Two Approaches to Teaching Software Components Using Ada 95

Major David S. Gibson
Department of Computer Science
United States Air Force Academy
David.Gibson@usafa.af.mil

Abstract

Abstract data types (ADTs) have been used for many years to teach the specification and implementation of software components. With the advent of object-oriented programming languages (OOPLs), many ADT specifications and implementations are now encoded in OOPLs as classes. OOPLs typically provide a variety of language mechanisms that support data abstraction and make classes convenient for encoding ADT specifications and implementations. In particular, the inheritance mechanism supported by OOPL classes provides useful ways of relating ADT specifications and implementations. The Ada programming language has always provided solid support for specification and implementation of ADTs with its package construct. The 1995 revision to Ada (Ada 95) added a variety of powerful new language mechanisms such as tagged types (full-fledged inheritance), abstract types, and hierarchical library units. Each of these new Ada language mechanisms, in addition to packages, can be used in various combinations to support the specification, implementation, and extension of ADTs. In this paper, we compare two approaches used to teach to second-year undergraduates specification and implementation of software components in Ada. The first approach is a traditional Ada approach relying solely on the package mechanism to support data abstraction. The second approach relies heavily on Ada's newer object-oriented language mechanisms in addition to hierarchical library units.

1. Background and Motivation

Since its introduction in the early 1980s, the Ada programming language has provided strong support for the development of reusable software components. Ada's package construct supports encapsulation and data abstraction by allowing component developers to separate a component's structural interface from its implementation. A component's structural interface (type names, operation names, and parameter profiles, etc.) may be placed in a package specification file and provided to clients who must understand the interface in order to use the component. A component's operation implementations, on the other hand, may be encapsulated in a package body placed in a separate file hidden from client view. Ada's private types may be used to ensure that component clients do not have direct access to a component's data representation, even when it is placed within the package specification. Furthermore, Ada's generic (parameterized) packages support specification and implementation of generalized components that may be customized for a variety of uses through parameter instantiation. A prominent example of Ada components developed using these language mechanisms is the collection of components described by Grady Booch [Booch87].

In the 1990s, object-oriented design (OOD) and object-oriented programming languages (OOPLs) have come to the forefront and have changed the way that software components are designed and encoded. In OOPLs such as C++ and Java, software components are typically encoded as classes. A class is essentially an abstract data type that supports inheritance and dynamic dispatching of operations. Using inheritance, a new class that is derived from an existing class may inherit some or all of the interface, operation implementations, and data representation of the existing class. Dynamic dispatching of operations (frequently referred to as polymorphism) occurs when selection of the operation implementation to invoke is deferred until run time and based on the class to which an object belongs. Several OOPLs such as C++

and Eiffel (but notably not Java) include parameterized class mechanisms with capabilities similar to those of Ada generic packages.

In 1995 a major new revision of the Ada programming language, Ada 95, was released as an international standard. The most significant changes to Ada were the addition of language mechanisms supporting object-oriented programming. Ada 95's tagged types and supporting new language mechanisms provide full-fledged inheritance and dynamic dispatching of operations similar to that of other OOPLs. Ada 95 also introduced hierarchical library units (child units) into the language. This new language mechanism provides a powerful means of extending existing packages that is largely orthogonal to extensions of tagged types using inheritance. As a result of these new additions to the language, Ada 95 supports a wide variety of approaches to the specification and implementation of software components. Several authors have advocated new, more object-oriented, approaches for the development of software components in Ada 95 (c.f., [Weller 95], [Kempe 95], [Gibson 97A] and [Beidler 97]). However, it is not clear what combination of language mechanisms provides the best approach to teach the development of software components using Ada.

The United States Air Force Academy is an accredited four-year undergraduate institution offering over 30 different majors in basic sciences, engineering, social sciences, and humanities. The Computer Science major requires 50 semester hours of traditional Computer Science, Discrete Mathematics, and Electrical Engineering courses in addition to 94 semester hours of academic core and 15 semester hours of military and physical education courses. Computer Science majors, along with all other cadets, typically take the Computer Science core course, Introduction to Computer Science, during their first year. This course combines a broad survey of computer science topics with an introduction to programming in Ada. In their second year, Computer Science majors usually take the half-semester Programming in Ada course and the full semester Fundamentals of Computer Science Course. The latter course is primarily a data structures course (ACM CS2) but includes substantial material on software design and analysis. It is in this course that we first introduce our Computer Science majors to software component design and implementation using Ada.

During the 1998-99 academic year, we used two very different approaches to teaching software components in the fall and spring offerings of our Fundamentals of Computer Science course. In Section 2 we describe the more conservative of the two approaches. This more traditional approach does not rely on Ada 95's new language mechanisms. In Section 3 we describe an object-oriented approach to components, which relies heavily on many of Ada 95's new language mechanisms. In Section 4 we discuss the advantages and disadvantages of the object-oriented approach as compared to the more traditional approach. Section 5 briefly presents conclusions about which of the two approaches is most appropriate under what circumstances. The six appendices include complete listings of each component discussed in this paper.

2. A Conservative Approach To Teaching Software Components

During the Spring 1999 semester we used a fairly traditional non-object-oriented approach to teaching software component design and implementation in our Fundamentals of Computer Science course. The software engineering principles we sought to convey by introducing component development were: modularity, data abstraction, encapsulation, and the differing roles of specification and implementation. Ada packages provide a very good vehicle for demonstrating and exercising each of these principles. Cadets enter this course with a moderate amount of Ada programming experience and should only need to learn about Ada's packages, private types, and generics in order to encode components using the traditional approach. Most of our Computer Science majors have little trouble learning how to use packages and private types. Teaching cadets how to encode and effectively use generic packages typically proves to be somewhat more challenging, but does not present a major hurdle.

For the first programming assignment in this course, students are given a (non-generic) package specification for a very large natural number component and a test driver client program for this component. They must develop a stack-based data representation and operation implementations that conform to the given component specification. Since the provided stack component is generic, cadets gain experience *using* generic packages from this exercise. For the second programming assignment, cadets are given the package specification of a generic list component (shown in Appendix 1) along with a test driver client program. For this exercise, cadets must develop *two* different implementations that conform to the list specification. One implementation of the list component is a very straightforward stack-based implementation. The other implementation is the cadets' first experience using pointers (Ada's access types).

There are several reasons for requiring students to produce two different list implementations in this exercise. First, this exercise drives home the benefits of abstraction. While both implementations must achieve the same specified functionality, the stack-based list implementation is far easier to understand and encode than the pointer-based list implementation. An obvious lesson learned from this exercise is that the implementation task is far easier when working at a higher level of abstraction as cadets do with the stack-based list implementation. Second, this exercise points out that a single component specification may have different implementations that differ in non-functional characteristics such as performance and maintainability. While the pointer-based list implementation offers slightly better performance for some operations, the stack-based implementation is clearly easier to understand and hence maintain. As we discuss in Section 3, the enhanced support for multiple implementations of a single component specification is one of the attractions of using a more object-oriented approach to components.

