ENGINEERING "UNBOUNDED" REUSABLE ADA GENERICS

Joseph E. Hollingsworth
Bruce W. Weide

Department of Computer and Information Science
The Ohio State University
2036 Neil Avenue Mall
Columbus, Ohio 43210-1277

holly@cis.ohio-state.edu, (614) 292-5813
weide@cis.ohio-state.edu, (614) 292-1517

Proceedings of the 10th Annual National Conference on Ada Technology, February 1992.

Abstract

Most current programming languages (including Ada) provide some means of allowing the programmer to dynamically allocate and deallocate heap storage. This permits construction of "unbounded" abstract data types, e.g., stacks, queues, one-way lists, etc. Unfortunately, the addition of dynamically allocated storage to the implementation of abstract data types is a complicated business. Unless special care is taken, it can lead to problems of storage leaks, dangling references, unwanted aliasing, and unexpected lengthy execution times (due to storage allocation and reclamation), among others. We propose a specific discipline for avoiding these problems.

1. Introduction

Section 4.2 of the Ada 9X Requirements[1] recognizes the following storage management problems associated with abstract data types (ADTs) implemented using dynamically allocated data structures ("unbounded" ADTs):

- Problem 1. Currently the implementer of an ADT has no sure way of regaining control over dynamically allocated storage. However, as noted in the Ada 9X Requirements[1], p. 19: "...the programmer should be able to gain control whenever storage is allocated and whenever a scope is deactivated."

- Problem 2. There is a proliferation of unbounded ADTs providing the same functionality, differing only in their storage management scheme. The need for different schemes is also recognized in the Ada 9X Requirements[1], p. 19: "...the ability to provide specialized storage management algorithms is often essential when tuning an application's performance." This is already happening in practice. Booch[4] typically provides two versions for each unbounded type, e.g., Stack_Sequential_Unbounded_Managed_Noniterator and Stack_Sequential_Unbounded_Unmanaged_Noniterator. These differ only in their storage management scheme.

- Problem 3. The client of an unbounded ADT typically has little control over the allocation and reclamation process, but needs this control. Again, this point is made in the Ada 9X Requirements[1], pp. 19-20: "For time-critical applications, storage allocation and reclamation actions must occur at predictable times and must be accomplished in a bounded amount of time."

This paper offers a discipline for designing unbounded ADTs based on a coherent set of engineering principles. By adhering to this discipline, an Ada software designer can develop a large class of unbounded ADTs that do not suffer from the storage management problems listed above.

One of the principles presented in this paper is related to work by Booch[4] and Musser and Stepanov[13] in that it involves an encapsulation for storage management. It is different from their work in that the encapsulation imposes stricter control over the client's use of the allocated

This material is based upon work supported by the National Science Foundation Grant No. CCR-9111892.

storage. Through this strict encapsulation, problems 2 and 3 can be addressed. Rosen[15] proposes an abstraction with stricter control than Booch or Musser and Stepanov, but this encapsulation is not used to build linked structures, as is done here. Work by Sherman[16], Muralidharan[12], and Baker[3] also addresses Problem 1.

The paper is organized as follows. Section 2 introduces a discipline for designing unbounded ADTs with acyclic linked representations. Section 3 introduces Nilpotent_Template, a package that encapsulates storage management for such ADTs. Section 4 demonstrates the use of Nilpotent_Template by two different clients, and Section 5 discusses alternative implementations for Nilpotent_Template. Section 6 extends the Nilpotent_Template concept to deal with tree and DAG structures, and Section 7 presents our conclusions.

2. Discipline for Designing Unbounded ADTs

Henceforth, when we discuss "generic packages," we mean Ada generic packages that export unbounded ADTs and operations to manipulate them. The main objective of this paper is to present a formally-based design discipline for such generic packages that is based on five engineering principles. The point is to show why the whole is greater than the sum of the parts (i.e., the engineering principles standing separately).

The individual engineering principles are:

- Principle 1. A generic package must export an initialize and a finalize operation for each exported type, to be called by the client on each variable of that type upon entry to and exit from its scope, respectively.

- Principle 2. A generic package must export only limited private types. A data movement operation that properly enforces the abstraction must be provided for variables of each exported type (e.g., Copy or Swap[7]).

- Principle 3. A generic package must export package initialize and finalize operations to be used by the client for each instance of the package upon entry to and exit from its scope, respectively.

- Principle 4. Storage management must be encapsulated in its own generic package. This abstraction must enforce strict control over access to allocated storage, and must admit a variety of possible implementations.

- Principle 5. A generic package must import storage management operations through generic parameters. These operations should be those provided by the generic package described in Principle 4.

Principle 1 is not new (see Sherman[16]), but it is necessary because the finalize operation is the only way in which the unbounded generic package can regain control of the storage allocated to variables of any type exported by the package. Principle 1 addresses problem 1.

Exporting each type as limited private, required by Principle 2, has been suggested by many, e.g., Hibbard[8], Booch[4], and Edwards[5]. A data movement operation that properly enforces the abstraction does so without creating an alias; this is required for modular verification of Ada generics (see Ernst[6]). For efficiency reasons[7] we choose the Swap operation in our designs, but the traditional Copy operation would also suffice.

Principle 3 requires a package finalization operation so that storage allocated to package-level variables can be reclaimed prior to the deactivation of the package (i.e., prior to leaving the scope in which the package was instantiated). Although there is a syntactic slot provided by Ada for optional package initialization, an explicit procedure is required of all packages by our discipline. This produces symmetric, consistent and uniform interfaces. Principle 3 also addresses problem 1.

