CHAPTER II

First Principles for Constructing Components in Ada

In this chapter we fix the programming language to be Ada and the component type to be abstract data types. We introduce principles for constructing Ada components that have abstraction barriers with no implementation leaks. This is achieved through the careful use of abstraction, information hiding, and encapsulation. Section 2.1 concentrates on principles for constructing a component’s package specification. Section 2.2 introduces principles that apply to the implementation of each client of a component. Section 2.3 introduces principles for implementing a component’s package body.

2.1 Component Interface Principles

This section defines principles that concern a component’s structural interface. The presentation begins with an example of a component whose structure is similar to components found in the literature. This component is then refined by the application of the principles. The introduction of each new principle is accompanied by an explanation of the improvement, an illustrative code example, and references to the literature.

The explanation of a particular principle often includes a discussion of how the principle supports the understandability property. Common sense tells us that if we strive for uniformity and consistency in the design of our components, then they will be more understandable by the client. Others have mentioned this also, e.g., Booch calls this the “Principle of Least Astonishment” [Booch 87, page 59]. Many times, in pursuit of uniformity and consistency, we promote the use of one construct over another and thus reduce the number of choices at any particular design step. In this sense, we are making the component design process more understandable for the engineer. Ultimately, we are reducing the cognitive load on both the component’s client programmer and the component’s designer, and that is exactly what we want.

The examples are based on an ordinary first-in, first-out (FIFO) queue. Conceptually, one can imagine such a component manipulating a sequence of items that are enqueued at one end of the queue and dequeued at the other end. Also, our examples sometimes include both the component’s structural interface (package specification) and its implementation (package body) because the effects of choices made in the design of a component’s structural interface often manifest themselves in the component’s implementation.
It is not often that we find other authors explicitly stating principles about component design. Therefore, the references given are usually to their program examples, which we assume (unless instructed otherwise) those authors consider to illustrate “good” practices.

2.1.1 The Unit of Modularity

Principle 1 — Make generic packages the unit of modularity. Physically separate the package specification and the package body by placing them in separate files.

Why use packages?

Packages provide the obvious means in Ada for creating abstract models for our programming types, or abstract data types (ADTs). The correctness and understandability properties are supported by the use of ADTs because clients can understand and reason with implementation-independent abstractions more easily than they can with the actual implementations.

Additionally, packages can provide language-enforced encapsulation of the component’s representation. Encapsulation supports the reusability property by forcing the client to rely on the component’s exported interface. Reliance on the exported interface enhances the chances for plug-compatible components. Encapsulation supports the understandability property through its support for the design technique of information hiding, i.e., the component is more understandable to the client programmer because irrelevant details are hidden. Encapsulation supports the correctness property because the component implementer is guaranteed that the client cannot inadvertently or intentionally manipulate the encapsulated representation by subverting the predefined interface.

Finally, if we did not make use of the package construct, then we would have to rely on procedures, functions and unencapsulated data types for our components. Clearly this is no better than what is currently done using earlier generation languages, e.g., C and Pascal, whose shortcomings are well known.

The following is an Ada package specification for a queue of integers component. It is similar in structure to components found in the literature; see [Ada 83, page 201], [Gehani 83, page 116], [Habermann 83, page 180], and [St. Dennis 86, page 88]:

```ada
package Queue_1 is
    procedure Enqueue (x: in Integer);
    procedure Dequeue (x: out Integer);
    Underflow: exception;
end Queue_1;
```

Queue_1 might be implemented using a standard singly linked list of nodes allocated from heap storage, as shown in the following package body. (Note: Unlike C or Pascal, Ada’s
access type (i.e., pointer type) does not require a special symbol (e.g., C’s “->” or Pascal’s “^”) to signify a pointer dereference.

```ada
with Unchecked_Deallocation;
package body Queue_1 is

--- Declarations --------------------------------------
--- Declarations --------------------------------------

type Node_Rep;

type Node is access Node_Rep;

type Node_Rep is record
  label: Integer;
  next: Node;
end record;

type Queue_Rep is record
  front: Node;
  back: Node;
end record;

type Queue is access Queue_Rep;
q: Queue;

--- Local Operations ----------------------------------
--- Local Operations ----------------------------------

procedure Free is new
  Unchecked_Deallocation (Node_Rep, Node);

--- Exported Operations -------------------------------
--- Exported Operations -------------------------------

procedure Enqueue ( 
  x: in Integer 
) is
begin
  q.back.next := new Node_Rep;
  q.back := q.back.next;
  q.back.label := x;
  q.back.next := null;
end Enqueue;

procedure Dequeue ( 
  x: out Integer 
) is
begin
  old_front: Node;
  if q.front = q.back then
    raise Underflow;
  else
    old_front := q.front;
    q.front := q.front.next;
    x := q.front.label;
    Free (old_front);
  end if;
end Dequeue;

begin
  q := new Queue_Rep;
  q.front := new Node_Rep;
  q.back := q.front;
end Queue_1;

A client of Queue_1 might look like the following:

with Queue_1;
  procedure Queue_1_Client is
    x: Integer;
  begin
    Enqueue (5);
    Enqueue (3);
    ...
    Dequeue (x);
    Dequeue (x);
  end Queue_1_Client;

Why use generic packages?

Before proceeding, we should note that Principle 1 is similar to, but not the same as, St. Dennis’ guideline: “Use library unit package specifications as the encapsulation mechanism for directly reusable software...” [St. Dennis 86, page 72]. First of all, St. Dennis’ guideline is for a “unit of reusability,” not a unit of modularity, i.e., it is only for components deemed “directly reusable.” Second, we choose generic packages instead of packages, which we explain next.

Note that Queue_1_Client gains access to package Queue_1 through Ada’s “with” construct. There is no way of preventing other parts of the program from “with”ing Queue_1, and there is no way of knowing if any other part of the program has “with”ed Queue_1 (short of inspecting the entire program’s source). This gives rise to a situation very much analogous to the FORTRAN “common” block. Any package-level variables declared in the package body are common (in the sense of FORTRAN’s common block) to all clients that “with” the package. To reason confidently about a client that “with”s a package requires the following: (1) first peeking into the package body to see if it maintains any state; and if it does (as in our example, i.e., see the Queue variable declared in the
package body of Queue_1 above), then (2) inspecting the entire program to determine if any other parts “with” the same package; and if they do, then (3) inspecting those parts of the program to see if they interfere with the client’s use of the package. Obviously, given this situation, we have lost our ability to reason locally about clients.

The Queue_2 package that follows is identical to Queue_1 except that it is generic. The client must first “with” Queue_2, and then instantiate it. All instances of Queue_2 are independent of each other, eliminating unwanted sharing between different clients. Furthermore, the client can create more than one instance if multiple separate queues are required.

generic package Queue_2 is
    procedure Enqueue (x: in Integer);
    procedure Dequeue (x: out Integer);
    Underflow: exception;
end Queue_2;

A client of Queue_2 might look like the following:

with Queue_2;
procedure Queue_2_Client is
    package Q1 is new Queue_2;
    package Q2 is new Queue_2;
    x: Integer;
begin
    Q1.Enqueue (5);
    Q2.Enqueue (3);
    ...
    Q1.Dequeue (x);
    Q2.Dequeue (x);
end Queue_2_Client;

2.1.2 Location of Abstract State

A problem with package Queue_2 is that there is no way to move the instance around the program, i.e., instances cannot be passed as parameters and therefore cannot be shared by or used to communicate among distinct program units, when that is appropriate. Package composition presents another problem. There are container components that are generic with respect to the items they contain, e.g., a stack can be generic and a client can instantiate it using the built-in type integer, to create a stack of integers. There is no way to compose one of these generic components with a package like Queue_2 because it does not export a type. To address these problems we introduce the next principle:

**Principle 2** — Export a type so that abstract state is maintained in variables of that type, not in package instances.
Here is an example of a package designed using Principle 2:

```verbatim
generic
package Queue_3 is
  type Queue is private;
  procedure Enqueue (q: in out Queue; x: in Integer);
  procedure Dequeue (q: in out Queue; x: out Integer);
Underflow: exception;
private
  type Queue_Rep;
  type Queue is access Queue_Rep;
end Queue_3;
```

Note the following changes that have occurred between Queue_2 and Queue_3:

- Type Queue is now exported as a private type. Abstract state is maintained in Queue variables declared by the client. Abstract state is no longer maintained in variables declared inside the package body. Exporting a type as private directs the compiler to support assignment and equality testing between variables of the type. This solves the problem of moving queues around in the program. (See [NASA 87, page 7-11], [Musser 89, page 15], [Luckham 90, page 198] and [Savitch 92, page 746] for examples of packages that export private types.)

- Enqueue and Dequeue now have formal Queue parameters so they operate on Queue variables passed as arguments.

- There is now a private part containing information about the representation. An alternative design choice could have exported type Queue as an “open” type, giving the client full access to the representation, i.e., there would be no encapsulation. We have chosen not to do this for the obvious reasons. However, one can find examples of this style of design in [Feldman 85, page 140], [Feldman 92, page 619], [Habermann 83, page 206] and [Skansholm 88, page 361].

Support for the properties is as follows:

- Composability — Package composition with container components is now possible since Queue_3 exports a type.

- Understandability — Reasoning about components that maintain abstract state in component instances rather than in variables of the exported type introduces an extra proof obligation [Krone 88]. (The extra obligation involves showing that there is no undesired interference between different clients of the same component instance.) Components that maintain abstract state only in variables of their exported types do not impose this extra proof obligation, reducing the cognitive load for the client programmer in reasoning about program behavior.

A client of Queue_3 might look like the following:

```verbatim
with Queue_3;
procedure Queue_3_Client is
  package Queue_Facility is new Queue_3;
  q1, q2: Queue_Facility.Queue;
```
x: Integer;
begin
    Queue_Facility.Enqueue (q1, 5);
    Queue_Facility.Enqueue (q2, 3);
    ...
    Queue_Facility.Dequeue (q1, x);
    Queue_Facility.Dequeue (q2, x);
end Queue_3_Client;

2.1.3 Variable Finalization

This section is concerned with components that dynamically allocate resources and how to reclaim those resources after they are no longer needed.

The following diagram illustrates one possible representation of our abstract Queue of integers (commonly found in data structures textbooks). It is a standard singly linked list of nodes allocated from heap storage. The type Queue is represented as an access type (pointer) to a record that maintains pointers to the front and the back of the one-way linked list.

![Figure 5 — Linked List Representation for Queue](image)

Suppose Queue_3’s implementation dynamically allocates storage when an item is enqueued and deallocates storage when an item is dequeued. If a client of Queue_3 ceases to need its Queue variable, then the only possible way for the client to guarantee that there is no storage leak is to explicitly iterate a loop dequeueing items until the queue is empty. Here is Queue_3’s package body:

```pascal
with Unchecked_Deallocation;

package body Queue_3 is
    type Node_Rep;
    type Node is access Node_Rep;
```
type Node_Rep is record
  label: Integer;
  next: Node;
end record;

type Queue_Rep is record
  front: Node;
  back: Node;
end record;

--- Local Operations -----------------------------------
--- Exported Operations -------------------------------

procedure Enqueue (
  q: in out Queue;
  x: in Integer
) is
begin
  if q = null then
    q := new Queue_Rep;
    q.front := new Node_Rep;
    q.back := q.front;
  else
    q.back.next := new Node_Rep;
    q.back := q.back.next;
  end if;
  q.back.label := x;
  q.back.next := null;
end Enqueue;

procedure Dequeue (
  q: in out Queue;
  x: out Integer
) is
  old_front: Node;
begin
  if q = null then
    raise Underflow;
  else
x := q.front.label;
old_front := q.front;
q.front := q.front.next;
Free (old_front);
if q.front = null then
  Free (q);
end if;
end if;
end Dequeue;
end Queue_3;

Relying on the client programmer to implement the client so that it empties each Queue when finished has the following problems:

- It presupposes an understanding of the Queue’s implementation, i.e., knowledge that it dynamically allocates resources; otherwise there is no point in emptying out the Queue. Thus, the required reasoning about the Queue inappropriately includes its implementation details.

- Emptying out the Queue may not be done consistently because of the effort required to write and debug the code. This can lead to storage leaks.

- Even if emptying out the Queue is done consistently, separate clients will end up with separate, probably identical loops for doing so — a missed opportunity for reuse.

- For some implementations of Queue (e.g., an array-based representation) emptying out the Queue is not required and leads to less efficient clients.

Since there is no language support in Ada for automatic finalization of variables when control leaves their scope, we have to rely on the client programmer to know when a variable needs finalization. However, that does not mean that we must rely on him/her to know how it should be finalized (other than making a simple procedure call). The following principle takes the “knowing how” out of the hands of the client programmer:

**Principle 3** — For each exported type T, export a finalization operation as follows:

```ada
procedure Finalize (x: in out T);
```

Support for the properties is as follows:

- Correctness — The designer of a component is required to consider the problem of variable finalization from the outset. Furthermore, the client programmer is more likely to invoke the Finalize operation than to design and debug his/her own ad hoc finalization routine.
• Reusability — Clients are more apt to need components that frugally use resources, e.g., a component that reclaims storage is more attractive than one that has storage leaks. Furthermore, by exporting a finalization operation, the component designer can implement some particularly interesting reclamation strategies in which the finalization operation can execute in constant time; see [Weizenbaum 63] and [Weide 86]. Compare this to the ad hoc finalization of the queue by the client outlined above, which necessarily executes in time that is linear in the length of the queue.

• Understandability — The client simply invokes the Finalize operation rather than being forced to construct some ad hoc finalization code for each of the variables that the client uses. The mode “in out” is required for the formal parameter for uniformity and consistency. If the exported type is represented by an access type, then Ada’s Unchecked_Deallocation generic procedure may be used to reclaim the storage. The procedure created by instantiating Unchecked_Deallocation for the exported type has an “in out” parameter. This forces Finalize’s parameter to have mode “in out.” To be consistent and uniform we choose to make all Finalize operations have “in out” mode for their formal parameter. Finally, since all exported types from all components have a finalization operation, there is more uniformity, and fewer decisions to be made. All these serve to reduce the cognitive load for the component’s designer and client programmer.

Package specification Queue_4 exports a Finalize operation which reclaims allocated storage. This is similar to packages found in [Hibbard 83, page 38] and [Edwards 90, page 40].

generic package Queue_4 is
  type Queue is private;
  procedure Finalize (q: in out Queue);
  procedure Enqueue (q: in out Queue; x: in Integer);
  procedure Dequeue (q: in out Queue; x: out Integer);
  Underflow: exception;
private
  type Queue_Rep;
  type Queue is access Queue_Rep;
end Queue_4;

Possible implementations of Finalize, Enqueue and Dequeue follow:

procedure Finalize (q: in out Queue)
  is
    old_node: Node;
begin
  if q /= null then
    while q.front /= null loop
      old_node := q.front;
      q.front := q.front.next;
      Free (old_node);
    end loop;
    Free (q);
  end if;
end Finalize;
procedure Enqueue (  
    q: in out Queue;  
    x: in Integer  
) is  
begin  
    if q = null then  
        q := new Queue_Rep;  
        q.front := new Node_Rep;  
        q.back := q.front;  
    elsif q.front = null then  
        q.front := new Node_Rep;  
        q.back := q.front;  
    else  
        q.back.next := new Node_Rep;  
        q.back := q.back.next;  
    end if;  
    q.back.label := x;  
    q.back.next := null;  
end Enqueue;

procedure Dequeue (  
    q: in out Queue;  
    x: out Integer  
) is  
    old_front: Node;  
begin  
    if q = null or else q.front = null then  
        raise Underflow;  
    else  
        x := q.front.label;  
        old_front := q.front;  
        q.front := q.front.next;  
        Free (old_front);  
    end if;  
end Dequeue;

The client can now finalize Queue variables simply by calling the Finalize operation at the end of the scope where a Queue variable has been declared; for example:

with Queue_4;  
procedure Queue_4_Client is  
    package Queue_Facility is new Queue_4;  
    q1, q2: Queue_Facility.Queue;  
begin  
    Queue_Facility.Enqueue (q1, 5);  
    Queue_Facility.Enqueue (q2, 3);  
    ...  
    Queue_Facility.Finalize (q2);
Queue_Facility.Finalize (q1);
end Queue_4_Client;

2.1.4 Variable Initialization

Examination of Queue_4’s package specification shows that there is no exported initialization operation. There are two problems with this design: (1) there is a lack of uniformity, i.e., if variables must be explicitly finalized, then it makes sense that they should also be initialized; (2) trying to hide initialization in the package body can make the exported operations less efficient, or force the designer to choose a less than optimal representation. This section discusses these two problems and introduces a principle to eliminate them.

Referring to the queue example, we see that packages Queue_1 and Queue_2 export no initialization operation. This is not a problem because initialization of the package body’s internal variables can be performed by package-specific initialization code. This code automatically gets executed upon package elaboration. Since packages Queue_3 and Queue_4 export type Queue, and clients declare variables of this type, the package-specific initialization code is of no help. In this situation, the burden of initialization falls to one of three places: it is done by all the exported operations; or it is done with default expressions supplied with the representation’s declaration; or it is done automatically if the representation is an access type (guaranteed by Ada). Since type Queue is represented as an access type, the representation of every Queue variable is automatically initialized to null. The utility of null as the initial value of a Queue variable’s representation is dubious. For example, in Queue_4, each time Finalize, Enqueue, and Dequeue are invoked, they must first check to see if their formal Queue parameter is equal to null. This introduces an unnecessary inefficiency into the implementation.

We can eliminate these inefficiencies and improve the clarity of the code by initializing Queue variables with a more useful initial configuration, such as the following:

![Figure 6 — Initial Configuration for a Queue Variable](image)

The “dummy” node is a standard method of eliminating special cases from the code for Enqueue and Dequeue because even an empty Queue variable still contains a node in its representation. Given this initial configuration, the test for null can be eliminated from each of the operations. Unfortunately, Ada provides no way, in general, to create an arbitrary desirable representation configuration such as this one as a default.

The following principle addresses variable initialization:
Principle 4 — For each exported type T, export an initialization operation as follows:

```ada
procedure Initialize (x: in out T);
```

Support for the properties is as follows:

- Correctness — The client programmer can determine locally if a locally declared variable has been initialized simply by looking for the call to Initialize at the beginning of the variable’s scope.

- Reusability — Initialization can be performed once and for all immediately after variable declaration and further tests for initialized variables no longer have to be performed by the other exported operations. This permits more efficient implementation of the exported operations through the elimination of special case detection code.

- Understandability — First, the mode “in out” is required for the formal parameter for uniformity and consistency. If the exported type is represented by an access type and if the storage allocated must be initialized, then the pointer will have to be dereferenced after the storage is allocated, e.g., see the implementation of Initialize below. That means the formal parameter is used on the left hand side of an assignment statement. Ada does not allow formal parameters of mode “out” to appear on the left hand side of an assignment statement. This forces Initialize’s parameter to have mode “in out.” To be consistent and uniform we choose to make all Initialize operations have “in out” mode for their formal parameter.

Finally, by exporting Initialize along with Finalize, our packages achieve symmetry with respect to variable initialization and finalization, adding additional support for understandability.

At this point we might be tempted to implement Initialize as a function returning the initial value. This works with private types, but not with limited private types because there is not an assignment operation available with limited private types. In upcoming Section 2.1.6 we introduce a principle that requires all exported types to be limited private. Therefore, at this time we implement Initialize as a procedure.

Queue_5 now exports an Initialize procedure which returns an initialized Queue. This is similar to packages found in [Hibbard 83, page 38] and [Edwards 90, page 40]. The change to Queue_4 involves adding Initialize to the package and eliminating the special case code from Finalize, Enqueue and Dequeue.

```ada
generic
package Queue_5 is
  type Queue is private;
  procedure Initialize (q: in out Queue);
  procedure Finalize (q: in out Queue);
  procedure Enqueue (q: in out Queue; x: in Integer);
```
**procedure** Dequeue (q: **in out** Queue; x: **out** Integer);
Underflow: **exception**;

**private**

type Queue_Rep;

type Queue is **access** Queue_Rep;
end Queue_5;

Possible implementations of Initialize, Finalize, Enqueue and Dequeue are as follows:

**procedure** Initialize (q: **in out** Queue)
  **is**
begin
  q := **new** Queue_Rep;
  q.front := **new** Node_Rep;
  q.back := q.front;
end Initialize;

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

**procedure** Finalize (q: **in out** Queue)
  **is**
  old_node: Node;
begin
  **while** q.front /= **null** **loop**
  old_node := q.front;
  q.front := q.front.next;
  Free (old_node);
  **end loop**;
  Free (q);
end Finalize;

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

**procedure** Enqueue (q: **in out** Queue; x: **in** Integer)
  **is**
begin
  q.back.next := **new** Node_Rep;
  q.back := q.back.next;
  q.back.label := x;
  q.back.next := **null**;
end Enqueue;

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

**procedure** Dequeue (q: **in out** Queue; x: **out** Integer)
  **is**
begin
  old_front: Node;
begin
  if q.front = q.back then
    raise Underflow;
  else
    old_front := q.front;
    q.front := q.front.next;
    x := q.front.label;
    Free (old_front);
  end if;
end Dequeue;

The client can now initialize and finalize Queue variables simply by calling the Initialize and Finalize operations at the beginning and end of the scope where a Queue variable has been declared; for example:

with Queue_5;
procedure Queue_5_Client is
package Queue_Facility is new Queue_5;
  q1, q2: Queue_Facility.Queue;
  x: Integer;
begin
  Queue_Facility.Initialize (q1);
  Queue_Facility.Initialize (q2);
  ...
  Queue_Facility.Enqueue (q1, 5);
  Queue_Facility.Enqueue (q2, 3);
  ...
  Queue_Facility.Finalize (q2);
  Queue_Facility.Finalize (q1);
end Queue_5_Client;

Before moving on to the next principle, it is worth mentioning that some (e.g., [Wallis 90, page 118]) suggest that explicitly exporting an initialization operation is not the correct approach for achieving variable initialization. They argue that initialization can be achieved implicitly by always using a record for the type’s representation and by specifying default values for each of the fields of the record at its declaration. This is not always a legitimate approach, because it will not work for limited private types. It requires the use of the assignment operator, and limited private types do not have that operator available to them. Later (Section 2.1.6) we show why all types should be limited private.

2.1.5 Testing Preconditions

Suppose a client of Queue_5 needs to Dequeue all the items from one queue, performing an action on some of the dequeued items, while enqueueing the rest of the items on another queue. The code might look something like the following:

begin
  loop
    Queue_Facility.Dequeue (q1, x);
    if x > 7 then
Queue_Facility.Enqueue (q2, x);
else
  -- Do something with x when x <= 7.
end if;
end loop;
exception
  when Queue_Facility.Underflow => ...;
end;

There are a number of unusual things about this loop. For example, it appears to be an
infinite loop since there is no test for an exit condition; the exit from the loop occurs only
when an exception is raised by Dequeue. The only way to determine the exit condition is
by examining the exception handler and by knowing that Dequeue raises the Underflow
exception. To associate the exception handler with the loop a begin-end block has to be
introduced.

Although it is possible to correctly design and implement clients of Queue_5 that are similar
to the above example, it should not be the case that the design of a component forces the
client to use such a baroque style. Intuitively, Dequeue has the precondition that it should
be called with a non-empty queue. The problem with Queue_5 is that the client is not able
to test this precondition directly. This observation gives rise to the next principle:

Principle 5 — If any exported operation has a precondition, export
operations sufficient for a client to test that precondition.

By adding to Queue_5’s package specification the following function (used to test for an
empty queue), we can rewrite the loop above using standard structured programming
techniques:

function Is_Empty (q: in Queue) return Boolean;

The rewritten loop is:

while not Is_Empty (q1) loop
  Queue_Facility.Dequeue (q1, x);
  if x > 7 then
    Queue_Facility.Enqueue (q2, x);
  else
    -- Do something with x when x <= 7.
  end if;
end loop;

There is an efficiency problem with just adding Is_Empty to Queue_5. If we allow
Queue_5 to continue to export the Underflow exception, then any client that uses standard
structured programming techniques — like the loop above — will necessarily pay an
execution-time price for it. This might tempt clients to program using the unusual style
found in the first loop. In the “while” loop above, the client tests for an empty queue at the
top of the loop. When Dequeue is called, it too must test for an empty queue in order to
know if it should raise the Underflow exception. But there is no way in this particular
situation that Dequeue could be called with an empty queue (because of the test at the top of
the loop), and therefore the second test is logically redundant. It should not be the case that
the design of the component forces the client to suffer a performance penalty in situations
similar to these.

Therefore, we propose the next principle:

** Principle 6 — Do not export any exceptions, and design exported
operations so that they do not raise any exceptions.**

The new package specification is:

```plaintext
generic
package Queue_6 is
  type Queue is private;
  procedure Initialize (q: in out Queue);
  procedure Finalize (q: in out Queue);
  procedure Enqueue (q: in out Queue; x: in Integer);
  procedure Dequeue (q: in out Queue; x: out Integer);
  function Is_Empty (q: in Queue) return Boolean;
private
  type Queue_Rep;
  type Queue is access Queue_Rep;
end Queue_6;
```

The operations Dequeue and Is_Empty might now be implemented as follows:

```plaintext
procedure Dequeue (q: in out Queue; x: out Integer)
begin
  old_front := q.front;
  q.front := q.front.next;
  x := q.front.label;
  Free (old_front);
end Dequeue;
```

