 and 8); and
- Basic debugging strategies (Chapters 2, 5, 6 and 9).

The above material covers a large portion of the suggested recommended curriculum for the first course, as proposed in [Koffman 84]. We do not assume, however, any coverage of access types or generics from the first course. Furthermore, although recursion could be introduced in the first course (e.g., chapter thirteen of [Feldman 92a]), we have opted to postpone discussion of this problem solving tool until the second course. We also postpone discussion of algorithm analysis until the second course, contrary to the suggestion of [Koffman 84]. In the future, the topic of procedure specifications will probably be moved into the first course.

An interesting discussion which focuses on the first course can be found in [Feldman 93], where an approach is described for designing and teaching the first course in a large class, i.e., over 350 students per quarter. It also provides several suggestive laboratory assignments which could be profitably utilized to inculcate the material previously enumerated. Also, [Feldman 92b] provides first person profiles of how twelve institutions incorporate Ada into their first and second courses.

4.2 Distinguishing features

This section identifies several distinguishing characteristics of the reuse-based approach which differ from traditional methods for teaching the second course. A comparison is made between the outline presented in Chapter 3 and various textbooks/outlines used to teach the second course in other contexts.

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 chapter. 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.

An approach for teaching the second course, which shares numerous similarities with the reuse-based approach, can be found in [Beidler 93], an approach which the authors term object-based. A striking similarity between the object-based and reuse-based approach is that both methods provide semantic specifications. Beidler's approach, however, opts to use informal assertions while our specifications are more formal. As an example, Figure 5 shows how the Pop procedure for a Stack is specified in [Beidler 93]. Figure 6 shows our specification of Pop. Here, the type Stack is modeled using mathematical string theory. The ensures clause contains an assertion which makes use of the concatenation operator, “*”.

```
procedure Pop(An_Element :    out Object_Type;
              The_Stack  : in out Stack_Type);
```

```
-------------------------------------------------------------------
-- Pre  Cond  : The_Stack is not empty
-- Post Cond  : Top of stack removed and placed in An_Element
-- Exceptions : Stack_Is_Empty
-------------------------------------------------------------------
```

A Procedure Specification Using the Object-based Approach
Figure 5

There are several important differences between the design of components presented in [Beidler 93] and the ones we have used. For example, many designs, including those in [Beidler 93], incorporate what we term secondary operations in the same specification as primary operations. Furthermore, a notable difference in the implementation of the object-based approach is the construction of student teams. The reuse-based approach, as stated earlier, forms a team only between the student and instructor, rather than among students.

```
procedure Pop(S : in out Stack;
              X : in out Item);
--! requires S /= empty_string
--! ensures #S = S * X
```

A Procedure Specification Using the Reuse-based Approach
Figure 6

In other details related to the outline of Chapter 3, 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.

This page intentionally left blank.

CHAPTER 5

Experience and Conclusions

Over the past four semesters, the approach described in this project report has been used to teach the second course in computer science at the West Virginia University. 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 report 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. By utilizing software reuse as the desired context for introducing traditional principles, core software engineering principles are also introduced at an early stage rather than abstaining from such material until the senior year.

The reuse-based approach instills software engineering principles without displacing traditional concepts taught in a second course, and thus provides a practical approach for adaptation in most schools. Promising feedback has been received from professors at other schools. A range of research and teaching schools, including The Ohio State University, Indiana University Southeast and Muskingum College, are committed to experimenting the reuse-based approach in the Fall of 1993.

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. The approach seems to minimize what Krueger terms *cognitive distance* [Krueger 92], a measure of the intellectual effort expended to take a software system from one level to another.

There is still much work that needs to be done with the implementation of the approach. For example, most of the proposed laboratory assignments that were mentioned under the *Variations* sections of Chapter 3 need to be constructed. Principles that can be profitably taught in the first course are also being investigated. It is possible to “push” some of these principles down into the first course, such as from chapters on procedure specifications and recursion. Additionally, we are currently keeping track of previous students who have taken the course using the approach. It would be interesting to investigate on how the approach affected students in courses encountered later in the curriculum. Current feedback from students, obtained through confidential evaluations, has been very positive.

Finally, though the overall approach is language-independent, Ada has proved to be a most suitable language for teaching these principles to freshmen students. An additional area for future research might explore how the approach could be implemented using languages other than Ada, e.g., C++, Pascal, and Ada 9X.