Principle 4 is followed in part both by Booch[4] and by Musser and Stepanov[13], but their encapsulation is not strict enough. In general, what is meant by "strict" is that there is no uncontrolled aliasing of the allocated storage, i.e., all copies of pointers are made by operations provided by the abstraction. This strict control allows the implementation to use alternatives for storage reclamation that otherwise would not be available (see Section 5). These alternatives address problems 2 and 3 in that the implementation of the abstract storage management package can be "tuned" to the client's needs without changes to the client (except in the package instantiation).

Principle 5 attacks the proliferation of different versions of functionally similar unbounded ADTs by making them parametric in the storage management scheme. Also, it gives the client control over major performance factors (storage allocation and reclamation), which is crucial if a high level of ADT reuse is to be achieved, especially in real-time systems[2]. Principle 5 addresses problems 2 and 3.

To understand the details of the discipline, it is necessary to work at five different levels, or layers, of software.

Principle 4, encapsulating storage management, is concerned with the lowest two levels, the storage management abstraction (see Section 3) and its implementation (Section 5). Principles 1, 2, 3, and 5 are concerned with the next two higher levels, a generic package exporting an unbounded ADT and its implementation, which is based on the storage management abstraction (Section 4). At the highest level is the client of this generic package, whose use of the generic package is discussed in Principles 1, 2, and 3 (Section 4). This view of the discipline follows the 3C (concept, content, and context) model of software structure (see Latour[11]).

3. An Abstraction for Acyclic Pointer Structures

In this section we introduce Nilpotent_Template, an abstraction for acyclic pointer structures. Understanding this abstraction is paramount for understanding the discipline. Why is this abstraction so important? It allows us to get pointers right once and for all, eliminating the troublesome details associated with using programming language pointers when implementing acyclic linked structures. Furthermore, if a generic package is parameterized by Nilpotent_Template, its performance can be tuned by the client with respect to storage management. The generic package stays the same, while the client simply selects and instantiates an alternative implementation of Nilpotent_Template (see Section 5 for a discussion of alternative implementations).

Experience shows that the abstraction may be difficult to understand, so we begin with a simple example, working up from there. Suppose one is developing a program that requires a singly linked list for storing a character string. There is one problem: the programming language being used does not support pointers. What can be done? Simulate the pointers using an array for storing the characters while maintaining a parallel array for storing the simulated pointer link to the next character. This is common in FORTRAN programs, and is taught in some introductory data structures courses and standard texts (see Horowitz[10]). For example, suppose we have the list (a, x, r). An implementation using simulated pointers and parallel arrays might look like the following:

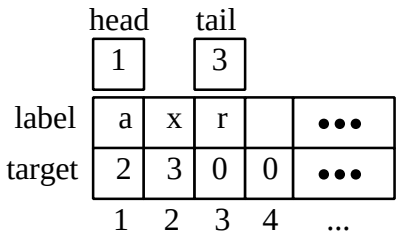


Figure 1 — Simulated pointers using parallel arrays.

To access the first item in the list, "a," use the value stored in the variable head to index into the label array. To find the next item in the list, simply index into the target array and use the value stored there as the index into the label array. The end of the list is reached when the value in the target array is zero.

Abstracting from this implementation leads to a specification for the desired generic abstract data type. There are actually two mappings at work in this example, a mapping from integers to characters, and a mapping from integers to integers. These mappings can be viewed as mathematical functions, label and target, having the following mathematical form:

label: integer $\rightarrow$ Item
target: integer $\rightarrow$ integer

By definition label and target are both total functions that form part of a complete mathematical model of the simulated pointer structure. To remain within the page limit, in examples we show only ordered pairs for which the domain value is of interest. For the above example, the functions have the following values:

label = {(1, a), (2, x), (3, r)}
target = {(1, 2), (2, 3), (3, 0)}

The Abstraction in the Form of an Ada Generic Package

The abstraction Nilpotent_Template presented below (see Figure 2) is based on the two functions label and target. It exports a program type called Position (modeled by the integers used as the domains of the functions), and it exports operations for manipulating the functions (i.e., for evaluating and changing them). The package is parameterized by the type Item. The specification has three parts: Ada code; mathematical specifications (in Ada comments beginning with --!); and English explanation (in Ada comments beginning with only --). The mathematical specifications are similar to those found in Pittel[14], with modifications to facilitate the presentation and to correspond with the Ada implementation. Upon first reading, think of the above example and use the associated explanations as an aid to understanding the specification. Then move on to the example client program found in the next section, referring back to the specification when necessary.

We have some experience introducing Nilpotent_Template to programmers in the classroom[*]. The students' first encounter with Nilpotent_Template was difficult. However, with some explanation of the specs along with an example similar to the one found in the next section, the students became comfortable with the abstraction and easily were able to use it to implement two different linked structure packages (queue and one-way list).

4. A Client of the Nilpotent_Template

This section demonstrates the use of the Nilpotent_Template introduced in the last section by providing two clients: an Ada procedure that instantiates Nilpotent_Template and an Ada generic package parameterized by Nilpotent_Template.

