Teaching the Second Course in Computer Science in a Reuse-Based Setting: A Sequence of Laboratory Assignments in Ada*

Jeff Gray
jgg@cs.wvu.wvnet.edu

Department of Statistics and Computer Science
West Virginia University
Morgantown, WV 26506

Correspondence: Murali Sitaraman - murali@cs.wvu.wvnet.edu, (304)-293-3607

* This research is funded in part by DARPA Grant # DAAL03-92-G-0412 awarded to West Virginia University and Muskingum College, New Concord, Ohio.

Abstract

We are currently exploring a new approach for introducing key software engineering and computer science principles in the second course of our curriculum. Our approach introduces software reuse as a context for providing a motivation toward learning the importance of principles such as abstraction, specification and design. An essential element for the success of the reuse-based approach is an appropriate series of lab assignments. First, we present the students with components designed and implemented by the lab instructor. Based only on the specifications, students learn to assemble these components and solve interesting problems. Later, students reuse these components to build layered implementations of other components. Only toward the end of the course are they taught how to write their own implementations from scratch, such as using pointers. This paper describes three lab assignments which illustrate our approach.

1 Introduction

Why teach reuse in an introductory course?

Most current undergraduate computer science curricula suffer from two fundamental problems, which often lead to several others. One problem is the absence of a context, and hence motivation, for learning fundamental principles of computer science (e.g. abstraction) in the second course. The other problem is late exposure to principles of software engineering, such as those found in a senior level course, resulting in relatively inexperienced graduates in applying these principles.

We believe these problems can be ameliorated by providing software reuse as a context in which to teach the fundamental principles of computer science. The reuse-based approach also motivates key software engineering principles that are often omitted when the course is taught outside of this context. The principles instilled by the lab assignments discussed in this paper 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;

-- Multiple implementations (with different efficiency characteristics) for a given specification.

Introduction to these principles early in their undergraduate careers will give students ample time to gain confidence in their abilities

by applying the principles throughout their remaining courses. This paper describes an approach toward the construction of laboratory assignments which attempt to meet the above goals. During the past year, such assignments have been used in a section of the second semester freshmen level computer science course at the West Virginia University.

Our definition of reuse

Recent literature on software reuse contains several different definitions or classifications of the term [6]. The definition of reuse used in this paper is one which is component-based [8, 11]. We view a reusable component 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 paper are based only on the specification and performance characteristics (e.g. performance efficiency) of the implementation. We concentrate on components which are *designed* for reuse. We are not concerned with definitions of the term that deal with code scavenging or other methods where the utilization of already existing software occurs by accident or serendipity.

Organization of the paper

The paper is contained in five sections. Sections two through four describe aspects of different laboratory assignments that have been used to introduce students to software engineering and reuse principles. Each of these sections contains the goals, descriptions, and possible variations on the theme of a particular lab. 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. A final section summarizes the paper and offers suggestions for possible areas of future work.

2 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 separation of the specification of a component from its implementation;

-- Acquaint the students with the notation of a specification language;

-- Construction of secondary operations.

Description

The purpose of this laboratory assignment is to solve a backtracking problem iteratively using a stack package provided by the instructor. Several different backtracking problems have been introduced to the students. Examples of problems we have 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;

-- Assisting a squirrel in climbing to the top of a tree, filled with many empty branches, to find an acorn.

All of the above problems share a common trait in that a decision must be made to explore down one of several paths. Also, an ability must be provided so that one can backtrack to previous spots and choose alternative paths when dead-ends are encountered.

The students are asked to solve these problems iteratively using a stack. The specification of an Ada package that provides a stack component is shown in Figure 1. The students are given a copy of this 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 the stack package to solve the backtracking problem.

The client program is then linked with the stack package to obtain an executable.

When the students are given the stack 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 stack 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. To add semantic information to Ada package specifications, we use a close dialect of the RESOLVE specification language [4, 10, 11]. RESOLVE specifications are formal, but yet succinct and understandable by freshmen who have been briefly exposed to topics covered in discrete mathematics.