APPENDIX A

Sample Lab Descriptions

Included in this appendix is a sample sequence of three laboratory assignment descriptions. These descriptions illustrate the format of how assignments are presented to students. Each assignment description provides students with the following information:

- An introduction to the assignment;
- A list of the goals, or principles, that the assignment strives to teach;
- A listing of supportive source code written by the lab instructor;
- A description of the deliverables that they must provide; and
- Instructions and suggestions for completing the assignment.

Sample Lab Assignment 1
Sieve of Eratosthenes

Computer Science 16
Summer 1993

Due Date

This project is due in three weeks.

Principles Taught

- The ability to understand formal and abstract expressions of a specification. You will become acquainted with the notation of a specification language;
- Specification-based component reuse;
- The construction of secondary operations; and
- Introduction to the Set_Template.

Project Description

An ancient Greek astronomer/mathematician named Eratosthenes invented a method for calculating prime numbers. For those of you who forget, a prime number is defined simply as an integer that can be evenly divided by no other whole number other than itself or 1. For example, the first five prime numbers are 2, 3, 5, 7, and 11. The algorithm works by first inserting all prospective numbers into a set. As prime numbers are discovered, all multiples of the primes are then removed from this prospective set. In a sense, during each iteration, the algorithm sifts through and removes all prospective numbers which are not prime.

For this assignment, you must implement the Sieve of Eratosthenes algorithm using a Set_Template. This template will be provided by the instructor. You also must construct several secondary operations that involve sets.

A general outline of the Sieve of Eratosthenes algorithm follows:

1.a) In the set called Prospects, place all numbers that fall within the range of numbers to be searched. For example, you will be required to find all primes from 1 to 100 so you must first place these numbers in Prospects.

1.b) Make sure the set called Primes is empty.

1.c). Assign the first prime number (2) to a variable called Next.

2) Continue incrementing Next until the current value of Next is a member of Prospects.

3) The value of Next which survived step 2 must be a prime number. Add Next to the set called Primes.

4) Remove all multiples of Next (including Next itself) from the set Prospects.

5) If Prospects is not empty, then go to step 2.

6) Print the set Primes.

Support Code

The following represents a package that is already implemented and provided for you. You may assume that the implementation of this package is correct.

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

  procedure Finalize(S : in out Set);



-- set operations
```

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

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

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

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

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

private

  type Set_Rep;
  type Set is access Set_Rep;

end Set_Template;
```

An instantiation of the above template, using integers, is also provided. The package is called Int_Sets and you will be instructed on how to access the object code.

The following file (named SETSEC.LIB in my account) represents the specifications for a package that you must implement. One of the operations, i.e., Union, is already written to give you an idea of what is needed. You must complete the remaining operations.

```
with Int_Sets;
package Secondary_Set_Ops is

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

  procedure Clear_Set(S : in out Int_Sets.Set);
  --! ensures S = {}

  procedure Print_Set(S : in out Int_Sets.Set);
  --! ensurese S = #S and output = S

  procedure Union(S1, S2, S3 : in out Int_Sets.Set);
  --! ensures S1 = #S1 and S2 = #S2 and
  --          for all X : Integer, X is_an_element_of S3 iff
  --             X is_an_element_of S1 or X is_an_element_of S2

  procedure Intersection(S1, S2, S3 : in out Int_Sets.Set);
  --! ensures S1 = #S1 and S2 = #S2 and
  --          for all X : Integer, X is_an_element_of S3 iff
  --             X is_an_element_of S1 and X is_an_element_of S2
```

```
  procedure Difference(S1, S2, S3 : in out Int_Sets.Set);
  --! ensures S1 = #S1 and S2 = #S2 and
  --          for all X : Integer, X is_an_element_of S3 iff
  --             X is_an_element_of S1 and X is_NOT_an_element_of S2

end Secondary_Set_Ops;
```

Deliverables

In addition to submitting a sample run of your program, you must provide the source listing for the files SIEVE.PRC (the driver program) and SETSEC.PKG (the secondary operations implementation).

Your sample run should resemble the following:

```
$ run sieve

What range do you want to search for primes (1-N) ? 100

97   89   83   79   73   71   67   61   59   53   47   43   41
37   31   29   23   19   17   13   11    7    5    3    2
```

Specific Instructions

1. Set up a link into my account so that you have access to Set_Template and Int_Sets. This is done with the following command:

    $ acs enter unit un036:[272.adalib] Set_Template, Int_Sets

The above command allows you to make use of Set_Template without having to compile the files yourself.