A Procedure to Create a Simple List

The procedure of Figure 3 creates the example list, (a, x, r), from Section 3. The reader should refer to the remainder of Figure 3 for three different views of the program's execution. Figure 3 has nine rows and four columns. The nine rows correspond with the nine lines in the program tagged with the comment "-- #." Column one contains the line number; column two contains an illustration of the abstract state; column three illustrates a simulated pointer representation; column four shows a standard representation using pointers and nodes with "next" fields.

[*] Eighteen graduate and upper-division undergraduate students, in a class called "Software Components Using Ada" (see Hollingsworth[9]).

```
--! concept Nilpotent_Template

--!        conceptual context

               generic
--!            conceptual parameters

--!                 type Item
                      type Item is limited private;
                      with procedure Initialize (x: in out Item);
                      with procedure Finalize (x: in out Item);
                      with procedure Swap (x1: in out Item; x2: in out Item);

--!            mathematics
--!                 math variables

--!                      used: integer
--!                      label: function from integer to math[Item]
--!                      target: function from integer to integer
--    Conceptually Nilpotent_Template maintains these three internal "state" variables.  The exported
--    operations manipulate these variables as well as their actual parameters.  Nilpotent_Template's
--    implementation (package body) is not obligated to represent these variables explicitly,
--    as they are mathematical abstractions, not Ada variables.

--!                 initially  "used = 0 and
--!                      for all i: integer (Item.init (label (i))  and  (target (i) = 0))"
--    Conceptually Nilpotent_Template dispenses unused positions beginning at integer number one.
--    Positions are dispensed to the client via the operation Attach_Label (see below).  Each time
--    a position is dispensed, the math variable used is incremented by one.  There is no danger of
--    eventual overflow because used is a mathematical integer, not an Ada integer.


    package Nilpotent_Template is
--!        interface

               procedure Initialize_Package;
               procedure Finalize_Package;

--!            type Position is modeled by integer
--!                 exemplar p
--!                 constraint "p >= 0"
--!                 initially  "p = 0  and  used = #used  and
--!                                 label = #label  and  target = #target"
--!                 finally         "used = #used  and  label = #label  and  target = #target"
                 type Position is limited private;
                 procedure Initialize (p: in out Position);
                 procedure Finalize (p: in out Position);
                 procedure Swap (p1: in out Position; p2: in out Position);
--   Conceptually the type Position is modeled by a mathematical integer.  Every variable p of type
--   Position is initially 0. The Initialize operation for a Position variable p does not have
--   an effect on the Nilpotent_Template's internal mathematical variables used, label and target,
--   nor does the Finalize operation.
--   In the post-condition (ensures clause) of an operation, the '#' preceding a variable
--   indicates the value of the variable at the beginning of the operation.  A variable without the
--   '#' stands for the value of the variable at the end of the operation.  The '#' is not used in a
--   pre-condition (requires clause).

               procedure Attach_Label (
                         p: in out Position;                    --! produces
                         x: in out Item                         --! consumes
                    );
--!            ensures    "used = #used + 1  and  p = used  and
--!                       for all i: integer (i /= p implies label (i) = #label (i))  and
--!                       label (p) = #x  and
--!                       target = #target"
```

```
--    Conceptually this operation allocates the next unused integer to be used as a Position value.
--    It alters the label function, mapping the new Position p to the Item x.  The new Position's
--    target is 0 because target initially maps every integer to 0.  The operation also consume
--    x; i.e., x is changed to an initial value for the type Item.

               procedure Swap_Label (
                         p: in out Position;                    --! preserves
                         x: in out Item                         --! alters
                    );
--!            requires   "p /= 0"
--!            ensures    "used = #used  and
--!                       for all i: integer (i /= p implies label (i) = #label (i))  and
--!                       label (p) = #x  and  x = #label (p)  and
--!                       target = #target"
--   Conceptually this operation allows a client to change the label function at Position p and
--   simultaneously to obtain the former label at Position p, by swapping.  Neither used nor target
--   is changed.

               procedure Apply_Target (
                         p: in out Position                     --! alters
                    );
--!            requires   "p /= 0"
--!            ensures    "p = target (#p)  and
--!                       used = #used  and  label = #label  and  target = #target"
--    Conceptually this operation applies the target function to p and sets p to the value
--    produced by the application.

               procedure Change_Target (
                         p1: in out Position;                   --! preserves
                         p2: in out Position                    --! preserves
                    );
--!            requires   "p1 /= 0 and p1 /= p2 and
--!                       there does not exist k :integer, (k = 0 and (target^k(p2) = p1))
--!            ensures    "used = #used  and  label = #label  and
--!                       for all i: integer (i /= p implies target (i) = #target (i)) and
--!                       target(p1) = p2"
--   Conceptually this operation allows a client to alter the target function by changing target(p1),
--   i.e., p1 now maps to p2 under the target function.  Neither used nor label is changed.
--   Note: The notation target^k(p2) denotes the iterated application of target to p2, k times.  For
--   example, target^2(p2) = target(target(p2)).  The requires clause must be met so that no
--   circular structures will be created!  In other words, target is a nilpotent function; hence
--   the name of the package.

               procedure Copy (
                         p1: in out Position;                   --! preserves
                         p2: in out Position                    --! produces
                    );
--!            ensures    "p2 = p1  and  used = #used  and  label = #label and target = #target"
--    Conceptually this operation allows a client to create a copy of a position p1.  None of the
--    internal mathematical variables are changed.  Note: This is similar to aliasing if pointers
--    were being used.  However this is the only way in which a client can create an alias.
--    Assignment is not available for limited private types.  Implementations of Nilpotent_Template
--    can take advantage of this situation so that dangling references and storage leaks are
--    never created.

               procedure Test_If_Equal (
                         p1: in out Position;                   --! preserves
                         p2: in out Position;                   --! preserves
                         equal: in out Boolean                  --! produces
                    );
--!            ensures    "(equal  iff  p1 = p2)  and
--!                       used = #used  and  label = #label  and  target = #target"
--    Conceptually this operation sets equal to True if and only if p1 and p2 are the same integer.

          private
```

```
            type Position_Rep;
            type Position is access Position_Rep;
        end Nilpotent_Template;
--! end Nilpotent_Template
```