In Figure 1, the type Stack is modeled as a mathematical string. Manipulations on a variable of type Stack are described using functions borrowed from mathematical string theory (e.g. the concatenation operator = "o", found in the Push and Pop operations). Operations are specified using a requires clause (pre-condition) and an ensures clause (post-condition). These clauses are mathematical assertions and not executable statements. The requires clause states what needs to be true before the operation is called while the ensures clause states what the operation will do provided the requires clause is satisfied at call time. A call to an operation without satisfying the requires clause is undefined and can do anything. Some operations may not have a requires clause. Each reference to a variable in the requires clause refers to the value of the variable at the time the procedure was invoked. In the ensures clause, however, the value of a variable at the time of procedure invocation is accessed by preceding the variable name with a '#' sign. Reference to a variable without the '#' sign refers to its value at the time the operation returns to the caller. Also, since most character sets, including ASCII, do not provide symbols for the universal quantifiers or lambda (i.e. the empty string in our specification), we resort to spelling out the definition of these symbols rather than giving the symbol itself. Aside from an explanation of the concatenation operator, which should be familiar to most readers, the above discussion provides an individual with enough detail to comprehend the meaning of each operation. One can simply view the stack operations as manipulations on a string whereby calls to the Push operation "consume" an item and place it at the end of a string (i.e. S = #S o X); the returned item is assigned an initial value depending on the type of the item. Calls to the Pop operation remove an item from the end of the string (i.e. #S = S o X).

The Basic_Stack_Template has been designed for reuse by following guidelines such as those in [4]. There are several differences between specifications designed using these guidelines and other specifications found in discussions like [1]. Details of these design issues are beyond the scope of this paper. An interested reader is referred to [2, 4, 5, 8, 11] for more detailed descriptions of design issues such as why standard operations called Initialize, Finalize, and Swap are provided for every type.

```
generic

  type T is limited private;

  with procedure T_Initialize(X : in out T);
  --! ensures T.Init(X)

  with procedure T_Finalize(X : in out T);

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

package Basic_Stack_Template is

  type Stack is limited private;
  --! type Stack is modeled by a string of T...

-- standard operations...

  procedure Initialize(S : in out Stack);
  --! ensures S = Lambda

  procedure Finalize(S : in out Stack);

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

-- primary Stack operations...

  procedure Push(S : in out Stack;
                 X : in out T);
```

```
  --! ensures (S = #S o X) and T.Init(X)

  procedure Pop(S : in out Stack;
                X : in out T);
  --! requires S /= Lambda
  --! ensures #S = S o X

  function Is_Empty(S : Stack)
                    return Boolean;
  --! ensures Is_Empty iff S = Lambda

private

  type Representation;
  type Stack is access Representation;

end Basic_Stack_Template;
```

Figure 1
Specification of a Stack Component

A final requirement of the assignment is to construct what are termed secondary operations for the component. Secondary operations provide additional functionality in using a particular component. These operations are often not included in the list of primary operations due to the fact that they can be implemented efficiently without underlying knowledge of the abstract data type representation. To illustrate the difference, the implementation of the primary stack operation called Push must have access to the underlying representation of a stack in order to properly add an element. It needs to know whether the stack is being represented using pointers, arrays, or layered on top of some other component. However, a secondary operation called Copy_Stack, for instance, does not need access to the representation and can be written simply using a loop with proper calls to the primary operations Pop and Push, in addition to the possible need of temporary variables.