2. Copy the files SETSEC.LIB and SETSEC.PKG from my account. You then need to compile the SETSEC.LIB file. Your first programming task is to then complete the implementation of the operations that I have not already written for you. The operations in SETSEC.PKG are to be written by making calls to the Set_Template operations.

3. Implement the driver routine. Use the algorithm presented in the Project Description as a guide.

4. Compile all needed files and link the driver. Run the driver and turn in to me, within three weeks, the required deliverables.

Sample Lab Assignment 2
Layered Implementations

Computer Science 16
Summer 1993

Due Date

This project is due in one week.

Principles Taught

In addition to borrowing some of the principles taught in the first assignment, this second assignment introduces the following new principles:

- The layered approach toward software construction;
- The need for separating the specification of a component from its implementation;
- Multiple implementations for the same specification; and
- Introduction to the List_Template.

Project Description

In the last assignment, the Set_Template package was provided for you. In this assignment, you get the chance to write your own implementation of Set_Template using a layered approach. You will be given access to a List_Template that you must use to write the set operations. Your implementation of the set operations must make calls to the List_Template only; no other method is allowed. Hopefully, you will find that this approach is not only easy, but also very productive in terms of the amount of time needed to complete the assignment. You should be able to complete all the requirements within one to two hours. After the Set_Template package is completed, you will relink your new version with the first assignment.

Support Code

The following represents a package that is already implemented and provided for you. You may assume that the implementation of this package is correct.

```
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

package List_Template is

  type List is limited private;
  --! abstract model: a pair of mathematical strings of Items,
  --!                 named Left and Right

-- standard operations

  procedure Initialize(L : in out List);
  --! ensures L.Left = empty_string and L.Right = empty_string

  procedure Finalize(L : in out List);

  procedure Swap(L1, L2 : in out List);
  --! ensures L1 = #L2 and L2 = #L1

-- List-specific operations

  procedure Move_To_Left_End(L : in out List);
  --! ensures L.Left = empty_string and L.Right = #L.Left * #L.Right

  procedure Advance(L : in out List);
  --! requires L.Right /= empty_string
  --! ensures L.Left * L.Right = #L.Left * #L.Right and
  --!         thereExists X : Item, s.t., L.Left = #L.Left * X

  procedure Add_Right(L : in out List;
                      X : in out Item);
  --! ensures L.Left = #L.Left and L.Right = X * #L.Right and
  --!         X = initial_Item

  procedure Remove_Right(L : in out List;
                         X : in out Item);
  --! requires L.Right /= empty_string
  --! ensures L.Left = #L.Left and #L.Right = X * L.Right

  procedure Swap_Rights(L1, L2: in out List);
  --! ensures L1.Left = #L1.Left and L2.Left = #L2.Left and
  --!       L1.Right = #L2.Right and L2.Right = #L1.Right

  procedure At_Right_End(L : in out List;
                         At_End : in out Boolean);
  --! ensures (L = #L) and (At_End = true iff L.Right = empty_string)

  procedure At_Left_End(L : in out List;
                        At_End : in out Boolean);
  --! ensures (L = #L) and (At_End = true iff L.Left = empty_string)

  private

      type List_Rep;
```

```
      type List is access List_Rep;
end List_Template;
```

Deliverables

In addition to submitting a sample run of your program, you must also provide the source listing for the file SET.PKG, which implements the set operations using a layered approach.

Specific Instructions

1. Set up a link into my account so that you have access to List_Template. This is accomplished in the same manner as the previous assignment by using the acs enter unit command.

2. Copy the file SET.PKG from my account. This file provides the instantiation needed to layer the set operations using a list. The individual operations, which are stubbed out, are to be completed by you.

3. Compile all needed files and relink with the first assignment. Run the driver and turn in to me, by next week, the required deliverables.

Sample Lab Assignment 3
"From-scratch" Implementations

Computer Science 16
Summer 1993

Due Date

This project is due in two weeks.

Principles Taught

In addition to borrowing some of the principles taught in the first and second assignments, this final assignment introduces the following new principles:

- Construction of software using raw implementations (e.g. access types);
- Dynamic memory deallocation; and
- Introduction to recursive programming.