Figure 2 — Nilpotent_Template Specification

```
with Nilpotent_Template,
     Built_In_Types;
use Built_In_Types;
-- Package Built_In_Types provides
Initialize,
-- Finalize and Swap operations for the
built
-- in Ada types Boolean, Character, Integer
-- and Float.

procedure Example_List is
  package Character_List_Facility is
    new Nilpotent_Template (
      Character,
      Initialize,
      Finalize,
      Swap);
  use Character_List_Facility;
  head, tail, new_tail: Position;
  c: Character;
begin
  Initialize (head);
  Initialize (tail);
  Initialize (new_tail);
  Initialize (c);                  -- 1
  c := 'a';
  Attach_Label (head, c);          -- 2
  Copy (head, tail);               -- 3
  c := 'x';
  Attach_Label (new_tail, c);      -- 4
  Change_Target (tail, new_tail); -- 5
  Apply_Target (tail);             -- 6
  c := 'r';
  Attach_Label (new_tail, c);      -- 7
  Change_Target (tail, new_tail); -- 8
  Apply_Target (tail);             -- 9
  Finalize (c)
  Finalize (new_tail);
  Finalize (tail);
  Finalize (head);
end Example_List;
```

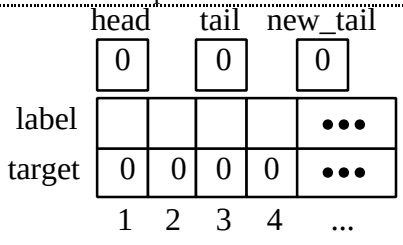
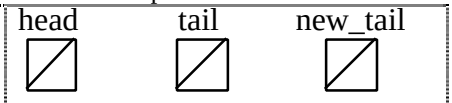
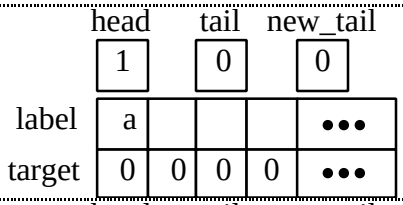
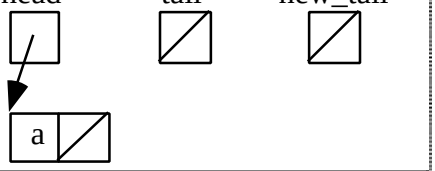
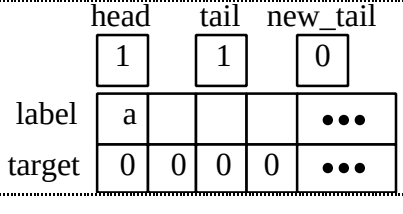
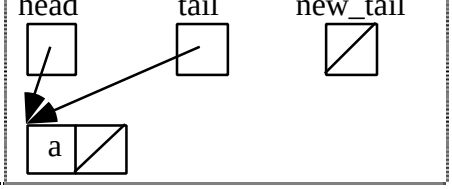
Line #	Abstract State	Simulated Pointer Representation	Standard Pointer Representation
1	head = 0 tail = 0 new_tail = 0 used = 0 label: { } target: { }		
2	head = 1 tail = 0 new_tail = 0 used = 1 label: { (1, a) } target: { (1, 0) }		
3	head = 1 tail = 1 new_tail = 0 used = 1 label: { (1, a) } target: { (1, 0) }		

Figure 3 — Three views of the example program's execution (continued to next page).