The specification package provided to students for this exercise specifies the structural and behavior interface of a generic "one way list" parameterized by the type of item contained within the list. The one way list differs from typical Ada list component specifications in that the state of the list includes a current position within the list. The position may be to the left of any item in the list or to the right of the last item in the list. The chief benefit of this design is that the list interface contains no pointers. Thus implementations of this component do not expose clients to potential aliasing problems, as do more common list component designs.

One notable aspect of the package specification provided to the cadets is that it explicitly provides no data representation for the list. The file and package names end in "NoRep" to indicate that this is a specification-only package. This approach helps students distinguish between the task of component specification (interface design) and component implementation (selection and encoding of data structures and algorithms). Furthermore, this strategy helps cadets understand that there are often alternative ways to represent the state of a component as this exercise demonstrates. When cadets encode implementations for this specification, they must create new package specifications (paired with corresponding package bodies). The new package specifications are renamed copies of the "NoRep" specification with the data representation provided in the private section of the new package specification. Appendices 2 and 3 show, respectively, the package specification and package body for a pointer-based implementation of the specification of One_Way_List_NoRep shown in Appendix 1.

Aside from the convention of having a separate specification-only package, there is nothing out of the ordinary in the approach to encoding component specifications and implementations described here and shown in Appendices 1-3. The chief advantage of using this approach to teach component development in Ada is its simplicity in terms of conservative use of Ada 95's many language mechanisms. We discuss advantages and disadvantages of using this approach in Section 4.

3. An Object-Oriented Approach To Teaching Software Components

During the Fall 1998 semester we used an object-oriented approach to teaching software component design and implementation in our Fundamentals of Computer Science course. This approach was ambitious in that it introduced cadets to several advanced Ada 95 language mechanisms very early in the course. Many of the ideas we used for encoding components in Ada 95 had their origin in the author's graduate work at The Ohio State University [Gibson97A, Gibson97B]. As with the traditional approach discussed in the last section, our goals using this approach included teaching modularity, data abstraction, encapsulation, and the differing roles of specification and implementation. However, an additional goal was to introduce students to OOP concepts early in the course in preparation for the fourth and final programming assignment, which focused on OOD and OOP concepts.

In addition to using Ada's packages and private types, the object-oriented component encoding incorporates tagged types, abstract (tagged) types, abstract operations, null record extensions, child units, and generic child units – all new language mechanisms added with the 1995 revision of Ada. Several of the more advanced cadets also were exposed to Ada 95's controlled (tagged) types as part of an optional extra credit assignment. In this section we discuss the way in which these language mechanisms are used. The details of this section may be difficult to follow for readers not very familiar with these Ada language mechanisms. We address the ramifications of this complexity in Section 4.

Ada 95's abstract types and abstract operations are useful for encoding "fully abstract" component specifications – specifications that are committed to no operation implementations and no data representations. In the more traditional approach, we used a NoRep package specification that included no data representation and was not associated with any package body. A NoRep package specification used in this way can be compiled for syntax checking of the specification code that it contains. However, only renamed implementation-specific copies of NoRep packages can be used directly for interface conformance checking. One advantage of using abstract types and abstract operations is that they can be used to create a package specification that is independent of any package bodies and yet may be used for interface conformance checking with client code and implementation packages. A package specification defining an abstract type and only abstract operations defines what is often called an "abstract base class" or, in Java, an interface. We refer to such a package with an abstract type (with no data representation) and all abstract operations as an *abstract component*. More specifically, we call a non-generic abstract component an *abstract instance* and we call a generic abstract component an *abstract template*.

For the first programming exercise during the Fall semester, we provided the cadets with the abstract instance (a non-generic interface specification) for large natural numbers. For the second programming assignment we provided the cadets with the abstract template (a generic interface specification) for a one way list. This package, named AT_One_Way_List, is shown in Appendix 4. The prefix "AT_" stands for *abstract template*. This package corresponds to the One_Way_List_NoRep package discussed in Section 2 and shown in Appendix 1.

With the traditional approach, the exported type List in One_Way_List_NoRep is declared simply as a limited private type:

```
type List is limited private;
```

With the object-oriented approach, the exported list type in AT_One_Way_List has the following declaration:

```
type List is abstract tagged limited null record;
```

The type List must be declared as abstract in this package since all of its primitive operations (methods) are declared as abstract operations. The type is declared as tagged (as are all abstract types) so that new types may be derived from it (inheriting its specification) and extended with concrete operations and a data representation. The "null record" component of this declaration indicates that no data representation is provided for this abstract List type. The type need not be declared as private since there is no data representation to hide from client code. Since variables may not be declared with an abstract type, an abstract type such as this serves only to define an interface from which concrete (non-abstract) types may be derived. In practice, this abstract interface specification may be used for structural conformance checking of implementations and of client code using implementations of this specification.

With the addition of *concrete components*, components that provided a complete data representation and implementations for all operations, our component encoding strategy gets a bit more complex. Not only do we derive a concrete component's type from the abstract type providing its specification, but we also make the package defining the concrete type a child unit of the package defining its abstract parent type. The package for AT_One_Way_List.CT_Stack (shown in Appendices 5 and 6) illustrates this relationship. The name of this package indicates that it is a child unit of the package AT_One_Way_List. The parent unit package name is always the prefix (followed by a period) of a child unit package name. The "CT_" prefix for the child unit suffix of the package name stands for *concrete template*.

The package specification for AT_One_Way_List.CT_Stack does three things. First, it declares the new concrete type List (more specifically, AT_One_Way_List.CT_Stack.List) that is derived from the abstract type List (AT_One_Way_List.List) in the declaration:

```
type List is new AT_One_Way_List.List with private;
```

The "with private" component of this declaration indicates that a private section with a possible extension to the List representation follows. Second, this package declares concrete operations, each of which conforms to an operation specified in the package of its abstract parent (AT_One_Way_List). These declarations are identical to those in the abstract parent except that they lack the "is abstract" part. Third, the package provides a private data representation for the List type. In this case, the List is represented with a pair of stacks that are defined by the following unusual pair of instantiations.

```
package AI_Item_Stack is new AT_Stack (Item => Item);

package CI_Item_Stack is new AI_Item_Stack.CT_Nodes;
```