```plaintext
function Is_Empty (q: in Queue)
begin
  return q.front = q.back;
  ```
Now the client is responsible to make sure that the precondition for each operation is satisfied prior to invoking that operation (for package Queue_6, only Dequeue has a precondition). Others (e.g., [Meyer 88, page 117]) have also proposed this delegation of responsibility, but most authors seem satisfied with exceptions and testing preconditions.

Support for the properties is as follows:

- **Correctness** — Client programmers can reason locally, i.e., they do not have to examine the code of the component’s exported operations to determine which operation raises which exception and under what circumstances.

- **Reusability** — Principle 6 leads to components and clients that work together more efficiently than they would otherwise (as demonstrated by the example above). It is also interesting to note that exception-less components can be used to implement layered components with exceptions, while incurring little or no performance penalty. The performance penalty usually amounts to an extra procedure call which can be eliminated through the use of procedure inlining. On the other hand, components that export exception-raising operations (e.g., Queue_6’s Dequeue), can be used to implement exception-less versions, but must always involve a performance penalty. There is a performance penalty because the exception-raising operation must perform a test in order to determine if there is an exceptional condition, regardless of the client program. In this regard, exception-less versions are more reusable than the exception-raising versions since they can be used to implement the exception-raising versions without a performance penalty, but not the other way around. Section 3.2.4 demonstrates how an exception-raising version can be layered on top of an exception-less version.

- **Understandability** — Engineers may use standard structured programming techniques when implementing component clients.

### 2.1.6 Limited Private Types

Key to our discipline is the ability of an engineer to be able to reason about a client within a local area around the client. The local area is limited to the client and the specifications of the packages on which the client directly depends. Excluded from the local area are any details concerning the representation or implementation of the packages on which the client depends. Figure 7 illustrates this point with an example of a client of a Queue package and some “Other Package.” The dashed line bounds the items that the engineer should be considering when reasoning about the client. They are the Client specification (the oval labeled Client), the Client implementation (the rectangle labeled Client Implementation), and the Queue and the Other Package’s specifications (the ovals labeled Queue and Other Package).
There are three ways in Ada in which a package may export a type: as an “open” type, a private type, or a limited private type. We eliminated the “open” option because it provides no language-enforced encapsulation. Currently, Queue_6 exports its type as a private type. The following example illustrates some of the subtleties involved in choosing between private and limited private types, and illustrates how easy it is to design packages that do not allow local reasoning about program behavior.

For the example, suppose we restrict our queue to be a bounded queue, i.e., conceptually it is a component that manipulates a sequence of items which are enqueued at one end and dequeued at the other, but allows at most some fixed number of items to be in the queue at any one time. Two possible representations for this queue might be a record containing an array with two indices, one index for the front of the queue and the other for the back (see Figure 8); or a record containing two pointers, one to the front of a singly linked list of the items, the other to the back of the list (see Figure 9).

![Figure 8 — Array Representation of a Bounded Queue](image)

Figure 9 — Singly Linked List Representation of a Bounded Queue

Note the first node in the linked-list representation of Figure 9 is labeled “dummy.” The queue Initialize operation initializes the queue with only the dummy node in the linked list, and it remains there throughout the lifetime of the queue. With this dummy node present, Enqueue never has to check for the special case of no nodes in the linked list, thus eliminating special case code. Dequeue, on the other hand, might be implemented so as to remove the node following the dummy node whenever it is invoked, leaving the original dummy node in place until Finalize is called.

The package specification for Bounded_Queue follows:

```
generic
package Bounded_Queue is
    type Queue is private;
    procedure Initialize (q: in out Queue);
    procedure Finalize (q: in out Queue);
    procedure Enqueue (q: in out Queue; x: in Integer);
    procedure Dequeue (q: in out Queue; x: out Integer);
    function Is_Empty (q: in Queue) return Boolean;
    function Is_Full (q: in Queue) return Boolean;
private
    ...
end Bounded_Queue;
```

The declarations in the private part for the array representation might be:

```
Max_Size: constant := 50;
    type Contents_Rep is array (1..Max_Size) of Integer;
    type Queue is record
        front, back: Integer;
        contents: Contents_Rep;
    end record;
```

The declarations for the singly linked list representation might be:

```
type Node_Rep;
    type Node is access Node_Rep;
    type Queue is record
        front, back: Node;
    end record;
```
Now suppose we reason about the behavior of the following client program by the traditional method of simulating execution of the code. The question is, what does the following client print?

```plaintext
with Bounded_Queue;
Procedure Client is
  package Queue_Facility is new Bounded_Queue;
  use Queue_Facility;
  q1, q2: Queue;
  x: Integer;
begin
  Initialize (q1); -- Line 1
  Initialize (q2); -- Line 2
  Enqueue (q1, 7); -- Line 3
  Enqueue (q1, 2); -- Line 4
  Enqueue (q1, 9); -- Line 5
  q2 := q1; -- Line 6
  Dequeue (q2, x); -- Line 7
  if q1 = q2 then -- Line 8
    print ("equal"); -- Line 9
  else
    print ("not equal"); -- Line 10
  end if;
  Finalize (q2);
  Finalize (q1);
end Client;
```

Before hand-executing Client, notice that it makes use of the explicitly exported operations Initialize, Enqueue, Dequeue and Finalize, and the implicitly exported operators `:=` (assignment) and `=` (equality testing). These implicit operations are automatically supported for variables of the exported type when it is a private type.

**Simulated execution with Queue’s array-based representation** — Lines 1 and 2 simply initialize q1’s and q2’s statically allocated storage (e.g., initialization of the indices front and back). Lines 3, 4 and 5 enqueue 7, 2 and 9 onto q1 giving us the configuration found in Figure 8. Line 6 makes use of the assignment statement to make a copy of q1’s representation so that q2 has the same values in each of its fields after the assignment. Line 7 dequeues the value 7 from q2, incrementing the index q2.front. Line 8 tests for equality between q1 and q2. When considering the conceptual model for a queue (i.e., a sequence of items) one should intuitively believe that q1 does not equal q2, since Client has dequeued an item from q2. The comparison at Line 8 compares the contents of the two records and they are indeed different (q1.front /= q2.front) so Line 10 is executed and “not equal” is printed out.

**Simulated execution with Queue’s linked-list representation** — Line 1 initializes q1 by dynamically allocating a dummy node from heap storage and setting q1.front and q1.back to point to this dummy node. Line 2 does the equivalent for q2. Lines 3, 4 and 5 enqueue 7, 2 and 9 onto q1 giving us the configuration found in Figure 9. Line 7 makes a copy of q1’s representation so that q2 has the same values in each of its fields after the assignment. Line 7 dequeues the value 7 from q2 by removing the node following the dummy node from the linked list. Line 8 tests for equality between q1 and q2. This time however, q1.front = q2.front and q1.back = q2.back and “equal” is printed out, counter to our intuition.
In this particular example, the engineer has lost the ability to reason locally. How? Reasoning about Client (correctly predicting its behavior by simulating its execution) requires understanding Bounded_Queue’s representation and its implementation. Figure 10 illustrates this by extending the reasoning area to include one of the Queue’s implementations.

Figure 10 — Non-local Reasoning Required by Bounded_Queue

It is sometimes the case that components work as we would intuitively expect them to, e.g., the Bounded_Queue represented by the array. However, we want all components to work that way all the time, and furthermore we do not want the client programmer to have to examine each component’s implementation to see if it does or does not. At the heart of the problem exposed by this example is the implicit exportation of “:=” and “=” operators. Intuitively, we expect these operators to perform copying and equality testing in the abstract, i.e., copy and compare Queue’s abstract model, regardless of the type’s representation. In reality, they cannot hope to do this because that would require the compiler to automatically generate code to perform a “deep” copy and a “deep” comparison. These operations, if they are exported at all, should be left to the package designer and implementer and not to the compiler. This leads us to the next principle:

**Principle 7** — Export all types as limited private types.

Limited private types provide the same encapsulation as private types, but do not implicitly have “:=” and “=” operators. The only operations available for the exported type are those
that are explicitly exported by the package. The following is the new package specification which exports Queue as a limited private type.

Section 2.1.4 notes that Initialize should be implemented using a procedure instead of a function because an Initialize function would require the “:=” operator (which is not available for limited private types). We mention this again because these subtleties can easily trip up package designers. For example, in [Blum 92, page 301] a stack package is presented that exports a limited private type and an initialization function. Ada compilers are perfectly happy to compile packages designed like this. It is only when the client programmer tries to compile a client program containing a call to the initialize function (which must be on the right hand side of an assignment statement) that the Ada compiler will complain.

The use of limited private types has been recommended by [Booch 87, page 55], [Hibbard 83, page 42], and [Wallis 90, page 119] to name a few. Others use private types in their examples, e.g., [Caverly 86, page 187], [Feldman 92, page 493], and [Musser 89, page 47].

```ada
generic
package Queue_7 is
  type Queue is limited private;
  procedure Initialize (q: in out Queue);
  procedure Finalize (q: in out Queue);
  procedure Enqueue (q: in out Queue; x: in Integer);
  procedure Dequeue (q: in out Queue; x: out Integer);
  function Is_Empty (q: in Queue) return Boolean;
private
  type Queue_Rep;
  type Queue is access Queue_Rep;
end Queue_7;
```

Support for the properties is as follows:

- Correctness — Given the example above, we see that the client programmer does not know a priori if a component that exports a private type behaves according to intuition (as it does with the array representation) or not (as it does not with the linked list representation). Consequently, if a component exports private types instead of limited private types, then the client programmer must always examine the representation and the implementation of the component to reason confidently about its behavior. Principle 7 eliminates this insidious loss of the ability to reason within a local area around the client.

2.1.7 Data Movement Operation

One reason given in the literature [Naiditch 89, page 225] for choosing private types over limited private types is the availability of assignment. Since Principle 7 eliminates this option, we are faced with the question of how to move around values of variables declared as limited private types. Two possible alternatives for data movement are copying and swapping. For efficiency reasons discussed in [Harms 91], we choose Swap as our data movement operation. Others, such as [Booch 87, page 146] and [Uhl 90, page 96] provide a Copy operation. The designer of a component observing Principle 8, which
requires Swap for all exported types, might decide instead to provide a Copy operation. Note that programming with Swap instead of Copy leads the client programmer to use a slightly different programming style, the “swapping style” discussed in [Harms 91].

**Principle 8** — For each exported type T, export a data movement operation as follows:

```plaintext
procedure Swap (x1: in out T; x2: in out T);
```

Here is an example of our Queue package designed by following Principle 8:

```plaintext
generic
package Queue_8 is
    type Queue is limited private;
    procedure Initialize (q: in out Queue);
    procedure Finalize (q: in out Queue);
    procedure Swap (q1: in out Queue; q2: in out Queue);
    procedure Enqueue (q: in out Queue; x: in Integer);
    procedure Dequeue (q: in out Queue; x: out Integer);
    function Is_Empty (q: in Queue) return Boolean;
private
    type Queue_Rep;
    type Queue is access Queue_Rep;
end Queue_8;
```

Swap must have two parameters (e.g., x1 and x2) of the same type. The result of the operation is that the values of the supplied actual parameters are swapped, i.e., x1 gets x2’s original value and x2 gets x1’s original value. An implementation for Queue_8’s Swap is:

```plaintext
procedure Swap (q1: in out Queue; q2: in out Queue)
    is
        temp: Queue;
    begin
        temp := q1;
        q1 := q2;
        q2 := temp;
    end Swap;
```

Support for the properties is as follows:

- Composability — Generic composition by type requires some kind of data movement operation. Principle 8 guarantees that we have a data movement operation. (Generic composition is discussed further in Section 2.1.11 and Section 2.1.12.)
• Reusability — The Swap operation can always be implemented to work in constant
time [Harms 91]. In general, this is not possible for Copy operations that copy
abstract values. Therefore, with Swap the efficiency of the data movement operation
is always acceptable (indeed, optimal).

2.1.8 Operations as Procedures

[Hollingsworth 91] suggests some compelling reasons for layering some operations on top
of a component instead of directly exporting them. Since limited private types (unlike
private types) do not implicitly have an equality testing operation, one candidate for
layering might be an Are_Equal operation. Our unit of modularity is the generic package
(see Principle 1 in Section 2.1.1), so normally we would layer an operation by using a
generic package. The details of how to do this are left to Section 3.1. For this example
only, we use Ada’s generic function construct to layer an Are_Equal operation on top of
Queue_8. The generic function specification is as follows:

generic

    type Queue is limited private;
    with procedure Initialize (q: in out Queue);
    with procedure Finalize (q: in out Queue);
    with procedure Swap (q1: in out Queue; q2: in out Queue);
    with procedure Enqueue (q: in out Queue; x: in Integer);
    with procedure Dequeue (q: in out Queue; x: out Integer);
    with function Is_Empty (q: in Queue) return Boolean;

function Are_Equal (q1: in out Queue; q2: in out Queue) return Boolean;

A possible implementation for Are_Equal would seem to be:

function Are_Equal (q1: in out Queue; q2: in out Queue) return Boolean is
    equal : Boolean;
    x1, x2: Integer;
    catalyst_1, catalyst_2: Queue;
begin
    Initialize (catalyst_1);
    Initialize (catalyst_2);
equal := true;

while equal and not Is_Empty (q1) and not Is_Empty (q2)
loop
  Dequeue (q1, x1);
  Dequeue (q2, x2);
  equal := equal and (x1 = x2);
  Enqueue (catalyst_2, x2);
  Enqueue (catalyst_1, x1);
end loop;

if not Is_Empty (q1) then
  equal := false;
  loop
    Dequeue (q1, x1);
    Enqueue (catalyst_1, x1);
    exit when Is_Empty (q1);
  end loop;
end if;

if not Is_Empty (q2) then
  equal := false;
  loop
    Dequeue (q2, x2);
    Enqueue (catalyst_2, x2);
    exit when Is_Empty (q2);
  end loop;
end if;

Swap (q1, catalyst_1);
Swap (q2, catalyst_2);

Finalize (catalyst_2);
Finalize (catalyst_1);

return equal;
end Are_Equal;

There is one big problem with this code: The compiler will not compile the above function because Ada requires all formal parameters of functions to be “in” mode only. The above function has been written with “in out” mode parameters. Another Ada requirement is that an operation’s formal “in” mode parameter cannot be supplied as an argument in a call to a procedure where the corresponding formal parameter is “in out” mode. Therefore, we cannot fix this problem by changing the parameter modes of q1 and q2 to be “in.” If we did, then again the compiler would not compile Are_Equal because it calls the procedures Dequeue and Swap whose formal parameters have “in out” mode.

There are two possible solutions to this problem: (1) do not layer any operations normally viewed as functions, but explicitly export them from the package; or (2) use the procedure construct for all layered operations, adding an additional parameter to the parameter list for those that would have been functions (the value that the function would have returned is
returned through this additional parameter). The first alternative is not attractive because package specifications could potentially grow to include a vast array of exported operations, some of which would probably be of little use to the general user of the package. How could the designer think of them all anyway? Furthermore, there is potential for improved productivity and better quality if layering is used, as is discussed in [Hollingsworth 91]. Therefore, we choose the second alternative, which gives us the following principle:

**Principle 9** — Export and import (through generic formal parameters) all operations as procedures. If an operation would normally be a function, add an additional formal parameter (of the type to be returned) to the parameter list, and return the value to the caller through this parameter.

Support for the properties is as follows:

- **Composability** — Choosing to implement an operation $P$ as a function can lead to composability problems when we layer $P$ on top of other operations, as Are_Equal demonstrates above. By choosing to implement $P$ as a function, we might preclude some particularly interesting (even necessary) implementations of operations that need to temporarily dismantle and then reconstruct the abstract values of their parameters in order to perform the desired computation, e.g., the Are_Equal operation. What is really needed is for functions to produce no net change in their parameters’ abstract values. Trying to enforce this by not allowing temporary changes in the representations unnecessarily restricts the software engineer from choosing some useful, even necessary, implementations.

- **Reusability** — By layering an operation whenever possible, instead of directly exporting it from the component, we achieve a higher level of reusability. This is because the layered operation can be used with any implementation of the component. For example, Are_Equal (made into a procedure) does not even need to be recompiled in order to work with an alternate implementation of the Queue package.

- **Understandability** — Principle 9 reduces the cognitive load through its requirement of uniformity, consistency, and by eliminating a dead-end design alternative.

Later (Section 2.1.16) we discuss alternative implementations (package bodies) for the same package specification (i.e., concept). Due to the nature of Ada packages, we must at least duplicate the package specification for each implementation and then modify it slightly to give it a unique name. To make it so the client programmer can easily unplug one implementation from the client program and plug in another (i.e., “plug-compatibility”), we want each duplicated package specification to export the exact same structural interface for its operations. When a component is first designed, the designer might not recognize the existence of alternative useful implementations. For a particular implementation of a component, it might make sense to design an operation using formal parameter modes “in” or “out.” But for alternative implementations which might use layering, this choice might lead to composability problems similar to those demonstrated by the Are_Equal example above. There are two possible solutions this problem: (1) forgo plug-compatible components and allow different package specifications (for the same concept) to export different structural interfaces (i.e., possibly changing “in” or “out” to “in out”); or (2)
require all formal parameters to have the mode “in out,” supporting all future alternative implementations.

Since we want to provide an environment that fosters the development of plug-compatible, alternative implementations for the same concept, we introduce the following principle:

**Principle 10** — Make all formal parameters have the mode “in out.” The only exception is for built-in scalars (e.g., Integer, Boolean, etc.) where only a value is needed by the called operation. In this case “in” mode is appropriate.

From another perspective, one can view Principle 10 as leading the way to reduced complexity, and improved uniformity in our Ada components. [Harms 90, page 83] describes the purpose of parameter passing modes in the language RESOLVE as follows: “The parameter mode describes how a parameter is used in the exchange of information between invoker and operation....” He goes on to say that “Parameter modes do not, however, describe the parameter passing mechanism used to exchange information between invoker and operation.” See also [Harms 91].

Ada has tried to combine these two different points of view with limited success. For example, the “in” mode is trying to capture the idea of “no net change” in the actual parameter’s abstract value, and at the same time is also trying to capture how information is exchanged under this mode between invoker and operation [Ada 83, page 107]. Principle 10 begins to decouple these two aspects of parameter passing in Ada as Harms has done in RESOLVE, by specifying the parameter passing mechanism. Later (Section 2.1.14) discusses the use of four abstract parameter modes, introduced by Harms, which specify how a parameter is used by the called operation.

Finally, we explain why the “in” mode exception for built-in scalars is allowed, and why we do not have symmetry and allow “out” mode for built-in scalars. There is no harm in allowing “in” mode for parameters of built-in scalar types whose abstract values are not supposed to be changed by an operation. Specifically, no aliasing can result from parameter passing with these types. Allowing “in” mode is simply a matter of convenience, as demonstrated by the following example. The for-loop here would not compile if we required Enqueue’s Integer parameter to be “in out” mode:

```ada
for j in 1..10 loop
   Enqueue (q, j);
end loop;
```

This is because the loop parameter (i.e., j) is considered to be a constant within the loop body. Therefore, “the loop parameter must not be given as an ‘out’ or ‘in out’ parameter of a procedure or entry call statement” [Ada 83, page 99]. To achieve the above loop’s intended functionality, the programmer would have to introduce a locally declared temporary variable, as in the following:

```ada
for j in 1..10 loop
   i := j;
   Enqueue (q, i);
```

It is also convenient to be able to use constants of built-in types as arguments to procedures, as in:

\[
\text{Enqueue (q, 15);}
\]

As for “out” mode only for scalars, the only benefit to be gained is that the actual
parameter’s value might not be copied to the formal parameter when the procedure is
invoked (a possible minor performance improvement [Ada 83, page 107]). However, the
use of “out” mode (and “in” mode for that matter) introduces the potential for composability
problems when layering as discussed above. As it turns out, the composability problem in
this case is only a minor irritant. This is because the layered operation can “funnel” the
values from “in” mode parameters, and to “out” mode parameters, through locally declared
temporary variables, and use the temporary variables when calling other operations with
mode “in out.” Since allowing “in” mode for scalars does improve convenience but “out”
mode only provides a possible minor performance improvement, we allow the exception
for “in” mode for scalars but not for “out” mode.

Support for the properties is as follows:

- Composability and Reusability — See support for Principle 9, above.

- Understandability — We have decoupled “how a parameter is passed” from “how a
  parameter is used,” and thus reduced the cognitive load associated with Ada’s
  parameter modes.

Our Queue package now looks like:

```ada
generic
package Queue_9 is
  type Queue is limited private;
  procedure Initialize (q: in out Queue);
  procedure Finalize (q: in out Queue);
  procedure Swap (q1: in out Queue; q2: in out Queue);
  procedure Enqueue (q: in out Queue; x: in Integer);
  procedure Dequeue (q: in out Queue; x: in out Integer);
  procedure Test_If_Empty (q: in out Queue; empty: in out Boolean);

private
  type Queue_Rep;
  type Queue is access Queue_Rep;
end Queue_9;
```
The interface for the layered operation that tests for equality between two Queues becomes:

```plaintext
procedure Test_If_Equal (  
    q1: in out Queue;  
    q2: in out Queue;  
    equal: in out Boolean);
```

2.1.9 Package Finalization

This section is concerned with finalization of all items declared internal to the package body, e.g., variables and other package instances. A good example of this is a component that performs storage management by maintaining a free list. A free list is a linked list of dynamically allocated “nodes” that are not currently needed by the component. [Booch 83] and [Musser 89], for example, have suggested this practice in Ada components. One important advantage in doing so is the ability to implement a constant time Finalize operation for linked structures of length N (see [Weizenbaum 63], [Weide 86], and [Hollingsworth 92]). One might expect that the performance of such an operation would have to be linear in the length of the linked structure (see Section 2.1.3 for an example of this), but it does not if the component maintains its own free list.

The problem with maintaining a free list is that there is potential for a storage leak to occur. This is because Ada allows the instantiation of packages anywhere a basic declaration is allowed. For example, a package can be instantiated in the declarative part of a procedure, and when control leaves the procedure’s scope, all access to the instance is lost. This is the point where a storage leak could occur if the component is maintaining a free list. The following principle addresses this problem:

**Principle 11** — Export a procedure (called Finalize_Package) that finalizes all items declared internal to the package body that need finalization, e.g., variables, other package instances, etc.

Here is our queue package example augmented to follow this principle:

```plaintext
generic  
package Queue_10 is  
    procedure Finalize_Package;  
    type Queue is limited private;  
    procedure Initialize (  
        q: in out Queue);  
    procedure Finalize (  
        q: in out Queue);  
    procedure Swap (  
        q1: in out Queue;  
        q2: in out Queue);  
    procedure Enqueue (  
        q: in out Queue;
```
procedure Dequeue (q: in out Queue; x: in out Integer);
procedure Test_If_Empty (q: in out Queue; empty: in out Boolean);

private
  type Queue_Rep;
  type Queue is access Queue_Rep;
end Queue_10;

Support for the properties is as follows:

- Reusability — Client programmers are more apt to reuse components that make careful use of resources. Components that export Finalize_Package fit into that category by allowing the client to return resources to the system when the package instance is no longer needed.

In the following section Queue’s package body and a client of Queue are provided. The package body illustrates the maintenance of the free list, the implementation of the Finalize_Package operation, and an implementation of a constant-time Finalize operation for type Queue.

2.1.10 Package Initialization

Ada supports package initialization by permitting the package designer to provide a sequence of statements in the package body. These statements are executed at instantiation time. We could make use of this capability to initialize all items declared in the package body, but we won’t. Ada does not support package finalization, so we have to export the Finalize_Package operation. For the sake of uniformity and consistency in the package interface, we offer the following principle:

Principle 12 — Export a procedure (called Initialize_Package) that initializes all items declared internal to the package body that need initialization, e.g., variables, other package instances, etc.

Here is our queue package example augmented to follow this principle:

```ada
generic
package Queue_11 is
  procedure Initialize_Package;
  procedure Finalize_Package;
  type Queue is limited private;
  procedure Initialize (q: in out Queue);
```
procedure Finalize (  
q:  in out Queue);

procedure Swap (  
q1:  in out Queue;  
q2:  in out Queue);

procedure Enqueue (  
q:  in out Queue;  
x:  in Integer);

procedure Dequeue (  
q:  in out Queue;  
x:  in out Integer);

procedure Test_If_Empty (  
q:  in out Queue;  
empty:  in out Boolean);

private

type Queue_Rep;

type Queue is access Queue_Rep;

end Queue_11;

Here is the package body:

with Unchecked_Deallocation;

package body Queue_11 is

-------------------------------------------------------
--- Declarations --------------------------------------
-------------------------------------------------------

type Node_Rep;

type Node is access Node_Rep;

type Node_Rep is record  
  label: Integer;  
  next: Node;
end record;

type Queue_Rep is record  
  front: Node;  
  back: Node;
end record;

free_list: Node;

-------------------------------------------------------
--- Local Operations ----------------------------------
-------------------------------------------------------

procedure Free is new  
  Unchecked_Deallocation (Node_Rep, Node);

procedure Free is new
   Unchecked_Deallocation (Queue_Rep, Queue);

procedure Get_Node (  
   new_node: in out Node  
) is
begin
   if free_list /= null then
      new_node := free_list;
      free_list := free_list.next;
   else
      new_node := new Node_Rep;
   end if;
   new_node.next := null;
end Get_Node;

procedure Return_Node (  
   old_node: in out Node  
) is
begin
   old_node.next := free_list;
   free_list := old_node;
end Return_Node;

--- Exported Operations -----------------------------------

procedure Initialize_Package is
begin
   free_list := null;
end Initialize_Package;

procedure Finalize_Package is  
   old_node: Node;
begin
   while free_list /= null loop
      old_node := free_list;
      free_list := free_list.next;
      Free (old_node);
   end loop;
end Finalize_Package;

procedure Initialize (  
   q: in out Queue  
)
is
begin
q := new Queue_Rep;
Get_Node (q.front);
q.back := q.front;
end Initialize;

-------------------------------------------------------
procedure Finalize ( q: in out Queue ) is
begin
q.back.next := free_list;
free_list := q.front;
Free (q);
end Finalize;

-------------------------------------------------------
procedure Swap ( q1: in out Queue; q2: in out Queue ) is
begin
temp: Queue;
temp := q1;
q1 := q2;
q2 := temp;
end Swap;

-------------------------------------------------------
procedure Enqueue ( q: in out Queue; x: in Integer ) is
begin
Get_Node (q.back.next);
q.back := q.back.next;
q.back.label := x;
end Enqueue;

-------------------------------------------------------
procedure Dequeue ( q: in out Queue; x: in out Integer ) is
begin
old_front: Node;
old_front := q.front;
q.front := q.front.next;
x := q.front.label;
procedure Test_If_Empty (  
    q: in out Queue;  
    empty: in out Boolean  
  ) is  
begin  
  empty := (q.front = q.back);  
end Test_If_Empty;  

end Queue_11;  

The following is a client of Queue_11:  

procedure Queue_11_Client is  
  package Queue_Facility is new Queue_11;  
  q1, q2: Queue_Facility.Queue;  
begin  
  Queue_Facility.Initialize_Package;  
  Queue_Facility.Initialize (q1);  
  Queue_Facility.Initialize (q2);  
  Queue_Facility.Enqueue (q1, 5);  
  Queue_Facility.Enqueue (q2, 3);  
  Queue_Facility.Finalize (q2);  
  Queue_Facility.Finalize (q1);  
  Queue_Facility.Finalize_Package;  
end Queue_11_Client;  

2.1.11 Conceptual Type Parameters  

For the next two sections, we turn our attention to the problem of component composition.  
Examining Queue_11 we see that it exports a queue of type integer. We cannot use it for  
other types such as float or boolean. In a language such as Modula-2, the obvious solution  
is to make a physical copy of the component and systematically substitute everywhere the  
desired type for integer. In Ada we can achieve reuse a little more elegantly by using the  
generic construct: 

Principle 13 — Parameterize the component by each ADT that it  
manipulates but does not export.
Support for the properties is as follows:

- **Composability** — The component is parameterized by the types it manipulates and therefore can be instantiated in the client with the types that the client needs.

- **Reusability** — Increased reuse of code is achieved because each instance uses the same code “template.” This eliminates the kind of duplication of code that arises in languages such as Modula-2.

In the generic package specification for Queue that follows, notice that we have chosen for illustration to import the type as a private type. This follows [Hibbard 83, page 38], [St. Dennis 86, page 84], [Booch 87, page 146], [Watt 87, page 357], [Luckham 90, page 198], and [Feldman 92, page 619], to name a few.