Project Description

In this assignment, you will continue to implement a component. The method of implementation for this assignment, however, will be one which is based upon a raw approach to software construction. What this means is that you will be required to construct the component using only Ada constructs (i.e. access types or arrays). You are not allowed to use the layered approach in this assignment. You will probably find that this method is much more difficult than the one used in assignment two. There are, however, situations in which the raw approach must be used, either due to efficiency reasons or the lack of additional components suitable for layering. The raw implementation that you will complete for this assignment is the List_Template that you have already seen and used. In this case, you must implement the list operations using only access types. In addition, you are required to fully implement the standard Finalize routine for Lists. This will be accomplished using the generic procedure Unchecked_Deallocation. You will be given more information about this requirement in the lab class. A final requirement of the assignment asks you to rewrite the Print_Set operation from assignment one. The twist to the new implementation is that it must be written recursively.

Support Code

No new support code is needed to complete this assignment.

Deliverables

In addition to submitting a sample run of your program, you must also provide the source listing for the file LIST.PKG, which implements the set operations using access types. You also must deliver the source code changes to Print_Set.

Specific Instructions

1. Copy the file LIST.PKG from my account. This file provides the operations that you need to write. Complete the package using access types.

2. Rewrite the Print_Set operation recursively.

3. Compile all needed files and relink with the second assignment. Run the driver and turn in to me, within two weeks, the required deliverables.

APPENDIX B

Lab Solutions

The contents of this appendix provide partial solutions to the laboratory assignments described in Appendix A. Rather than presenting the complete code for each assignment, only the student deliverables are given. A brief description of the code precedes each source listing.

Assignment 1

The student deliverables for assignment one include the construction of the driver program in addition to the implementation of various secondary set operations. The specifcation for Set_Template is provided to the students in text form while the implementation of Set_Template is only provided in object code for linking purposes. An instantiation of Set_Template for integers is also provided to the students in object form and is called Int_Sets. The following pages present a possible solution for the driver routine and the implementation of the secondary set operations.

SIEVE.PRC

```
with Int_Sets;
with Secondary_Set_Ops;
with Text_IO;

procedure Sieve is

Primes    : Int_Sets.Set;
Prospects : Int_Sets.Set;
Empty  : Boolean;
Member : Boolean;
X, Y, N : Integer;
Next    : Integer;

package Int_IO is new Text_IO.Integer_IO(Integer);

begin

  Int_Sets.Initialize(Primes);
  Int_Sets.Initialize(Prospects);

  Text_IO.Put("What range do you want to search for primes (1-N) ? ");
  Int_IO.Get(N);

  ------------------------------------------------------------
  -- First, begin by filling the set Prospects with all the needed
  -- numbers.
  ------------------------------------------------------------

  X := 2;
  while X <= N
  loop

    Y := X;
    Int_Sets.Add_Item(Prospects, Y);
    X := X + 1;

  end loop;
```

SIEVE.PRC (cont.)

```
------------------------------------------------------------
-- The following block of code computes the prime number between 1-N.
------------------------------------------------------------

Next := 2;
loop

  ------------------------------------------------------------
  -- Find the next number contained in the set Prospects...
  ------------------------------------------------------------

  Int_Sets.Is_Member(Prospects, Next, Member);
  loop

    exit when Member;

    Next := Next + 1;
    Int_Sets.Is_Member(Prospects, Next, Member);

  end loop;

  ------------------------------------------------------------
  -- The number just found must be a prime. Record the result.
  ------------------------------------------------------------

  Y := Next;
  Int_Sets.Add_Item(Primes, Y);

  ------------------------------------------------------------
  -- The next block of code removes all multiples of the current
  -- prime from the set of prospective numbers.
  ------------------------------------------------------------

  X:= Next;
  while X <= N
  loop

    Int_Sets.Remove_Item(Prospects, X);
    X := X + Next;

  end loop;

  ------------------------------------------------------------
  -- Finally, exit when there are no more prospects...
  ------------------------------------------------------------

  Int_Sets.Is_Empty(Prospects, Empty);
  exit when Empty;

end loop;
```

SIEVE.PRC (cont.)

```
------------------------------------------------------------
-- Print the results and exit...
------------------------------------------------------------

Text_IO.Put("The prime numbers, between 1-");
```

```
   Int_IO.Put(N, 0);
   Text_IO.Put(" are:");
   Text_IO.New_Line(3);

   Secondary_Set_Ops.Print_Set(Primes);

   Int_Sets.Finalize(Primes);
   Int_Sets.Finalize(Prospects);


end Sieve;
```