This pair of generic package instantiations demonstrates how client code may obtain an instance of a concrete type (type Stack in this case) specified by an abstract template and implemented by a concrete template. In this example, the concrete one way list package is a client of the stack component. These instantiations also demonstrate the most complex aspect of this approach to encoding and components. The first instantiation above (of AT_Stack) creates the abstract type AI_Item_Stack.Stack. The actual parameter Item is whatever type is used as the actual parameter to an instantiation of AT_One_Way_List. Since AT_One_Way_List.CT_Stack is a child unit of AT_One_Way_List, it has direct visibility to the formal parameter type Item defined in its parent. This is the primary motivation for making an implementation package a child unit of its specification package. The second instantiation shown above is the instantiation of the generic child unit AT_Stack.CT_Nodes. This package provides a generic pointer-based stack implementation in a generic child unit. The syntax for this instantiation is odd in that it appears that a package named AI_Item_Stack.CT_Nodes is being instantiated. Furthermore, there does not appear to be any generic parameter in this case since the child unit itself does not add any parameters to the single parameter of its parent unit AT_Stack. Nevertheless, since its parent AT_Stack is generic, AT_Stack.CT_Nodes must also be a generic package that must be instantiated to generate a usable concrete instance, CI_Item_Stack in this case. Client code using an implementation of AT_One_Way_List would include a pair of instantiations similar to this to create a concrete instance providing a usable List type.

The package body of AT_One_Way_List.CT_Stack, shown in Appendix 6, completes the stack-based implementation of a one way list. This package body provides implementations of the operations specified by AT_One_Way_List. AT_One_Way_List.CT_Stack and One_Way_List_Nodes (shown in Appendices 2 and 3) are two very different implementations of the same list specification. In the second programming exercise, cadets develop both of these List implementations (using one of the two component approaches). While cadets find the stack-based implementation much easier to code and debug, they also learn that the pointer-based implementation is slightly more efficient. In particular, the Move_To_Start and Move_To_Finish operations require linear time in the stack-based implementation whereas they only require constant time in the pointer-based implementation.

4. Comparison of the Two Approaches

In this section we discuss the relative pros and cons of the two approaches to teaching software component development with Ada 95. We frame this discussion in terms of the advantages and disadvantages of the object-oriented approach discussed in Section 3 as compared to the more conservative approach discussed in Section 2.

4.1 Advantages of the Object-Oriented Approach

The primary advantages of the object-oriented approach that we used during the Fall 98 semester are:

1. better separation of specifications from implementations
2. better support for multiple implementations of the same component specification
3. early introduction of object-oriented concepts and programming techniques, and
4. early introduction of component extension using child units.

The first advantage listed above is primarily a pedagogical issue. Prior to Ada 95, package specifications exporting a private type had to have the data representation of the type included in the package specification rather than the package body. The disadvantage of this requirement is that component specifications include specific implementation information – a concrete data representation. Although this requirement makes compilation easier, it blurs the important distinction between component specification (interface design) and component implementation (selection and encoding of data representations and algorithms). It is possible to hide representation details in the package body using access types. However, defining an abstract type with a null record representation provides a much cleaner approach. This technique, as illustrated by the package AT_One_Way_List in Appendix 4, enables an abstract component to be encoded with no commitment to potential data representations.

The second advantage listed above closely relates to the first advantage. Prior to Ada 95, each package specification could be associated with at most one (identically named) package body. The only way to have a single package specification serve as the interface for multiple implementation packages was to use various configuration management tricks that allowed differing package bodies to have the same name. The use of abstract tagged types, makes it possible to decouple an abstract component package specification from an implementation package. While this separation is achieved using the NoRep package specifications in the traditional approach, the traditional approach provides no compiler-enforced conformance checking between the NoRep specification and implementation packages. Deriving the concrete implementation type from the abstract specification type, as discussed in Section 3, does ensure compile-time checking of structural conformance between a single abstract type and any number of derived concrete types. This form of abstract-to-concrete type conformance is sometimes called *specification inheritance*.

In current practice, components typically are designed with a single implementation in mind. However, having cadets develop multiple implementations that conform to a single specification has several benefits. First, it helps cadets better distinguish between component specification and implementation since a specification is not tightly coupled to a single implementation. Cadets generally think about software at a very concrete level at this point in their education. Having cadets work with component specifications that are disconnected from specific implementations encourages them to think at a higher level of abstraction. Second, multiple implementations encourage more generalized design of component interfaces since commitments to specific implementations need not influence the design process. Finally, multiple implementations encourage cadets to consider different performance and other non-functional tradeoffs that must be made when selecting data representations and algorithms. The second programming exercise in our Fundamentals of Computer Science is designed to bring all of these issues to the forefront.

Introducing cadets to OO concepts and programming techniques early in the Computer Science curriculum is the most practical benefit of using the OO approach to components. We believe that computer science majors should have a solid understanding of OO concepts by the time they graduate. In a course such as our Fundamentals of Computer Science, however, it is difficult to devote a significant amount of time to OO analysis, design, and programming since there are so many other important topics to cover. Despite the advanced nature of some of the language mechanisms involved, using this style of components provides cadets with a relatively gentle introduction to OOP in Ada. The OO components make relatively simple use of inheritance. Initially, only specification inheritance is used, ensuring that concrete types conform to their abstract specifications. While we discuss code inheritance as a way of extending concrete components, cadets do not use code inheritance until their final assignment. Furthermore, these OO components are not designed for dynamic dispatch (as are the "polymorphic components" described in John Beidler's *Data Structures and Algorithms* text [Beidler 97]). Components that support dynamic dispatching require the use of additional advanced language mechanisms such as class-wide types and pointers to class-wide types. Nevertheless, introducing Ada's OOP language mechanisms with the first packaged software components that cadets see allows us to move on to more advanced OOP concepts by end of this course.

The fourth and final programming assignment in our Fundamentals of Computer Science course has students develop a graphical game with an object-oriented design and implementation. In this exercise, cadets use code inheritance (concrete type extension) to extend provided classes. This assignment also has students work with an event queue that uses dynamic dispatching to handle events for different classes of objects. Having worked with Ada's OOP language mechanisms since the beginning of the class, cadets are able to handle the new OO concepts introduced in this programming assignment relatively easily. Furthermore, introducing cadets to OOP in this course helps students gain a deeper understanding of OO concepts when they are taught OOP in upper-level courses such as our Programming Languages course.

Finally, we believe that introducing cadets to Ada's hierarchical library units early in the Computer Science major is also beneficial. Child units provide a very powerful mechanism for extending existing components whether or not they are used in conjunction with object-oriented mechanisms. Child units also are useful for teaching incremental development of large systems. Since non-generic child units are conceptually simple, cadets tend to have little trouble learning how to use them. As we discussed in Section 3, generic child units are an advanced language mechanism and clearly are more difficult for students to understand. Nevertheless, generic child units serve an important role in this style of OO components. Since a generic child unit has visibility to the formal generic parameters of its parent, a generic child unit serving as a concrete template may refer directly to its parent unit's formal parameters. For example, the instantiation of AT_Stack in AT_One_Way_List.CT_Stack (see Appendix 5) uses Item, the generic formal parameter of AT_One_Way_List. Without using a generic child unit in this way, it would be much more complicated to relate generic specifications and implementations using inheritance.