```pascal
generic
type Item is private;
package Queue_12 is
  procedure Initialize_Package;
  procedure Finalize_Package;
  type Queue is limited private;
  procedure Initialize (q: in out Queue);
  procedure Finalize (q: in out Queue);
  procedure Swap (q1: in out Queue; q2: in out Queue);
  procedure Enqueue (q: in out Queue; x: in out Item);
  procedure Dequeue (q: in out Queue; x: in out Item);
  procedure Test_If_Empty (q: in out Queue;
                           empty: in out Boolean);
private
  type Queue_Rep;
  type Queue is access Queue_Rep;
end Queue_12;
```

The only difference between Queue_12’s package body and Queue_11’s package body (provided in the previous section) appears in Enqueue’s and Dequeue’s structural interface. Perhaps surprisingly, no changes are necessary to any of the operations’ implementations.

### 2.1.12 Conceptual Type’s Standard Operations

As noted at the end of the last section, no changes are necessary to any of implementations of Queue_11’s operations when it is transformed into Queue_12; only the structural interfaces of Enqueue and Dequeue change. This indicates that we are on the right track.
However, there are still a few problems that need to be addressed. For example, if we look at the implementation of Enqueue, we see that it makes use of assignment to get the item into the queue:

```plaintext
procedure Enqueue (  
  q: in out Queue;  
  x: in out Item  
) is  
begin  
  Get_Node (q.back.next);  
  q.back := q.back.next;  
  q.back.label := x;  
end Enqueue;
```

The use of assignment should raise a red flag. Our components do not have assignment, and furthermore, our components export limited private types. Queue_12 expects a private type. There is a mismatch between the types that our components export and the types that components like Queue_12 can import. This means that components (as they currently stand) cannot readily be composed with each other. Others (e.g., [Hibbard 83, page 38] and [Booch 87, page 146]) also have this problem but do not face it head-on. To solve it we introduce the following principle:

**Principle 14** — For each type parameter to a generic package, import the type as limited private and import the type’s standard operations: Initialize, Finalize and Swap.

The generic package specification for Queue that follows is similar to packages found in [Muralidharan 90, page 517], [Weide 91, page 47], and [Hollingsworth 92, page 90].

```plaintext
generic  
  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);  
package Queue_13 is  
  procedure Initialize_Package;  
  procedure Finalize_Package;  
  type Queue is limited private;  
  procedure Initialize (  
    q: in out Queue);  
  procedure Finalize (  
    q: in out Queue);  
  procedure Swap (  
    q1: in out Queue;
```
q2: in out Queue);

procedure Enqueue (  
q: in out Queue;  
x: in out Item);

procedure Dequeue (  
q: in out Queue;  
x: in out Item);

procedure Test_If_Empty (  
q: in out Queue;  
empty: in out Boolean);

private

  type Queue_Rep;
  type Queue is access Queue_Rep;

end Queue_13;

The implementations of Get_Node, Return_Node, Enqueue and Dequeue change slightly because of Principle 14:

procedure Get_Node (  
  new_node: in out Node  
) is
begin
  if free_list /= null then
    new_node := free_list;
    free_list := free_list.next;
  else
    new_node := new Node_Rep;
  end if;
  new_node.next := null;
  Initialize (new_node.label);
end Get_Node;

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

procedure Return_Node (  
  old_node: in out Node  
) is
begin
  Finalize (old_node.label);
  old_node.next := free_list;
  free_list := old_node;
end Return_Node;

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

procedure Enqueue (  
  q: in out Queue;  
x: in out Item  
) is
begin
  Get_Node (q.back.next);
  q.back := q.back.next;
  Swap (q.back.label, x);
end Enqueue;

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

procedure Dequeue (q: in out Queue; x: in out Item)
begin
  old_front: Node;

  old_front := q.front;
  q.front := q.front.next;
  Swap (q.front.label, x);
  Return_Node (old_front);
end Dequeue;

Notice that the implementations demonstrate why we need to import all three standard operations. Get_Node now takes responsibility for initializing new_node.label (which is of type Item), when a new node comes into existence. Enqueue and Dequeue both use Swap to move the value into and out of the queue. Finally, Return_Node finalizes old_node.label prior to returning the old node to the free list. Without Initialize for type Item, Enqueue would be swapping an uninitialized Item back to the client. Without Finalize for type Item, a storage leak might be introduced because the item in old_node.label (of Return_Node) could not be finalized.

Support for the properties is as follows:

- Composability — Our components can be composed with each other. Note that this raises a question about composition with the built-in types (e.g., Integer, Boolean, etc.), because those types do not have Initialize, Finalize and Swap operations. Section 4.1 addresses this problem.

- Correctness — The component implementer has all the necessary information concerning the imported type for reasoning locally. All the implementer needs to know about type Item is that it is limited private and has the standard operations Initialize, Finalize and Swap.

- Reusability — Our components are more reusable because of their support for composability. Also, they are more reusable because of their ability to handle initialization and finalization of imported types in a uniform manner (see Get_Node and Return_Node above), preventing uninitialized variables and storage leaks.

- Understandability — The uniformity and consistency required by Principle 14 reduce the cognitive load. We should always import a type and its Initialize, Finalize and Swap operations, and always instantiate a generic component with a type and its Initialize, Finalize and Swap operations.

2.1.13 Fully Parameterized Components

In this section we introduce a principle that supports composability and reusability. It is based on work by Sitaraman, who notes that one aspect of component reusability is the ability of the client programmer to be able to tune the components used by the client so that
they meet performance needs; see [Sitaraman 90]. If the engineer cannot tune the component, and the component does not meet the client program’s performance requirements, then the engineer probably will be forced to redesign the component (creating a functionally similar component with different performance characteristics). Obviously, this is not the kind of reuse that we are seeking. To address this problem, we propose the following principle based on Sitaraman’s work:

**Principle 15** — Given a component C that needs to use some other components C1, ..., Cn in its implementation, fully parameterize component C with each of Ci’s exported types and each of its operations (except Ci’s Initialize_Package and Finalize_Package). List Ci’s types and operations in the same order and using the same names as they have in Ci’s package specification.

Up to now, our queue component has been implemented by a direct implementation using Ada’s records and pointers. It has not been implemented by using any other components. We have an abstraction for singly linked lists, called One_Way_List, that can be used in place of the direct implementation currently used by the queue component. The following is the queue component fully parameterized by the type and operations provided by One_Way_List (see Appendix A for One_Way_List’s package specification):

```
generic
  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);