SETSEC.PKG

```
with Text_IO;
package body Secondary_Set_Ops is

  package Int_IO is new Text_IO.Integer_IO(Integer);

  procedure Copy(S1, S2 : in out Int_Sets.Set) is

  Temp  : Int_Sets.Set;
  Empty : Boolean;
  X, Y  : Integer;

  begin

    Int_Sets.Initialize(Temp);

    ------------------------------------------------------------
    -- The following loop moves all of the items from set S1 into
    -- set Temp.
    ------------------------------------------------------------

    Int_Sets.Is_Empty(S1, Empty);
    while not Empty
    loop

      Int_Sets.Remove_Any_One_Item(S1, X);
      Int_Sets.Add_Item(Temp, X);
      Int_Sets.Is_Empty(S1, Empty);

    end loop;

    ------------------------------------------------------------
    -- The following loop moves all of the items from set Temp into
    -- sets S1 and S2.
    ------------------------------------------------------------

    Int_Sets.Is_Empty(Temp, Empty);
    while not Empty
    loop

      Int_Sets.Remove_Any_One_Item(Temp, X);
      Y := X;
      Int_Sets.Add_Item(S1, X);
      Int_Sets.Add_Item(S2, Y);
      Int_Sets.Is_Empty(Temp, Empty);

    end loop;

    Int_Sets.Finalize(Temp);

  end Copy;
```

SETSEC.PKG (cont.)

```
  procedure Clear_Set(S : in out Int_Sets.Set) is
```

```
Empty : Boolean;
X     : Integer;

begin

  ------------------------------------------------------------
  -- The following loop simply continues to remove items from set
  -- S until there are no items left.
  ------------------------------------------------------------

  Int_Sets.Is_Empty(S, Empty);
  while not Empty
  loop

    Int_Sets.Remove_Any_One_Item(S, X);
    Int_Sets.Is_Empty(S, Empty);

  end loop;

end Clear_Set;


procedure Print_Set(S : in out Int_Sets.Set) is

Temp  : Int_Sets.Set;
Empty : Boolean;
X     : Integer;

begin

  Int_Sets.Initialize(Temp);

  ------------------------------------------------------------
  -- This code makes a copy of the set S. It then removes and prints
  -- all items from the copied set.
  ------------------------------------------------------------

  Copy(S, Temp);

  Int_Sets.Is_Empty(Temp, Empty);
  while not Empty
  loop

    Int_Sets.Remove_Any_One_Item(Temp, X);
    Int_IO.Put(X, 0); Text_IO.New_Line;
    Int_Sets.Is_Empty(Temp, Empty);

  end loop;
```

SETSEC.PKG (cont.)

```
  Int_Sets.Finalize(Temp);

end Print_Set;
```

```
procedure Union(S1, S2, S3 : in out Int_Sets.Set) is

Temp1, Temp2 : Int_Sets.Set;
Empty : Boolean;
X     : Integer;

begin

  Int_Sets.Initialize(Temp1);
  Int_Sets.Initialize(Temp2);

  Clear_Set(S3);
  Copy(S1, Temp1);
  Difference(S1, S2, Temp2);

  -------------------------------------------------------------
  -- The following loop copies all items found in S1 into the
  -- set S3.
  -------------------------------------------------------------

  Int_Sets.Is_Empty(Temp1, Empty);
  while not Empty
  loop

    Int_Sets.Remove_Any_One_Item(Temp1, X);
    Int_Sets.Add_Item(S3, X);
    Int_Sets.Is_Empty(Temp1, Empty);

  end loop;

  -------------------------------------------------------------
  -- The following loop copies all items found in S2, but
  -- not in S1, into the set S3.
  -------------------------------------------------------------

  Int_Sets.Is_Empty(Temp2, Empty);
  while not Empty
  loop

    Int_Sets.Remove_Any_One_Item(Temp2, X);
    Int_Sets.Add_Item(S3, X);
    Int_Sets.Is_Empty(Temp2, Empty);

  end loop;
```