4.2 Problems with the Object-Oriented Approach

The primary disadvantages of the object-oriented approach as compared to the more traditional approach are:

1. students are exposed to many advanced language mechanisms very early
2. there is currently no text book that adequately complements this style of components
3. instructors must be very knowledgeable about advanced Ada 95 language mechanisms, and
4. the complex interaction of language mechanisms can lead to cryptic compiler messages.

The most obvious drawback of the OO component approach is that cadets get exposed to several advanced Ada 95 language mechanisms very early in this course. Even with prior Ada experience, the sight of syntax like "**`type List is abstract tagged limited null record;`**" can be somewhat intimidating. While instructors explained each of the language mechanisms used as they were introduced, relatively few students were able to fully understand the code they were seeing near the beginning of the course. This required instructors to tell some students, "Don't worry about understanding all of it now, you'll understand it later." For some students, this approach is frustrating. Furthermore, some of the weaker students in the course never learned the concepts involved.

To gain a better understanding of the students' perspective on this matter, we asked all of the students to answer the following two questions on anonymous end-of-course critique forms.

> Advanced Ada language mechanisms such as *abstract tagged types*, *abstract operations*, *child units*, and *generic child units* were introduced early in CS225.
>
> Question 1: When these concepts were first introduced, how difficult was it for you to learn them?

Question 2: Now that you have used these language mechanisms on several assignments, how well do you understand them?

The following tables summarize responses to these two questions from the 43 students completing the Fall 1998 offering of Fundamentals of Computer Science. This data does not include the responses from several weaker students who dropped out of the course before completing the course.

Question 1 - Initial Learning	
Response	**Frequency**
Very difficult	28%
Somewhat difficult	47%
Not too difficult	12%
Easy	12%
Very easy	2%

Question 2 - Final Understanding	
Response	**Frequency**
Do not understand at them at all	0%
Understand them a little	12%
Understand well enough to use them	23%
Understand them pretty well	37%
Understand them very well	28%

The data shown above confirms that the majority of cadets in the course initially had difficulty understanding the Ada programming mechanisms used by the OO component approach. Over half of the cadets reported that they initially found it “very difficult” or “somewhat difficult” to understand the advanced Ada language mechanisms used by the components. By the end of the course, however, cadets had completed four programming exercises using Ada's OOP mechanisms. The responses to the second question indicate that at the end of the course the majority of cadets felt that they understood Ada's OOP mechanisms “pretty well” or “very well”.

There are no course textbooks that use the OO component style presented here. Clearly, having a textbook that presents Ada components developed in this style would be beneficial to the students. We use Michael Feldman's *Software Construction and Data Structures with Ada 95* [Feldman 96] as the course text for the Fundamentals of Computer Science course. The components presented in this text (and provided in electronic form) do not use any of Ada's OOP mechanisms. During the Fall 98 and Spring 99 semesters, we did not make significant use of Feldman's components as had been done in past offerings of this course. Instead, students were provided with several examples of components developed in the styles described in Sections 2 and 3. During the Fall 98 semester when we used the OO approach, we provided students with a 4-page handout describing the philosophy, syntax, and usage of the OO-style components Unfortunately, we did not provide this handout to the cadets at the very beginning of the semester when it would have been most useful and might have positively influenced the responses to the first question above.

One of the more practical problems with teaching the OO component style is that it requires instructors who are very experienced with Ada 95 and are committed to the approach. An instructor not experienced with Ada 95's OOP mechanisms and child units is likely to have some difficulty explaining and using this style of components. In fact, even instructors very experienced in Ada may never have used constructs such as generic child packages. Furthermore, since initially the OO style components are more difficult for students to understand, instructors must be willing to work through greater challenges early in the course before seeing the potential payoffs of using this approach.

One other minor problem that we encountered using the OO component style dealt with compiler error messages. On several occasions during the fall semester, cadets made errors in their component code that produced error messages that were very difficult for the cadets to understand. At least several of these problems involved incorrect instantiation of generic child units. The error messages reported by the GNAT Ada 95 compiler accurately described the problems in the code. However, understanding the problem descriptions required a deeper understanding of the language mechanisms involved than the cadets possessed. In several cases, errors of this nature stumped even the brightest students in the class and required help from the instructor.

5. Conclusions

In this paper we have presented two approaches to teaching software component development in Ada 95. Both of these approaches are reasonable strategies for teaching undergraduates and each approach has advantages and disadvantages relative to the other. The best approach to use depends on student experience, course goals, and instructor knowledge and commitment. Based on our experience, we believe that teaching with the more traditional (non-object oriented approach) described in Section 2 is more attractive if:

- students are not yet comfortable with more basic Ada language mechanisms,
- teaching OOP early in the curriculum is not a priority,
- instructors are not well-versed in advanced Ada 95 language mechanisms, or
- instructors are not committed to OOP or this approach to encoding components.

Teaching with the object-oriented approach described in Section 3 is more attractive if:

- students have mastered basic Ada programming,
- one of the course goals is to introduce students to OOP,
- instructors understand the advanced language mechanisms used by this approach, and
- instructors are committed to teaching Ada, OOP, and software component design.

Teaching second-year undergraduates component development in Ada using the object-oriented approach is challenging due to the complexity of the language mechanisms involved. Nevertheless, we believe that the benefits of this approach warrant its serious consideration.

Bibliography

[Beidler 97] Beidler. *Data Structures and Algorithms: An Object-Oriented Approach Using Ada 95*. Springer-Verlag, New York, 1997.

[Booch 87] Grady Booch. *Software Components with Ada: Structures, Tools, and Subsystems*. Benjamin/Cummings, Menlo Park, CA, 1987.

[Feldman 96] Michael B. Feldman. *Software Construction and Data Structures with Ada 95*. Addison-Wesley Publishing Company, Reading, MA, 1996.

[Gibson 97A] David S. Gibson. *An Introduction to RESOLVE/Ada95*. Technical Report OSU-CISRC-4/97- TR23, The Ohio State University, Columbus, Ohio, 1997.

[Gibson 97B] David S. Gibson. *Behavioral Relationships Between Software Components*. PhD thesis, Department of Computer and Information Science, The Ohio State University, Columbus, Ohio, 1997.

[Kempe 95] Magnus Kempe. The composition of abstractions: Evolution of software component design with Ada 95. In *TRI-Ada'95 Conference Proceedings*, pages 394-405, New York, NY, 1995. ACM.

[Weller 95] David Weller. The Ada 95 Booch components. In *TRI-Ada'95 Conference Tutorial Proceedings*, pages 175-223, New York, NY, 1995. ACM.