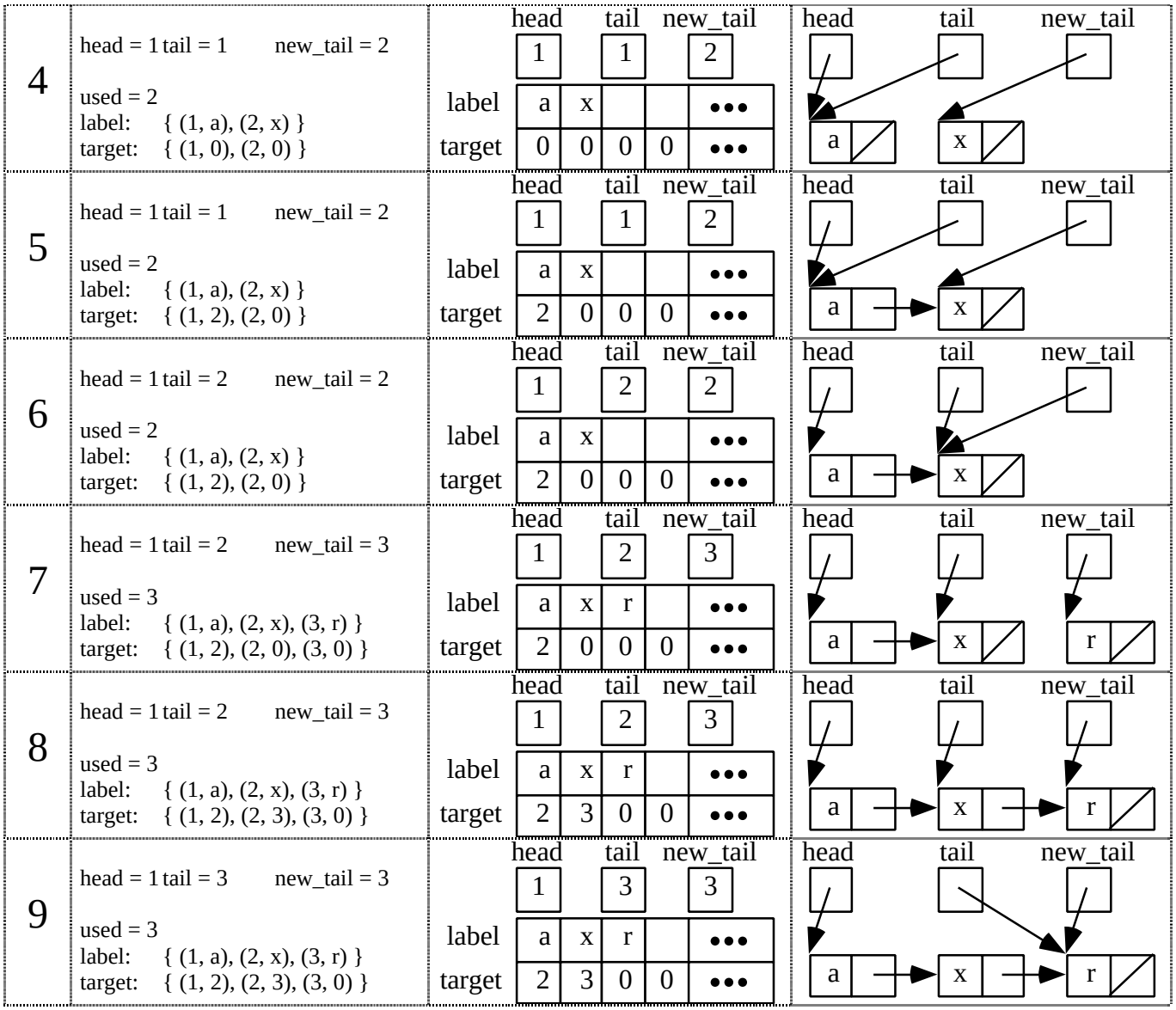


Figure 3 — Three views of the example program's execution.

The example program demonstrates how a client of the Nilpotent_Template can construct linked structures without directly using the programming language's pointer types. Additionally, this example illustrates that the programmer is now able reason about linked structures at an abstract level, as opposed to reasoning at the concrete or implementation level. Column three of Figure 3 demonstrates how a programmer reasons about linked structures in a language without pointers (e.g., FORTRAN); column four of Figure 3 demonstrates how a programmer working with a language that provides pointers reasons about linked structures (e.g., C, Pascal, or Ada). Column two, however, is how programmers should be reasoning about linked structures — at an abstract level.

Ada Generic Package Parameterized by Nilpotent_Template

It is time to examine the discipline's next three levels: a generic package (Figure 4), its implementation based on Nilpotent_Template (Figure 5), and a client of the generic package (following Figure 5). For simplicity of

presentation we have chosen stack. Abstractions such as queue, one-way list, and map illustrate the same points.

Stack_Template's specification follows the same format used for Nilpotent_Template, with formal mathematical specifications mixed with the Ada generic package specification. It has been engineered according to the principles outlined in Section 2. Conceptually, the type Stack is modeled as a mathematical string of Items (type Item is a parameter). Stack operations are specified in terms of mathematical string theory operations (e.g., concatenation). Close examination of the specification shows that the operations Initialize and Finalize satisfy Principle 1; Principle 2 is satisfied by the program type Stack being limited private, and by the Swap operation; Initialize_Package and Finalize_Package satisfy Principle 3; Principle 4 is satisfied by Nilpotent_Template itself. Nilpotent_Template's type and operations are generic parameters, satisfying Principle 5.

```
--! concept Stack_Template
--!        conceptual context
--!             uses
--!                  STRING_THEORY_TEMPLATE

                generic
--!             conceptual parameters

--!                  type Item
                       type Item is limited private;
                       with procedure Initialize (x: in out Item);
                       with procedure Finalize (x: in out Item);
                       with procedure Swap (x1: in out Item; x2: in out Item);

--!             mathematics
--!                  math facilities
--!                       STRING_THEORY is STRING_THEORY_TEMPLATE (math[Item])

--!        realization context
--!             realization parameters

--!                  facility Nilpotent_Facility is Nilpotent_Template (Item)
                       type Position is limited private;
                       with procedure Initialize(p: in out Position);
                       with procedure Finalize(p: in out Position);
                       with procedure Swap(p1, p2: in out Position);
                       with procedure Attach_Label(p: in out Position;
                                                   x: in out Item);
                       with procedure Swap_Label(p: in out Position;
                                                 x: in out Item);
                       with procedure Apply_Target(p: in out Position);
                       with procedure Change_Target(p1, p2: in out Position);
                       with procedure Copy(p1, p2: in out Position);
                       with procedure Test_If_Equal(p1, p2: in out Position;
                                                    equal: in out Boolean);

        package Stack_Template is
--!     interface

             procedure Initialize_Package;
             procedure Finalize_Package;

--!          type Stack is modeled by STRING
--!               exemplar s
--!               initially  "s = EMPTY"
               type Stack is limited private;
               procedure Initialize (s: in out Stack);
               procedure Finalize (s: in out Stack);
```

```
                procedure Swap (s1: in out Stack; s2: in out Stack);

            procedure Push (
                      s: in out Stack;                    --! alters
                      x: in out Item                      --! consumes
                 );
--!         ensures    "s = #s o #x"

            procedure Pop (
                      s: in out Stack;                    --! alters
                      x: in out Item                      --! produces
                 );
--!         requires   "s /= EMPTY"
--!         ensures    "#s = s o x"

            procedure Test_If_Empty (
                      s: in out Stack;                    --! preserves
                      empty: in out Boolean               --! produces
                 );
--!         ensures    "empty iff s = EMPTY"

       private
            type Stack is new Position;
       end Stack_Template;
--! end Stack_Template
```