SETSEC.PKG (cont.)

```
  Int_Sets.Finalize(Temp1);
  Int_Sets.Finalize(Temp2);

end Union;


procedure Intersection(S1, S2, S3 : in out Int_Sets.Set) is

Temp   : Int_Sets.Set;
Empty  : Boolean;
Member : Boolean;
```

```
X      : Integer;

begin

  Int_Sets.Initialize(Temp);

  Clear_Set(S3);
  Copy(S1, Temp);

  ------------------------------------------------------------
  -- The following loop moves items from S1. The items from S1
  -- are only added to S3 if they can also be found in S2.
  ------------------------------------------------------------

  Int_Sets.Is_Empty(Temp, Empty);
  while not Empty
  loop

    Int_Sets.Remove_Any_One_Item(Temp, X);
    Int_Sets.Is_Member(S2, X, Member);

    if Member then

      Int_Sets.Add_Item(S3, X);

    end if;

    Int_Sets.Is_Empty(Temp, Empty);

  end loop;

  Int_Sets.Finalize(Temp);

end Intersection;
```

SETSEC.PKG (cont.)

```
procedure Difference(S1, S2, S3 : in out Int_Sets.Set) is

Temp   : Int_Sets.Set;
Empty  : Boolean;
Member : Boolean;
X      : Integer;

begin

  Int_Sets.Initialize(Temp);

  Clear_Set(S3);
  Copy(S1, Temp);

  ------------------------------------------------------------
```

```
    -- The following loop moves items from S1. The items from S1
    -- are only added to S3 if they can NOT be found in S2.
    ------------------------------------------------------------

    Int_Sets.Is_Empty(Temp, Empty);
    while not Empty
    loop

      Int_Sets.Remove_Any_One_Item(Temp, X);
      Int_Sets.Is_Member(S2, X, Member);

      if not Member then

        Int_Sets.Add_Item(S3, X);

      end if;

      Int_Sets.Is_Empty(Temp, Empty);

    end loop;

    Int_Sets.Finalize(Temp);

  end Difference;


end Secondary_Set_Ops;
```

Assignment 2

As stated in the assignment description from Appendix A, assignment two is a relativiely easy assignment for the students to complete. The only true deliverable is the layered Set implementation. The following code represents a possible implementation of how a Set_Template could be layered using List_Template.

SET.PKG

```
with List_Template;
package body Set_Template is

-----------------------------------------------------------------
-- Instantiate the needed list package and complete the definition
-- of the set representation; in this case, layered on a list.
-----------------------------------------------------------------


  package Set_By_List is new List_Template(Item, Item_Initialize,
                                           Item_Finalize, Item_Swap);


  type Set_Rep is
    record

      The_List : Set_By_List.List;

    end record;


  procedure Initialize(S : in out Set) is

  begin

    ------------------------------------------------------------
    -- Allocate the space needed for this variable and then initialize
    -- the list field...
    ------------------------------------------------------------

    S := new Set_Rep;
    Set_By_List.Initialize(S.The_List);

  end Initialize;
```

SET.PKG (cont.)

```
procedure Swap(S1, S2 : in out Set) is

Temp : Set;

begin

  Temp := S1;
  S1 := S2;
  S2 := Temp;

end Swap;


procedure Finalize(S : in out Set) is

begin

  Set_By_List.Finalize(S.The_List);

end Finalize;


procedure Add_Item(S : in out Set;
                   X : in out Item) is

begin

  ------------------------------------------------------------
  -- Simply make a corresponding call to add an element to a list
  -- and then initialize the incoming item, as indicated in the specs.
  ------------------------------------------------------------

  Set_By_List.Add_Right(S.The_List, X);
  Item_Initialize(X);

end Add_Item;


procedure Remove_Item(S : in out Set;
                      X : in outItem) is

Temp : Item;
At_End : Boolean;
Answer : Boolean;

begin
```

SET.PKG (cont.)

```
  ------------------------------------------------------------
  -- Begin the search for the item at the beginning of the list.
  ------------------------------------------------------------
```

```
  Set_By_List.Move_To_Left_End(S.The_List);

  ------------------------------------------------------------
  -- The following loop continually removes items from the list
  -- and compares them to the value passed in X. If the value in
  -- X is found, then it removes the item from the list. All other
  -- items in the list are unaffected.
  ------------------------------------------------------------

  Set_By_List.At_Right_End(S.The_List, At_End);
  while not At_End
  loop

    Set_By_List.Remove_Right(S.The_List, Temp);

    Item_Comparator(Temp, X, Answer);
    if Answer then

      exit;

    else

      Set_By_List.Add_Right(S.The_List, Temp);
      Set_By_List.Advance(S.The_List);

    end if;

    Set_By_List.At_Right_End(S.The_List, At_End);

  end loop;

end Remove_Item;


procedure Remove_Any_One_Item(S : in out Set;
                              X : in outItem) is

begin

  ------------------------------------------------------------
  -- The following two lines of code simply return the first value
  -- that is stored in the list. This is all that the specifications
  -- require.
  ------------------------------------------------------------
```