Appendix 1 – One_Way_List_NoRep Package Specification

```
-- File: One_Way_List_NoRep
-------------------------------------------------------------------------
--
-- One_Way_List Specification - no data representation
--
-- Written by Major Gibson for Comp Sci 225, Spring 1999
--
------------------------------------------------------------------------
--
-- This package defines a generic (unbounded) list ADT.  The list is a
-- "one way list" since it may only be traversed in one (the forward)
-- direction.  A one way list value is a sequence of values of type Item
-- (where Item is a generic type parameter to this package) and a
-- position within that sequence.  An Item may be added to the list just
-- right of the current position (Add_Right).  As long as the current
-- position is not the end (rightmost) position of the list, the item
-- to the right of the current position may be removed (Remove_Right).
-- The current position may be advanced one position to the right
-- (Advance), changed to the beginning of the list (Move_To_Start), or
-- changed to the end of the list (Move_To_Finish).  The initial value
-- of a one way List is an empty list.
--
------------------------------------------------------------------------

generic

   ---------------------------------------------------------------------
   -- Generic Package Parameters

   type Item is private;   -- type of values held by the List

   ---------------------------------------------------------------------

package One_Way_List_NoRep is

   ---------------------------------------------------------------------
   -- Exported Type

   type List is limited private;

   -- type List is modeled by a pair of Strings (Left, Right) of values
   --                          of type Item
   --    exemplar l
   --    initialization ensures l = (<>, <>)

   --    A one way list List value is represented by two strings, Left
   --    and Right.  The current position within the list corresponds to
   --    the location between Left and Right.  The front of the list
   --    corresponds to the leftmost item in Left.  The end of the list
   --    corresponds to the rightmost item in Right.  Advancing one
   --    position in the list corresponds to moving the leftmost item
   --    of Right to the rightmost item in Left.  The initial value of a
   --    one way list is represented by a pair of empty strings.

   ---------------------------------------------------------------------
   -- Exported Exceptions

   Remove_At_End  : exception;  -- raised by Remove_Right
   Advance_At_End : exception;  -- raised by Advance;
```

```
----------------------------------------------------------------------
-- Explicit Exported Operations

procedure Move_To_Start (
      L : in out List
   );

-- ensures
--    L' = (<>, L.Left * L.Right)

--    The list value returned (L') is the same as the list value
--    provided (L) except that the position is to the left of the
--    leftmost (first) item in L.  For example, if L is initially
--    < 1 2 (position) 3 4 > (represented by <1 2>, <3 4>), then
--    after calling Move_To_Start(L), the new value of L (L') is:
--    <(position) 1 2 3 4 > (represented by <>, <1 2 3 4>).

---------------------------------------------------------------------

procedure Move_To_Finish (
      L : in out List
   );

-- ensures
--    L' = (L.Left * L.Right, <>)

--    The list value returned (L') is the same as the list value
--    provided (L) except that the position is to the right of the
--    rightmost (last) item in L.  For example, if L is initially
--    < 1 2 (position) 3 4 > (represented by <1 2>, <3 4>), then
--    after calling Move_To_Finish(L), the new value of L (L') is:
--    <1 2 3 4 (position)> (represented by <1 2 3 4>, <>).

---------------------------------------------------------------------

procedure Advance (
      L : in out List
   );

-- requires
--    L.Right \= <>
-- ensures
--    L' = there is an Item X such that
--          (L'.Left = L.Left * <X> and L.Right = <X> * L'.Right)
-- raises
--    Advance_At_End

--    If the position of L is at the end of the list when Advance is
--    called, then the exception Advance_At_Finish gets raised.
--    Otherwise, current position is moved one place to the right in
--    the list.  For example, if L = <1 2 (position) 3 4 >
--    (represented by (<1 2>, <3 4>)) when Advances is called, then
--    the value of L returned (L') would be <1 2 3 (position) 4>
--    (represented by (<1 2 3>, <4>)).  In this example, X in the
--    ensures clause is 3.


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

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

-- ensures
--    L' = (L.Left, <X> * L.Right)

--    Add_Right adds the value of item X to the right of the current
--    position in list L.  For example, if L = <1 2 (position) 3 4
--    (represented by (<1 2>, <3 4>)) and X = 5 when Add_Right is
--    called, then the value of L returned (L') would be
--    <1 2 (position) 5 3 4> (represented by (<1 2>, <5 3 4>)).

----------------------------------------------------------------------

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

-- requires
--    L.Right \= <>
-- ensures
--    (L'.Left = L.Left) and (L.Right = <X'> * L'.Right)
-- raises
--    Remove_At_End

--    Remove_Right removes the item right of the current position
--    from list L and sets X to this value.  For example, if
--    L = <1 2 (position) 3 4 (represented by (<1 2>, <3 4>)) when
--    Remove_Right is called, then the value of L returned (L')would
--    be <1 2 (position) 4> (represented by (<1 2>, <4>)) and the
--    value of X (X') would be set to 3.

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

```
   function Right_Size (
         L : in List
      ) return Natural;

   -- ensures
   --    result =  |L.Right|

   --    The function returns the total number of items to the right of
   --    the current position in list L.

   ----------------------------------------------------------------------
   function Left_Size (
         L : in List
      ) return Natural;

   -- ensures
   --    result =  |L.Size|

   --    The function returns the total number of items to the left of
   --    the current position in list L.

   ----------------------------------------------------------------------

   procedure Clear (
         L : in out List
      );

   -- ensures
   --    L' = (<>, <>)

   --    The Clear procedure removes all items (if any) from list L
   --    returning L (L') as an empty list (represented by a pair of
   --    empty strings (<>, <>).

   ----------------------------------------------------------------------

private

   -----------------------------------------------------------------------
   -- Data Representation

   type List is record
      null;                        -- data representation not provided
   end record;

   ----------------------------------------------------------------------

end One_Way_List_NoRep;
```

Appendix 2 – One_Way_List_Nodes Package Specification

```ada
-- File: One_Way_List_Nodes.ads
------------------------------------------------------------------------
--
-- One_Way_List Specification - using linked nodes data representation
--
-- Written by Major Gibson for Comp Sci 225, Spring 1999
--
------------------------------------------------------------------------
--
-- This package defines a generic (unbounded) list ADT.  The list is a
-- "one way list" since it may only be traversed in one (the forward)
-- direction.  A one way list value is a sequence of values of type Item
-- (where Item is a generic type parameter to this package) and a
-- position within that sequence.  An Item may be added to the list just
-- right of the current position (Add_Right).  As long as the current
-- position is not the end (rightmost) position of the list, the item
-- to the right of the current position may be removed (Remove_Right).
-- The current position may be advanced one position to the right
-- (Advance), changed to the beginning of the list (Move_To_Start), or
-- changed to the end of the list (Move_To_Finish).  The initial value
-- of a one way List is an empty list.
--
------------------------------------------------------------------------

generic

   ----------------------------------------------------------------------
   -- Generic Package Parameters

   type Item is private;   -- type of values held by the List

   ----------------------------------------------------------------------

package One_Way_List_Nodes is

   ----------------------------------------------------------------------
   -- Exported Type

   type List is limited private;