The assignment directs the student in assembling two secondary operations for stacks. The operations that the student must write are Reverse_Stack and Print_Stack. These operations are needed in the assignment to print the actual solutions to the problem that the client program discovers. Since the secondary operations require access to various primary operations, the needed primary operations must be passed as generic parameters. An example of how this might be accomplished is found in Figure 2. All the standard (i.e. Initialize, Finalize, and Swap) and primary operations of both the element type T and the Stack type are passed as generic parameters.

```
generic

  -- Semantic specifications for the following
  -- operations are the same as those found in
  -- Figure 1.

  type T is limited private;

  with procedure T_Init(X : in out T);
  with procedure T_Fin(X : in out T);
  with procedure T_Swap(X, Y : in out T);
  with procedure T_Print(X : in out T);

  type Stack is limited private;

  with procedure Initialize(S : in out Stack);
  with procedure Finalize(S : in out Stack);
  with procedure Swap(R, S : in out Stack);
  with procedure Push(S: in out Stack;
                      X: in out T);
  with procedure Pop(S: in out Stack;
                     X: in out T);
  with function  Is_Empty(S: in Stack)
                          return Boolean;

package Secondary_Stack_Ops is

-- secondary operations...

  procedure Reverse_Stack(S : in out Stack);
  --! ensures S = #S^R

  procedure Print_Stack(S : in out Stack);
  --! ensures (S = #S) and (output = S)

end Secondary_Stack_Ops;
```

Figure 2
Specification of a Component for Secondary Stack Operations

Variations

As stated above, there are three variations to the backtracking problem which we have used as laboratory assignments. Similar labs that make use of abstract data types other than a stack could 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.

3 Student as an implementer of a layered component

Goals

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

-- Construction of new components by layering them on top of existing components;

-- Multiple, plug-compatible, implementations (with different efficiency characteristics) for a given specification.

Description

This section describes an assignment that is along the same idea as the last section but offers somewhat of a change in the implementation of the stack package. In this assignment, the students are given the specification to a list component shown in Figure 3. 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 the stack package which they have already seen and used. They must implement the stack operations solely by making calls to the operations of Figure 3 and are not allowed to use any form of pointers or array constructs. Thus, a stack package is implemented by layering it on top of another component. The lab described in the previous section is reused in this case by re-linking it with the new stack implementation. The assignment should assist the students in beginning to think about how multiple implementations for the same specification are constructed (see [9]). 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 preexisting standard components rather than "re-inventing the wheel".

The concept used to represent the list component in Figure 3 is different from the typical list concept presented in textbooks like [1]. In particular, the abstract idea of lists is presented without discussing pointers or access types. A type called list is modeled as two strings of some other type T. 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 Reset and Advance operations are used to traverse through the list. Using the above values of list L, a call to the Reset operation, followed by a call to Remove 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 (i.e. function At_Right_End). The operation Swap_Right will not be used in this assignment. It has been provided for future assignments that may implement secondary operations since it has been found useful in constructing efficient implementations of a Copy_List operation [11].

The students have often found that this assignment can be completed within several hours. Almost all of the required stack operations that they must write can be implemented with merely one line of code. For example, code to implement the Push operation would simply entail making the proper call to a corresponding list operation (i.e. Insert). Similar reasoning follows for the other stack operations

provided the students take care to preserve the FILO ordering of the stack. A student only needs to understand the specification of the list component well enough to discern what calls correspond to similar notions within the stack operations. This reinforces the concept of specification-based component reuse.

```
generic

  type T is limited private;

  with procedure T_Initialize(X : in out T);
  --! ensures T.Init(X)

  with procedure T_Finalize(X : in out T);

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

package Lists is

  type List is limited private;
  -- type List is modeled by a pair of strings of T,
  -- named Left and Right

-- standard operations

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

  procedure Finalize(L: in out List);

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

Figure 3
Specification of a List Component

```
-- primary List operations

  procedure Reset(L: in out List);
  --! ensures (L.Left = Lambda) and
  --!         (L.Right = #L.Left o #L.Right)

  procedure Advance(L: in out List);
  --! requires L.Right /= Lambda
  --! ensures :
  --! (L.Left o L.Right = #L.Left o #L.Right) and
  --!  (thereExists x: T, s.t., L.Left = #L.Left o x)

  function At_Right_End(L: in List)
                              return Boolean;
  --! ensures At_Right_End iff L.Right = Lambda

  procedure Insert(L: in out List;
                   X: in out T);
  --! ensures (L.Left = #L.Left) and
  --! (L.Right = X o #L.Right) and T.Init(X)

  procedure Remove(L: in out List;
                   X: in out Item);
  --! requires L.Right /= Lambda
  --! ensures (L.Left = #L.Left) and
  --! (#L.Right = X o L.Right)