SET.PKG (cont.)

```
  Set_By_List.Move_To_Left_End(S.The_List);
  Set_By_List.Remove_Right(S.The_List, X);

end Remove_Any_One_Item;


procedure Is_Member(S : in out Set;
                    X : in out Item;
```

```
                    Member : in out Boolean) is

Temp : Item;
At_End : Boolean;
Answer : Boolean;

begin

 ------------------------------------------------------------
  -- Begin the search for the item at the beginning of the list.
  ------------------------------------------------------------

  Set_By_List.Move_To_Left_End(S.The_List);

  Member := False;

  ------------------------------------------------------------
  -- The following loop continually removes items from the list
  -- and compares them to the value passed in X. The boolean Member
  -- is updated accordingly based upon whether the value in X is
  -- found.
  ------------------------------------------------------------

  Set_By_List.At_Right_End(S.The_List, At_End);
  while not At_End
  loop

    Set_By_List.Remove_Right(S.The_List, Temp);

    Item_Comparator(Temp, X, Answer);
    if Answer then

      Member := True;
      Set_By_List.Add_Right(S.The_List, Temp);
      exit;

    else

      Set_By_List.Add_Right(S.The_List, Temp);
      Set_By_List.Advance(S.The_List);

    end if;
```

SET.PKG (cont.)

```
    Set_By_List.At_Right_End(S.The_List, At_End);

  end loop;

end Is_Member;


procedure Is_Empty(S : in out Set;
                   Empty : in out Boolean) is

begin
```

```
    ------------------------------------------------------------
    -- The following two lines check to see if the Set S is empty.
    -- To determine if S is empty, the two lines simply check if
    -- there are any elements at the front of the list.
    ------------------------------------------------------------

    Set_By_List.Move_To_Left_End(S.The_List);
    Set_By_List.At_Right_End(S.The_List, Empty);

  end Is_Empty;


end Set_Template;
```

Assignment 3

This last example assignment requires the students to write a recusrive implementation of the Print_Set operation from assignment one. Also, it is in this assignemnt where students are introduced to constructing implementations using access types. The following pages first demonstrate a recursive solution to Print_Set. An implementation of List_Template, using access types, is then given.

PRINT_SET.PRC

```
  procedure Recursive_Print_Set(S : in out Int_Sets.Set) is

  Empty : Boolean;
  X     : Integer;

  begin

    Int_Sets.Is_Empty(S, Empty);
    if not Empty then

      Int_Sets.Remove_Any_One_Item(S, X);

      Int_IO.Put(X, 0);
      Text_IO.New_Line;

      Recursive_Print_Set(S);

      Int_Sets.Add_Item(S, X);

    end if;

  end Recursive_Print_Set;
```

LIST.PKG

```
with Unchecked_Deallocation;
package body List_Template is

  ---------------------------------------------------------------
  -- The following type declarations are used in completing the
  -- definition of List_Rep.
  ---------------------------------------------------------------

  type Node;
  type Node_Access is access Node;
```

LIST.PKG (cont.)

```
type Node is
  record

    The_Item : Item;
    Next     : Node_Access;

  end record;

type List_Rep is
  record

    Front  : Node_Access;
    Cursor : Node_Access;

  end record;

procedure Free_Node is new Unchecked_Deallocation(Node, Node_Access);

procedure Initialize(L : in out List) is

begin

  ------------------------------------------------------------
  -- Creates a dummy node which makes other operations easier to
  -- write.
  ------------------------------------------------------------

  L := new List_Rep;
  L.Front := new Node;
  L.Front.Next := null;
  L.Cursor := L.Front;

end Initialize;


procedure Finalize(L : in out List) is

Temp : Node_Access;

begin

  ------------------------------------------------------------
  -- The following code finalizes a list in linear time. Constant
  -- time finalization is not covered in the reuse-based second
  -- course.
  ------------------------------------------------------------

  L.Cursor := L.Front;
```