   ----------------------------------------------------------------------
   -- Exported Exceptions

   Remove_At_End  : exception;  -- raised by Remove_Right
   Advance_At_End : exception;  -- raised by Advance;

   ----------------------------------------------------------------------
   -- Explicit Exported Operations

   procedure Move_To_Start (
         L : in out List
      );

   procedure Move_To_Finish (
         L : in out List
      );

   procedure Advance (
         L : in out List
      );

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

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

   function Right_Size (
         L : in List
      ) return Natural;

   function Left_Size (
         L : in List
      ) return Natural;

   procedure Clear (
         L : in out List
      );

   ----------------------------------------------------------------------
   -- Representation

private

   type Node_Rep;                     -- representation of place in List

   type Node_Ptr is access Node_Rep;  -- pointer to rep. of List place

   type Node_Rep is                   -- complete definition of Node_Rep
      record
         Data : Item;          -- value of an item in the List
         Next : Node_Ptr;      -- pointer to next place toward list end
      end record;

   type List is
      record
          First : Node_Ptr;        -- pointer to first place in list
          Place : Node_Ptr;        -- pointer to node left of position
           Last : Node_Ptr;        -- pointer to last node in list
          LSize : Natural := 0;    -- # of item to left of position
          RSize : Natural := 0;    -- # of item to right of position
      end record;

   -- This pointer-based representation holds the List items in a
   -- linked list of nodes (of type Node_Rep).

   ----------------------------------------------------------------------

end One_Way_List_Nodes;
```

Appendix 3 – One_Way_List_Nodes Package Body

```
-- File: One_Way_List_Nodes.adb
------------------------------------------------------------------------
--
-- One_Way_List Implementation - using linked nodes data representation
--
-- Written by Major Gibson for Comp Sci 225, Spring 1999
--
------------------------------------------------------------------------
--
-- This package defines a generic (unbounded) list ADT.  The list is a
-- "one way list" since it may only be traversed in one (the forward)
-- direction.  A one way list value is a sequence of values of type Item
-- (where Item is a generic type parameter to this package) and a
-- position within that sequence.  An Item may be added to the list just
-- right of the current position (Add_Right).  As long as the current
-- position is not the end (rightmost) position of the list, the item
-- to the right of the current position may be removed (Remove_Right).
-- The current position may be advanced one position to the right
-- (Advance), changed to the beginning of the list (Move_To_Start), or
-- changed to the end of the list (Move_To_Finish).  The initial value
-- of a one way List is an empty list.
--
------------------------------------------------------------------------
-- Context

with Unchecked_Deallocation;    -- generic memory deallocation procedure

------------------------------------------------------------------------

package body One_Way_List_Nodes is

   ---------------------------------------------------------------------
   -- Local operations

   procedure Free is new        -- This procedure is used to "free" or
      Unchecked_Deallocation (  -- deallocate the memory used by a
         Object => Node_Rep,    -- Node_Rep.  The memory returned to the
           Name => Node_Ptr     -- OS is pointed to by Free's parameter.
      );

   ---------------------------------------------------------------------
   -- Explicit Exported Operations

   procedure Add_Right (
         L : in out List;
         X : in Item
      ) is
         Ptr : Node_Ptr;          -- pointer to nodes to be added
   begin

      Ptr := new Node_Rep;        -- create new node to insert
      Ptr.Data := X;              -- set value of new node to X
      L.RSize := L.RSize + 1;     -- add 1 to size of list

      if L.First = null then      -- add first node to list
         L.First := Ptr;             -- point first to single node
         L.Last  := Ptr;             -- point last to single node

      elsif L.LSize = 0 then       -- at beginning of non-empty list
```

```
      Ptr.Next := L.First;        -- next after new node is first
      L.First  := Ptr;            -- first node is new node
   else                         -- in or at end of non-empty list
      Ptr.Next := L.Place.Next;   -- new node next is right of pos
      L.Place.Next := Ptr;        -- next right of Place is new node
      if Ptr.Next = null then     -- if adding to end of list
         L.Last := Ptr;              -- point Last to new node
      end if;
   end if;

end Add_Right;

---------------------------------------------------------------------

procedure Remove_Right (
      L : in out List;
      X : out Item
   ) is
      Ptr : Node_Ptr;           -- pointer to nodes to be removed
begin

   if L.RSize = 0 then          -- if at end of list
        raise Remove_At_End;      -- then raise exception
   end if;

   if L.LSize = 0 then          -- if at beginning of non-empty list
      Ptr := L.First;              -- point Ptr to first node
      L.First := Ptr.Next;         -- First points to node after 1st
      if L.RSize = 1 then          -- if removing last node
         L.Place := null;             -- set Place and Last to null
         L.Last  := null;
      end if;
   else
      Ptr := L.Place.Next;      -- point Ptr to node after position
      L.Place.Next := Ptr.Next; -- new next is after node to remove
      if L.RSize = 1 then       -- if removing from end of list
         L.Last := L.Place;     -- point Last current place in list
      end if;
   end if;

   X := Ptr.Data;               -- set X to value in node to go
   Free(Ptr);                   -- free memory of removed node
   L.RSize := L.RSize - 1;      -- subtract 1 from size of list

end Remove_Right;

---------------------------------------------------------------------

procedure Move_To_Start (
      L : in out List
   ) is
begin

   L.Place := null;              -- set Place to null
   L.RSize := L.RSize + L.LSize; -- set RSize to # of items in list
   L.LSize := 0;                 -- set LSize to 0

end Move_To_Start;
```

Appendix 3 – One_Way_List_Nodes Package Body (Continued)

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

procedure Move_To_Finish (
      L : in out List
   ) is

begin

   L.Place := L.Last;            -- set Place to last node or null
   L.LSize := L.LSize + L.RSize; -- set LSize to # of items in list
   L.RSize := 0;                 -- set RSize to 0


end Move_To_Finish;

---------------------------------------------------------------------

procedure Advance (
      L : in out List
   ) is

begin

   if L.RSize = 0 then           -- if at end of list
      raise Advance_At_End;         -- raise exception
   end if;

   if L.LSize = 0 then           -- if at beginning of list
      L.Place := L.First;           -- move position to after First
   else
      L.Place := L.Place.Next;      -- move forward one node in list
   end if;

   L.RSize := L.RSize - 1;       -- subtract 1 from size of right
   L.LSize := L.LSize + 1;       -- add 1 to size of left list

end Advance;

---------------------------------------------------------------------

function Right_Size (
      L : in List
   ) return Natural is

begin

   return(L.RSize);      -- return number of items right of position

end Right_Size;
```