Figure 4 — Generic Stack Specification

```
package body Stack_Template is

EMPTY_STACK_REP: Position;

procedure Initialize_Package is
begin
     Initialize (EMPTY_STACK_REP);
end Initialize_Package;

procedure Finalize_Package is
begin
     Finalize (EMPTY_STACK_REP);
end Finalize_Package;

procedure Initialize (s: in out Stack) is
begin
     Initialize (Position (s));
end Initialize;

procedure Finalize (s: in out Stack) is
begin
     Finalize (Position (s));
end Finalize;

procedure Swap (s1: in out Stack; s2: in out Stack) is
begin
     Swap (Position (s1), Position (s2));
end Swap;
```

```
procedure Push (s: in out Stack; x: in out Item) is
    new_top: Position;
begin
    Initialize (new_top);
    Attach_Label (new_top, x);
    Change_Target (new_top, Position (s));
    Swap (new_top, Position (s));
    Finalize (new_top);
end Push;

procedure Pop (s: in out Stack; x: in out Item) is
begin
    Swap_Label (Position (s), x);
    Apply_Target (Position (s));
end Pop;

procedure Test_If_Empty (s: in out Stack; empty: in out Boolean) is
begin
    Test_If_Equal (Position (s), EMPTY_STACK_REP, empty);
end Test_If_Empty;

end Stack_Template;
```

Figure 5 — Stack_Template Implementation

Below is a simple Stack_Client that is faithful to Principles 1, 3, and 5 of the discipline, by initializing and finalizing both the program variables and instantiated packages.

```
with Built_In_Types,
  Nilpotent_Template, Stack_Template;
use Built_In_Types;

procedure Stack_Client is

  package Nilpotent_Facility
    is new Nilpotent_Template (
      Character,
      Initialize,
      Finalize,
      Swap);

  package Stack_Facility
    is new Stack_Template (
      Character,
      Initialize,
      Finalize,
      Swap,
      Nilpotent_Facility.Position,
      Nilpotent_Facility.Initialize,
      Nilpotent_Facility.Finalize,
      Nilpotent_Facility.Swap,
      Nilpotent_Facility.Attach_Label,
      Nilpotent_Facility.Swap_Label,
      Nilpotent_Facility.Apply_Target,
      Nilpotent_Facility.Change_Target,
      Nilpotent_Facility.Copy,
      Nilpotent_Facility.Test_If_Equal);

  use Stack_Facility;

  s1: Stack;
  c: Character;

begin
  Nilpotent_Facility.Initialize_Package;
  Stack_Facility.Initialize_Package;
  Initialize (s1);
  Initialize (c);
  c := 'a';
  Push (s1, c);
  c := 'b';
  Push (s1, c);
  Finalize (c);
  Finalize (s1);
  Stack_Facility.Finalize_Package;
  Nilpotent_Facility.Finalize_Package;
end Stack_Client;
```

5. Nilpotent_Template Implementation

Before discussing alternative implementations for Nilpotent_Template, we note the fundamental properties shared by all alternatives. Remember that Nilpotent_Template has been designed for building acyclic linked structures. Consequently, the precondition for Change_Target requires that no circularity be introduced into the linked structure that is under construction (see Figure 2). This requirement allows implementations to maintain reference counts for all dynamically allocated storage (see Weide[17], Rosen[15]).

For example, when the client invokes Attach_Label, storage is allocated for a new position, and its reference count is set to one. The counts are updated by all operations of the Nilpotent_Template that change the values of Position variables or the target function. When the count reaches zero (note that the count can be decremented by a call to Finalize, Attach_Label, Change_Target, Apply_Target, or Copy) there are no positions that have access to the allocated storage. At this time, the allocated storage may be reclaimed. Because of its interface, Nilpotent_Template has complete control over aliasing of allocated storage. If it did not, then the reference count system would break down.

This is also why circular linked structures are not allowed by the Nilpotent_Template as it stands. If they are allowed (by removing the requires clause of Change_Target), it is possible to build a structure where each piece of allocated storage has a reference count equal to one, but no position has access to the structure. To detect this situation, the implementation has to follow the target chain of a position whenever a reference count is decremented; otherwise storage leaks might occur. Following the target chain is potentially inefficient.