LIST.PKG (cont.)

```
  while L.Cursor.Next /= null
  loop

    Temp := L.Cursor.Next;
```

```
    L.Cursor.Next := L.Cursor.Next.Next;
    Free_Node(Temp);

  end loop;

end Finalize;


procedure Swap(L1, L2 : in out List) is

Temp : List;

begin

  Temp := L1;
  L1 := L2;
  L2 := Temp;

end Swap;


procedure Move_To_Left_End(L : in out List) is

begin

  ------------------------------------------------------------
  -- The following moves the cursor to the front, or left end,
  -- of the list.
  ------------------------------------------------------------

  L.Cursor := L.Front;

end Move_To_Left_End;


procedure Advance(L : in out List) is

begin

  ------------------------------------------------------------
  -- The following simply moves the cursor to the next element.
  ------------------------------------------------------------

  L.Cursor := L.Cursor.Next;

end Advance;
```

LIST.PKG (cont.)

```
procedure Add_Right(L : in out List;
                    X : in out Item) is

Temp : Node_Access;

begin

  ------------------------------------------------------------
  -- The following places the value, stored in X, to the right
```

```
  -- of the cursor. Note the use of call by swapping inherent in
  -- the code.
  ------------------------------------------------------------

  Temp := new Node;
  Item_Swap(Temp.The_Item, X);
  Temp.Next := L.Cursor.Next;
  L.Cursor.Next := Temp;
  Item_Initialize(X);

end Add_Right;


procedure Remove_Right(L : in out List;
                       X : in out Item) is

begin

  ------------------------------------------------------------
  -- The following removes the item to the right of the cursor
  -- and stores the value in X.
  ------------------------------------------------------------

  Item_Swap(X, L.Cursor.Next.The_Item);
  L.Cursor.Next := L.Cursor.Next.Next;

end Remove_Right;


procedure Swap_Rights(L1, L2 : in out List) is

Temp : Node_Access;

begin

  Temp := L1.Cursor.Next;
  L1.Cursor.Next := L2.Cursor.Next;
  L2.Cursor.Next := Temp;

end Swap_Rights;
```

LIST.PKG (cont.)

```
procedure At_Right_End(L : in out List;
                       At_End : in out Boolean) is

begin

  ------------------------------------------------------------
  -- The variable At_End will represent whether the cursor is at
  -- the right end of the list.
  ------------------------------------------------------------

  At_End := (L.Cursor.Next = null);

end At_Right_End;
```

```
  procedure At_Left_End(L : in out List;
                        At_End : in out Boolean) is

  begin

    ------------------------------------------------------------
    -- The variable At_End will represent whether the cursor is at
    -- the front, or left end, of the list.
    ------------------------------------------------------------

    At_End := (L.Front = L.Cursor);

  end At_Left_End;


end List_Template;
```

This page intentionally left blank.

BIBLIOGRAPHY

[Beidler 93] Beidler, J., "An Object-based Approach to CS2/CS7," Computing Sciences Dept., University of Scranton, Scranton, PA, Final Report for ONR Grant MDA972-92-J-1003.

[Bentley 82] Bentley, J.L., *Writing Efficient Programs*, Prentice-Hall, 1982.

[Biggerstaff 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.

[Feldman 92a] Feldman, M. B., and E. B. Koffman, *Ada Problem Solving and Program Design*, Addison-Wesley, 1992.

[Feldman 92b] Feldman, M.B., "Ada Experience in the Undergraduate Curriculum," *Communications of the ACM*, 35(17), November 1992, pp. 53-67.

[Feldman 93] Feldman, M. B., "Ada in a Very Large CS1 Course," In *Seventh Annual Ada Software Engineering Education and Training Symposium*, January 1993.

[Garlan 92] Garlan, D., "Formal Methods for Software Engineers: Tradeoffs in Curriculum Design," In *Proceedings of the Sixth SEI Conference on Software Engineering Education*, 1992, pp. 131-142.

[Gray 93] 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.

[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.

[Hollingsworth 92a] Hollingsworth, J. E. and B. W. Weide, "Engineering 'Unbounded' Reusable Ada Generics," *Proceedings of the Tenth National Conference on Ada Technology*, ANCOST, Inc., Arlington, Virginia, February 1992, pp. 82-97.

[Hollingsworth 92b] 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.

[Koffman 84] Koffman, E. B., "Recommended Curriculum for CS1," *Communications of the ACM*, October 1984, pp. 998-1001.

[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.

[Lutz 92] Lutz, M., "Formal Methods and the Engineering Paradigm," *Proceedings of the Sixth SEI Conference on Software Engineering Education*, 1992, pp. 121-130.

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

[Muralidharan 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.

[Muralidharan 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): 8-24, 1990.

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