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

   function Left_Size (
         L : in List
      ) return Natural is

   begin

      return(L.LSize);      -- return number of items left of position

   end Left_Size;

   ---------------------------------------------------------------------

   procedure Clear (
         L : in out List
      ) is
   begin

      for i in 1 .. L.LSize + L.RSize loop -- for each item in the list
         L.Place := L.First;               -- point Place to first node
         L.First := L.Place.Next;          -- point First to next node
         Free(L.Place);                    -- free memory of first node
      end loop;
      L.Last := null;                      -- set Last to null
      L.RSize := 0;                        -- set left size to 0
      L.LSize := 0;                        -- set right size to 0

   end Clear;

   ---------------------------------------------------------------------

end One_Way_List_Nodes;
```

Appendix 4 – AT_One_Way_List Package Specification

```
-- File: at_one_way_list.ads
-----------------------------------------------------------------------
--
-- One_Way_List Specification
--
-- Written by Major Gibson for Comp Sci 225, Fall 1998
--
-----------------------------------------------------------------------
--
-- This package defines a generic (unbounded) list ADT.  The list is a
-- "one way" list since it may only be traversed in one (the forward)
-- direction.  A one way List value is a sequence of values of type Item
-- (where Item is a generic type parameter to this package) and a
-- position within that sequence.  An Item may be added to the list just
-- right of the current position (Add_Right).  As long as the current
-- position is not the end (rightmost) position of the list, the item
-- to the right of the current position may be removed (Remove_Right).
-- The current position may be advanced one position to the right
-- (Advance), changed to the beginning of the list (Move_To_Start), or
-- changed to the end of the list (Move_To_Finish).  The initial value
-- of a one way List is an empty list.
--
-----------------------------------------------------------------------

generic

   --------------------------------------------------------------------
   -- Generic Package Parameters

   type Item is private;   -- type of values held by the List

   --------------------------------------------------------------------

package AT_One_Way_List is

   --------------------------------------------------------------------
   -- Exported Type

   type List is abstract tagged limited null record;

   -- type List is modeled by a pair of Strings (Left, Right) of values
   --                         of type Item
   --    exemplar l
   --    initialization ensures l = (<>, <>)

   --    A one way list value is represented by two strings, Left
   --    and Right.  The current position within the list corresponds to
   --    the location between Left and Right.  The front of the list
   --    corresponds to the leftmost item in Left.  The end of the list
   --    corresponds to the rightmost item in Right.  Advancing one
   --    position in the list corresponds to moving the leftmost item
   --    of Right to the rightmost item in Left.  The initial value of a
   --    one way list is represented by a pair of empty strings.

   --------------------------------------------------------------------
   -- Exported Exceptions

   Remove_At_End  : exception;  -- raised by Remove_Right
   Advance_At_End : exception;  -- raised by Advance;

   --------------------------------------------------------------------
   -- Explicit Exported Operations

   procedure Move_To_Start (
         L : in out List
      ) is abstract;

   -- ensures
   --    L' = (<>, L.Left * L.Right)

   --    The list value returned (L') is the same as the list value
   --    provided (L) except that the position is to the left of the
   --    leftmost (first) item in L.  For example, if L is initially
   --    < 1 2 (position) 3 4 > (represented by <1 2>, <3 4>), then
   --    after calling Move_To_Start(L), the new value of L (L') is:
   --    <(position) 1 2 3 4 > (represented by <>, <1 2 3 4>).

   --------------------------------------------------------------------

   procedure Move_To_Finish (
         L : in out List
      ) is abstract;

   -- ensures
   --    L' = (L.Left * L.Right, <>)

   --    The list value returned (L') is the same as the list value
   --    provided (L) except that the position is to the right of the
   --    rightmost (last) item in L.  For example, if L is initially
   --    < 1 2 (position) 3 4 > (represented by <1 2>, <3 4>), then
   --    after calling Move_To_Finish(L), the new value of L (L') is:
   --    <1 2 3 4 (position)> (represented by <1 2 3 4>, <>).

   --------------------------------------------------------------------

   procedure Advance (
         L : in out List
      ) is abstract;

   -- requires
   --    L.Right \= <>
   -- ensures
   --    L' = there is an Item X such that
   --          (L'.Left = L.Left * <X> and L.Right = <X> * L'.Right)
   -- raises
   --    Advance_At_End

   --    If the position of L is at the end of the list when Advance is
   --    called, then the exception Advance_At_Finish gets raised.
   --    Otherwise, current position is moved one place to the right in
   --    the list.  For example, if L = <1 2 (position) 3 4 >
   --    (represented by (<1 2>, <3 4>)) when Advances is called, then
   --    the value of L returned (L') would be <1 2 3 (position) 4>
   --    (represented by (<1 2 3>, <4>)).  In this example, X in the
   --    ensures clause is 3.

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

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

-- ensures
--    L' = (L.Left, <X> * L.Right)

--    Add_Right adds the value of item X to the right of the current
--    position in list L.  For example, if L = <1 2 (position) 3 4
--    (represented by (<1 2>, <3 4>)) and X = 5 when Add_Right is
--    called, then the value of L returned (L') would be
--    <1 2 (position) 5 3 4> (represented by (<1 2>, <5 3 4>)).

---------------------------------------------------------------------

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

-- requires
--    L.Right \= <>
-- ensures
--    (L'.Left = L.Left) and (L.Right = <X'> * L'.Right)
-- raises
--    Remove_At_End

--    Remove_Right removes the item right of the current position
--    from list L and sets X to this value.  For example, if
--    L = <1 2 (position) 3 4 (represented by (<1 2>, <3 4>)) when
--    Remove_Right is called, then the value of L returned (L') would
--    be <1 2 (position) 4> (represented by (<1 2>, <4>)) and the
--    value of X (X') would be set to 3.

---------------------------------------------------------------------

function Right_Size (
      L : in List
   ) return Natural is abstract;

-- ensures
--    result =  |L.Right|

--    The function returns the total number of items to the right of
--    the current position in list L.

---------------------------------------------------------------------
function Left_Size (
      L : in List
   ) return Natural is abstract;

-- ensures
--    result =  |L.Size|

--    The function returns the total number of items to the left of
--    the current position in list L.

---------------------------------------------------------------------

   procedure Clear (
         L : in out List
      ) is abstract;

   -- ensures
   --    L' = (<>, <>)

   --    The Clear procedure removes all items (if any) from list L
   --    returning L (L') as an empty list (represented by a pair of
   --    empty strings (<>, <>).