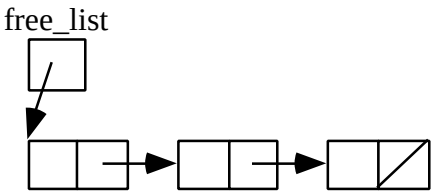
The Nilpotent_Template abstraction without the non-circulatrity constraint is useful for constructing circular structures, but does not admit an especially efficient implementation. On the other hand, it is no *less* efficient than using language-supplied primitives to allocate and deallocate storage, and it still supports abstract reasoning about client program behavior.

Here are four possible alternative strategies for storage reclamation:

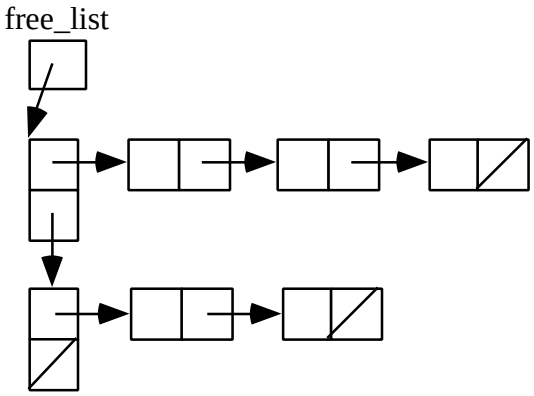
1) Use the underlying run time system's garbage collector.

2) Use UNCHECKED_DEALLOCATION.

3) Maintain an internal free list of individual pieces of storage allocated by Attach_Label, as shown:



4) Maintain an internal free list of freed linked structures. These structures have been constructed using Attach_Label and Change_Target, and have then been freed:



The first two strategies are straightforward, not warranting further discussion. The third strategy performs well when individual pieces of storage are freed (e.g., when a stack is popped), but suffers when an entire linked structure is freed. For example, if a client finalizes a stack of size N, this alternative's performance is necessarily linear in N. Why? Because it has to take each piece of storage off the stack and place it onto the free list. This might lead one to believe that the Finalize operation for a linked structure is inherently a linear-time operation, but it is not. The fourth strategy accommodates a constant time Finalize operation for linked structures of length N (see Weizenbaum[18], Weide[17]). As stated above, this implementation maintains a free list of freed linked structures rather than a free list of individually allocated pieces of storage. The Finalize operation simply adds the entire size N linked structure to the free list, in constant time. With this storage management strategy, all Nilpotent_Template operations take constant time.

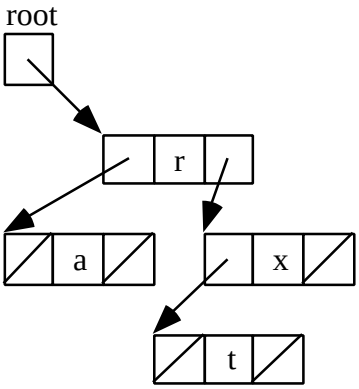
It is worth mentioning other factors that are relevant when the storage reclamation strategy maintains a free list. Unfortunately a full treatise on these factors would require another paper. For example, there might be a need to place an upper bound on the size of the free list; it might be to a client's advantage to populate the free list prior to the client requesting storage;and there is a question as to

when is the best time to finalize the Items in the data structure being reclaimed. That is, when the storage being reclaimed contains the only reference to some other large data structure, when should that data structure be finalized?

By following the discipline outlined in this paper, Nilpotent_Template can be implemented using any of the above strategies. What's more, when generic packages are designed according to the discipline, they can be instantiated with any implementation of Nilpotent_Template and work equally well, achieving plug compatibility. Finally, when clients of a generic package faithfully adhere to the initialize/finalize requirements of the discipline, no storage leaks or dangling references can be created.

6. Extending the Nilpotent Template Abstraction

With the Nilpotent_Template a software engineer can design and reason at an abstract level about singly linked structures, but not about multiply linked structures such as binary trees. To support multiply linked structures, Nilpotent_Template must be extended to support multiple target functions. The following example demonstrates a binary tree and its corresponding representation using simulated pointers and parallel arrays:



root

	1				
label	r	a	x	t	•••
target 1	2	0	4	0	•••
target 2	3	0	0	0	•••

The target function now has the following mathematical form:

target: integer × integer → integer

The target function for the above example is:
target = { ((1, 1), 2), ((1, 2), 0), ((1, 3), 4), ((1, 4), 0),
((2, 1), 3), ((2, 2), 0), ((2, 3), 0), ((2, 4), 0) }

N_Way_Nilpotent_Template has an additional generic parameter by which the client specifies the number of targets. As it turns out, Nilpotent_Template is a special case of N_Way_Nilpotent_Template (obtained by setting the number of targets to one). In practice we have implemented N_Way_Nilpotent_Template, and instantiated it with one target function to build singly linked structures and with two target functions to build binary trees.

7. Summary and Conclusion

We have introduced an engineering discipline for the construction of Ada generic packages that export "unbounded" ADTs, and their clients. The discipline addresses three problems:

- P1) regaining control over dynamically allocated storage;
- P2) the proliferation of unbounded ADTs differing only in their storage management scheme; and
- P3) no client control over an unbounded ADT's storage management scheme.

The discipline requires:

- R1) a totally encapsulated storage management module;
- R2) generic packages parameterized by and implemented with the storage management module;
- R3) total encapsulation of all types, including a data movement operation that properly enforces the abstraction;
- R4) initialization and finalization operations for all types and packages;
- R5) faithful use of initialize/finalize operations by all clients.

Problem P1 is addressed by requirements R3, R4 and R5. Problems P2 and P3 are addressed by R1 and R2.