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

private

  type Representation;
  type List is access Representation;

end Lists;
```

Figure 3 (cont.)
Specification of a List Component

Variations

Although the above description layers a stack 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 stack 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. As an example using a FIFO queue to build a stack, if the push operation executes in constant time, then the pop operation must run in linear time due to the need to retrieve the element at the end of the queue since the ordering of the two data structures (i.e. FIFO versus LIFO) differs.

4 Student as an implementer of a reusable component built from-scratch

Goals

In addition to the principles named in sections two and three, this lab introduces the following new concept:

-- Use of access types to efficiently implement components from-scratch.

Description

This section describes variations to a laboratory assignment that is often presented toward the end of a 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 stack 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 stack as in section three, the idea of pointers could be used to implement the stack 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_Right operation, from one list variable to another variable. Correspondingly, the students may be asked to write secondary operations for the list package to provide the facilities for printing and reversing lists.

A proviso could be added to the assignment which states that all primary operations need to be written in constant time. This would be mentioned in conjunction with a statement reminding them that the implementer and client of a component are often different individuals. 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 from scratch if the component does not meet their performance requirements. In this paper we do not go into any details on how the list operations are constructed in constant time but additional information on implementing unbounded reusable components can be found in [3].

5 Conclusions

The structure of most current curricula tends to introduce the fundamental principles of computer science void of any particular context. An introductory course based on a software reuse setting would assist in providing a needed context to introduce these principles. Early exposure to these principles would aid students in applying the ideas toward a vast majority of the programming projects that they would encounter throughout the remainder of their undergraduate careers.

In this paper we presented one approach toward providing a context for teaching the fundamental principles of computer science. With our approach, laboratory assignments are used to inculcate the fundamental principles of computer science whereby software reuse is used as a primary motivator. As examples, a subset of our laboratory assignments currently used at the West Virginia University were described. These assignments first require the student to become a client of reusable components. Later in the semester they are given the opportunity to actually implement their own components at a lower-level (e.g. using pointers).

There is still much work that needs to be done with the implementation of our approach. For example, most of the proposed laboratory assignments that were mentioned under the *Variations* sections need to be constructed. We are also currently working toward conducting a survey to determine the impact of the reuse-based approach as being applied by previous students in other courses in our curriculum.

Acknowledgments

I am indebted to my advisor, Murali Sitaraman, for the help he has offered in completing this paper. His invaluable suggestions were always beneficial whenever I found myself at a crossroad.

Selected References

1. Booch, G., *Software Components with Ada*, Benjamin/Cummings, Menlo Park, CA, 1987.

2. Edwards, S., *An Approach for Constructing Reusable Software Components in Ada*, IDA Paper P-2378, Institute for Defense Analyses, Alexandria, VA, September 1990.

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

4. Hollingsworth, J.E., "Software Componenet Design-for-Reuse: A Language Independent Discipline Applied to Ada", Ph.D. dissertation, Deptartment of Computer and Information Science, The Ohio State University, Columbus, OH, 1992.

5. Harms, D.E. and Weide, B.W., "Copying and Swapping: Influences on the Design of Reusable Software Components", *IEEE Transactions on Software Engineering* 17, 5, May 1991, pp. 424-435.

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

7. 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.

8. 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.

9. 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.

10. Sitaraman, M., Welch, L., and Harms, D.E., "On the Specification of Reusable Software Components", *International Journal of Software Engineering and Knowledge Engineering*, 3, June, 1993, to appear.

11. Weide, B.W., Ogden, W.F, and Zweben, S.H, "Reusable Software Components″, *Advances in Computers*, M.C. Tovits, ed., Academic Press, Vol. 33 (1991), 1-65.

Biography

Jeff Gray can be reached at 1027 Grandview Rd., Glen Dale, WV, 26038. He holds a Bachelor of Science (1991) degree from West Virginia University and is currently pursuing the Master of Science degree.