   ---------------------------------------------------------------------

end AT_One_Way_List;
```

Appendix 5 – AT_One_Way_List.CT_Stack Package Specification

```ada
-- File: AT_One_Way_List-CT_Stack.ads
-----------------------------------------------------------------------
--
-- One_Way_List – Stack-based Implementation (package specification)
--
-- Written by Major Gibson for Comp Sci 225, Fall 1998
--
-------------------------------------------------------------------------
--
-- This package provides the operation headers and stack-based data
-- representation for a generic (unbounded) list ADT.  The list is a
-- "one way" list since it may only be traversed in one (the forward)
-- direction.  A one way List value is a sequence of values of type Item
-- (where Item is a generic type parameter to this package) and a
-- position within that sequence.  An Item may be added to the list just
-- right of the current position (Add_Right).  As long as the current
-- position is not the end (rightmost) position of the list, the item
-- to the right of the current position may be removed (Remove_Right).
-- The current position may be advanced one position to the right
-- (Advance), changed to the beginning of the list (Move_To_Start), or
-- changed to the end of the list (Move_To_Finish).  The initial value
-- of a one way List is an empty list.
--
-----------------------------------------------------------------------
-- Context

with AT_Stack.CT_Nodes;

-----------------------------------------------------------------------

generic

   --------------------------------------------------------------------
   -- Generic Package Parameters (of AT_One_Way_List)

   -- type Item is private;   -- type of values held by the List

   --------------------------------------------------------------------

package AT_One_Way_List.CT_Stack is

   --------------------------------------------------------------------
   -- Exported Type

   type List is new AT_One_Way_List.List with private;

   --------------------------------------------------------------------
   -- Exported Exceptions (of AT_One_Way_List)

   -- Remove_At_End  : exception;  -- raised by Remove_Right
   -- Advance_At_End : exception;  -- raised by Advance;

   --------------------------------------------------------------------
   -- Explicit Exported Operations

   procedure Move_To_Start (
         L : in out List
      );

----------------------------------------------------------------------

procedure Move_To_Finish (
      L : in out List
   );

---------------------------------------------------------------------

procedure Advance (
      L : in out List
   );

----------------------------------------------------------------------

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

---------------------------------------------------------------------

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

---------------------------------------------------------------------

function Right_Size (
      L : in List
   ) return Natural;

---------------------------------------------------------------------

function Left_Size (
      L : in List
   ) return Natural;

---------------------------------------------------------------------

procedure Clear (
      L : in out List
   );
   ----------------------------------------------------------------------
   -- Representation

private

   -- Create an abstract package instance of AT_Stack that specifies a
   -- stack of Items - the same type of items in the List

   package AI_Item_Stack is new AT_Stack (
            Item => Item
      );

   -- Create a concrete package instance of AI_Bounded stack by
   -- selecting the CT_Nodes (pointer-based) implementation of
   -- AI_Bounded_Stack

   package CI_Item_Stack is new AI_Item_Stack.CT_Nodes;
   use CI_Item_Stack;

   type List is new AT_One_Way_List.List with
      record
         Left  : Stack;
         Right : Stack;
      end record;

   -- This stack-based representation holds the List items in two
   -- stacks: Left and Right.  The first item in the list
   -- corresponds to the bottom item in the left stack.  The last item
   -- in the list corresponds to the bottom item in the Right stack.
   -- The current position in the list corresponds to the position
   -- between the Left and Right stack tops.  Picture the list
   -- < 1 2 3 (position) 4 5 > as the following two stacks:
   --
   --   +-------   -------+
   --   | 1 2 3       4 5 |
   --   +-------   -------+
   --     Left      Right
   --

   ----------------------------------------------------------------------

end AT_One_Way_List.CT_Stack;
```

Appendix 6 – AT_One_Way_List.CT_Stack Package Body

```
-- File: AT_One_Way_List-CT_Stack.adb
----------------------------------------------------------------------
--
-- One_Way_List Stack-Based Implementation (package body)
--
-- Written by Major Gibson for Comp Sci 225, Fall 1998
--
----------------------------------------------------------------------
--
-- This package provides the operation implementations for the stack-
-- based data representation for a generic (unbounded) list ADT.  The
-- list is a "one way" list since it may only be traversed in one (the
-- forward) direction.  A one way List value is a sequence of values of
-- type Item (where Item is a generic type parameter to this package)
-- and a position within that sequence.  An Item may be added to the
-- list just right of the current position (Add_Right).  As long as the
-- current position is not the end (the rightmost) position of the list,
-- the item to the right of the current position may be removed
-- (Remove_Right).  The current position may be advanced one position to
-- the right (Advance), changed to the beginning of the list
-- (Move_To_Start), or changed to the end of the list (Move_To_Finish).
-- The initial value of a one way List is an empty list.
--
----------------------------------------------------------------------

package body AT_One_Way_List.CT_Stack is

   -------------------------------------------------------------------
   -- Explicit Exported Operations

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

      Push(L.Right, X);            -- push X onto Right stack

   end Add_Right;

   -------------------------------------------------------------------

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

      if Size(L.Right) = 0 then      -- position is end of list
            raise Remove_At_End;     -- then raise exception
      end if;
      Pop(L.Right, X);               -- pop X from Right stack

   end Remove_Right;

   --------------------------------------------------------------------

procedure Move_To_Start (
      L : in out List
   ) is
      X : Item;                 -- temporary item holder
begin

   for i in 1 .. Size(L.Left) loop  -- move all items on left stack
      Pop(L.Left, X);               -- to right stack
      Push(L.Right, X);
   end loop;

end Move_To_Start;

--------------------------------------------------------------------

procedure Move_To_Finish (
      L : in out List
   ) is
      X : Item;                 -- temporary item holder
begin

   for i in 1 .. Size(L.Right) loop  -- move all items on right stack
      Pop(L.Right, X);               -- to left stack
      Push(L.Left, X);
   end loop;

end Move_To_Finish;

--------------------------------------------------------------------

procedure Advance (
      L : in out List
   ) is
      X : Item;                 -- temporary item holder
begin

   if Size(L.Right) = 0 then   -- if at end of list
      raise Advance_At_End;    -- then raise exception
   end if;
   Pop(L.Right, X);            -- move leftmost item on right stack
   Push(L.Left, X);            -- to rightmost item on left stack

end Advance;

--------------------------------------------------------------------

function Right_Size (
      L : in List
   ) return Natural is

begin

   return(Size(L.Right)); -- return number of items right of position

end Right_Size;
```

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

   function Left_Size (
         L : in List
      ) return Natural is

   begin

      return(Size(L.Left)); -- return number of items left of position

   end Left_Size;

   ----------------------------------------------------------------------

   procedure Clear (
         L : in out List
      ) is

   begin

      Clear(L.Right);     -- remove all items right of position
      Clear(L.Left);      -- remove all items left  of position

   end Clear;

   ---------------------------------------------------------------------

end AT_One_Way_List.CT_Stack;
```