The discipline, when properly applied, actually goes farther. It guarantees that: no storage leaks or dangling references can be created; unbounded generics are plug compatible with respect to storage management (giving the client control at instantiation time over the storage management scheme employed); the storage manager can be implemented so that all of its operations execute in constant time, including the Finalize operation for linked

structures of size N; the interface of unbounded generics is uniform and consistent (permitting compositions such as stacks of one-way lists through generic instantiation).

Straightforward, comprehensible solutions to problems such as storage management, client control over performance, plug compatibility, uniformity, consistency and composability are not easy to find. One cannot take a half-hearted approach to solving these problems, i.e., one cannot adopt principles 1, 3, and 5, for example, and hope to gain much. Only when we examine the entire picture and are willing to take a different approach at all levels, do we come up with a comprehensive discipline for solving these problems.

Acknowledgments

We are pleased to thank the members of the Reusable Software Research Group at The Ohio State University for their comments and many helpful discussions on the content of this paper.

References

1. *Ada 9X Project Report: Ada 9X Requirements*, Office of the Under Secretary of Defense for Acquisition, Washington, D.C., Dec. 1990.

2. Allen, D., et al., eds., "Catalog of Interface Features and Options for the Ada Runtime Environment," *Ada Letters*, Vol. 11, No. 8, Fall 1991, 11-2.

3. Baker, H., "Structured Programming with Limited Private Types in Ada: Nesting is for the Soaring Eagles," *Ada Letters*, Vol. 9, No. 5, July/Aug. 1991, 79-90.

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

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

6. Ernst, G.W., Hookway, R.J., Menegay, J.A., and Ogden, W.F., "Modular Verification of Ada Generics," *Comp. Lang.*, Vol. 16, No. 3/4, 1991, 259-280.

7. Harms, D.E., and Weide, B.W., "Copying and Swapping: Influences on the Design of Reusable Software Components," *IEEE Trans. on Software Eng.*, Vol. 17, No. 5, May 1991, 424-435.

8. Hibbard, P., Hisgen, A., Rosenberg, J., Shaw, M., and Sherman, M., *Studies in Ada Style*, Springer-Verlag, New York, 1983.

9. Hollingsworth, J.E., Weide, B.W., and Zweben, S.H, "Confessions of Some Used-Program Clients," *Proceedings 4th Annual Workshop on Software Reuse*, Herndon, VA, Nov. 1991.

10. Horowitz, E., and Sahni, S., *Fundamentals of Data Structures*, Computer Science Press, Inc., Rockville, Maryland, 1976.

11. Latour, L., Wheeler, T., and Frakes, W., "Descriptive and Predictive Aspects of the 3Cs Model: SETA1 Working Group Summary," *Third Annual Workshop: Methods and Tools for Reuse*, Syracuse Univ. CASE Center, Syracuse, NY, June 1990.

12. Muralidharan, S., and Weide, B.W., "Should Data Abstraction Be Violated to Enhance Software Reuse?," *Proceedings 8th Annual National Conference on Ada Technology*, ANCOST, Inc., Atlanta, GA, Mar. 1990, 515-524.

13. Musser, D., and Stepanov, A., *The Ada Generic Library: Linear List Processing Packages*, Springer-Verlag, New York, 1989.

14. Pittel, T.S., *Pointers in RESOLVE: Specification and Implementation*, M.S. Thesis, Department of Computer and Information Science, The Ohio State University, Columbus, Ohio, June 1990.

15. Rosen, S.M., "Controlling Dynamic Objects in Large Ada Systems," *Ada Letters*, Vol. 7, No. 5, Sept./Oct. 1987, 79-92.

16. Sherman, M., Hisgen, A., and Rosenberg, J., "A Methodology for Programming Abstract Data Types in Ada," *Proceedings of the AdaTEC '82 Conference on Ada*, ACM, Arlington, VA, Oct. 1982.

17. Weide, B.W., "A New ADT and Its Applications in Implementing 'Linked' Structures," Technical report, Department of Computer and Information Science, The Ohio State University, Columbus, Ohio, OSU-CISRC-TR-86-3, Jan. 1986.

18. Weizenbaum, J., “Symmetric List Processor,” *CACM*, Vol. 6, No. 9, Sept. 1963, 524 - 544.

Joe Hollingsworth holds an undergraduate degree from Indiana University and a master’s degree from Purdue University. Before returning to school as a Ph.D. candidate at The Ohio State University, he worked at Texas Instruments. He has also consulted for Battelle Memorial Institute on issues of software design in Ada. In his recent research at OSU he has developed a compiler, linker, and run-time system for RESOLVE, and has worked on a set of engineering principles that can be used to develop generic reusable software components in Ada. His Internet address is holly@cis.ohio-state.edu, and his postal address is Department of Computer and Information Science, 2036 Neil Avenue Mall, Columbus, Ohio 43210.

Bruce W. Weide is Associate Professor of Computer and Information Science at The Ohio State University. He received his B.S.E.E. degree from the University of Toledo in 1974 and the Ph.D. in Computer Science from Carnegie Mellon University in 1978. He has been at Ohio State since 1978. His research interests include various aspects of reusable software components and software engineering in general: software design-for-reuse, formal specification and verification, data structures and algorithms, and programming language issues. He has also published recently in the area of software support for real-time and embedded systems. His Internet address is weide@cis.ohio-state.edu, and his postal address is the same as shown above.