type List is limited private;
  with procedure Initialize ( s: in out List);
  with procedure Finalize ( s: in out List);
  with procedure Swap ( s1: in out List; s2: in out List);
  with procedure Move_To_Start ( s: in out List);
  with procedure Move_To_Finish ( s: in out List);
  with procedure Advance ( s: in out List);
  with procedure Add_Right ( s: in out List; x: in out Item);
  with procedure Remove_Right ( s: in out List; x: in out Item);
```
The following is the package body for Queue_14. It is hard to tell whether it actually performs the same function as previous implementations of queue (it really does), because we do not yet have a complete functional description of One_Way_List’s operations (see Appendix A). But that is not the point of showing Queue_14’s package body. What should be noticed is: (1) the implementation is based on One_Way_List’s abstraction, not on pointers and nodes, and there is no explicit allocation or deallocation of heap storage; (2) the implementation of Initialize, Finalize and Swap is by simple call-through to List’s Initialize, Finalize and Swap; (3) the implementations of Enqueue and Dequeue consist of two procedure calls each; (4) there is no need to maintain a free list, and therefore the implementation of Initialize_Package and Finalize_Package is trivial. (A free list might be maintained by One_Way_List, depending on the implementation selected by the client programmer for One_Way_List at the time Queue_14 is instantiated.)
--- Exported Operations -----------------------------------

procedure Initialize_Package is
begin
  null;
end Initialize_Package;

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

procedure Finalize_Package is
begin
  null;
end Finalize_Package;

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

procedure Initialize (q: in out Queue)
is
begin
  Initialize (q.rep);
end Initialize;

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

procedure Finalize (q: in out Queue)
is
begin
  Finalize (q.rep);
end Finalize;

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

procedure Swap (q1: in out Queue;
                q2: in out Queue)
is
begin
  Swap (q1.rep, q2.rep);
end Swap;

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

procedure Enqueue (q: in out Queue;
                    x: in out Item)
is
begin
  Move_To_Finish (q.rep);
  Add_Right (q.rep, x);
end Enqueue;
procedure Dequeue (q: in out Queue; x: in out Item) is
begin
    Move_To_Start (q.rep);
    Remove_Right (q.rep, x);
end Dequeue;

procedure Test_If_Empty (q: in out Queue; empty: in out Boolean) is
begin
    Test_If_At_Start (q.rep, empty);
    if empty then
        Test_If_At_Finish (q.rep, empty);
    end if;
end Test_If_Empty;

end Queue_14;

The advantage of this approach (i.e., building the Queue component on top of One_Way_List) should be quite clear: simplicity. When queue is built directly using Ada’s records and pointers, there is nothing a client programmer can do to tune it for a particular client program, short of making changes to the package body. Now the client programmer can supply a different implementation for One_Way_List when instantiating Queue_14. This permits him/her to choose a One_Way_List implementation that best suits the needs of the client program. One way in which two One_Way_List packages might differ is, for instance, in their handling of free storage.

Support for the properties is as follows:

• Composability — Given component C, and its dependency on components C1, ..., Cn, we have made C’s dependence on each Ci’s abstraction explicit. More importantly, we have effectively decoupled C’s implementation from each Ci’s implementation. With component C being fully parameterized, the client programmer is able to compose C with any of Ci’s implementations at instantiation time, and do so without making code modifications to C or to Ci, or in principle even recompiling them.

• Reusability — The components are performance-tunable through component composition, thus making them more reusable.
The cumulative effect of Principles 1 through 15 leaves us with components that are composable, support reasoning in a local area, are relatively reusable and are relatively understandable. But we are not quite done. Recall from Section 1.3.1 that the definition of correctness includes support for abstract reasoning about a component’s behavior. So far, none of our principles address this issue. Therefore, we introduce the following principle which supports correctness, reusability, and understandability.

**Principle 16** — Formally specify the behavior of each component using a mathematical specification language.

For our components we choose a mathematical model-based approach to formal specifications. Other approaches, e.g., algebraic specifications, could also be used. In the model-based approach, abstract component behavior is explained by mathematically modeling the program objects with mathematical objects. The specification language used is derived from RESOLVE (REusable SOftware Language with Verifiability and Efficiency); see [Hegazy 89], [Harms 90], [Sitaraman 90], and [Weide 91] for more details about RESOLVE. For our purposes, the RESOLVE specification language permits us to do the following: to model the component’s types based on existing mathematical theories; to supply preconditions and postconditions for the component’s operations (in terms of the type’s mathematical model); to specify the conventions observed by the component’s implementation (see Section 2.2.3); and to specify the correspondence between the type’s mathematical model and its actual representation (see Section 2.2.3).

The following is our Queue example augmented by RESOLVE-style formal specifications. The formal specifications appear as “formal comments.” They are introduced by “--!” and are embedded in the Ada package specification.

```ada
--! concept Queue_Template

generic

--! conceptual context

--!

uses

--!

STRING_THEORY_TEMPLATE

--!

conceptual parameters

--!

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

uses
One_Way_List_Template

realization parameters

facility One_Way_List_Facility is
One_Way_List_Template (Item)

type List is limited private;
with procedure Initialize (s: in out List);
with procedure Finalize (s: in out List);
with procedure Swap (s1: in out List;
                   s2: in out List);
with procedure Move_To_Start (s: in out List);
with procedure Move_To_Finish (s: in out List);
with procedure Advance (s: in out List);
with procedure Add_Right (s: in out List;
x: in out Item);
with procedure Remove_Right (s: in out List;
x: in out Item);
with procedure Swap_Rights (s1: in out List;
s2: in out List);
with procedure Test_If_At_Start (s: in out List;
at_start: in out Boolean);
with procedure Test_If_At_Finish (s: in out List;
at_finish: in out Boolean);

package Queue_15 is

interface

procedure Initialize_Package;
procedure Finalize_Package;
type Queue is modeled by STRING
exemplar q

--!

initialization
ensures "q = EMPTY"
--!

type Queue is limited private;
procedure Initialize (q: in out Queue);
procedure Finalize (q: in out Queue);
procedure Swap (q1: in out Queue;
q2: in out Queue);

procedure Enqueue (q: in out Queue;
x: in out Item)
--! alters
--! consumes

ensures "q = APPEND (#q, #x)"

procedure Dequeue (q: in out Queue;
x: in out Item)
--! alters
--! produces

requires "q /= EMPTY"

ensures "PREPEND (q, x) = #q"

procedure Test_If_Empty (q: in out Queue;
empty: in out Boolean)
--! preserves
--! produces

ensures "empty iff (q = EMPTY)"

private

type Queue is record
    rep: List;
end record;
end Queue_15;
--! end Queue_Template

Explanation of the specifications in the generic parameter section. The first thing to note is that there are two parts to the Ada generic parameter section: the conceptual context part and the realization context part. The keyword "conceptual context" signals to the engineer that this part of the specification contains the items needed to explain Queue's abstract behavior. For Queues, the type Item and the math facility STRING_THEORY are required. Note that in RESOLVE, mathematical theories are reusable theory components. For our specification of Queue we need the STRING_THEORY_TEMPLATE instantiated with the math model of the imported type Item. For a formal definition of a basic STRING_THEORY_TEMPLATE, see [Weide 91, page 17]. The instance is known from
now on in the specification as STRING_THEORY, and it exports a type called STRING and some mathematical functions, such as EMPTY, APPEND, and PREPEND.

The keyword “realization context” signals that this part contains the items needed to implement Queue. For Queues, the One_Way_List_Facility (i.e., One_Way_List_Template instantiated with type Item) is needed. Note that the Ada code (that actually does the importing) immediately follows each specification.

Explanation of the type specification. In the statement “type Queue is modeled by STRING,” STRING is a short (unambiguous) name for STRING_THEORY.STRING, i.e., type Queue is modeled by the type STRING exported from the math facility STRING_THEORY. The type STRING is a string of (math model of) items because STRING_THEORY was created by instantiating STRING_THEORY_TEMPLATE with type (math model of) Item. Therefore, the statement “type Queue is modeled by STRING” should be interpreted as “type Queue is modeled as a string of items.” The next statement, “exemplar q,” names a variable q, of type Queue, which can be used in the remainder of Queue’s type definition. The initialization statement’s “ensures ‘q = EMPTY’” clause describes the initial value of variables of type Queue. For Queues, the initial value is the empty string; EMPTY is a constant exported by STRING_THEORY that denotes a string containing no items.

Explanation of the operations’ specifications.

• The specification for Initialize appears in the type definition’s “initialization” clause. It is the job of Initialize to ensure (i.e., guarantee) that its Queue parameter is initialized when it returns to the invoker, i.e., that it has a value satisfying initialization’s ensures clause.

• Note that Queue_15 has no Finalize specification (if it did, it too would appear in the type definition) because conceptually (in Queue_15’s case) there is nothing for Finalize to do. This does not mean that Finalize necessarily does nothing. It may need to release held resources (e.g., heap storage), but since these resources are not conceptual details, they are not mentioned in the specification. This procedure is essentially a “hook” for the implementation to release resources.

• Swap’s specification is not mentioned either, only because it is the same no matter what the type is (i.e., it swaps the values of its two parameters).

• Enqueue must ensure that when it is finished the value of the new q (i.e., the value of q at the termination of the call to Enqueue) equals the old q (i.e., #q, the value of q at the beginning of the call) with the value of old x appended to the right end of the string q. Note that the formal parameters have the parameter modes describing how they are to be used by Enqueue ([Harms 90, Harms 91]), i.e., parameter q has “alters” mode and parameter x has “consumes” mode. Alters mode indicates that useful information is passed to Enqueue (i.e., the value of q at the outset) and useful information is passed back to the invoker of Enqueue (i.e., the value of q at the end of the call). Consumes mode indicates that useful information is passed to Enqueue, and an initial value is passed back.

• Dequeue must ensure that x’s new value, if prepended to q’s new value, equals q’s old value. The “produces” mode for parameter x indicates that useful information is passed back to the invoker, and the information originally in x (at the time of the call) is irrelevant. Dequeue also has a precondition which is stated in the “requires” clause. For Dequeue to guarantee its ensures clause, it requires that q not be empty at
the time of the call. If q should happen to be empty at the time of the call, then
Dequeue guarantees nothing.

- Test_If_Empty must ensure that parameter empty is true if and only if q’s value is a
string containing no items. The “preserves” mode for parameter q indicates that
useful information is passed to Test_If_Empty, and also indicates that Test_If_Empty
guarantees that q’s conceptual value upon return will be the same as it was at the
beginning of the call.

- An implication of following Principle 16 (using RESOLVE/Ada formal comments) is
that each operation’s formal parameter appears on a separate line and consists of the
parameter name, Ada mode, type, and its abstract parameter mode appearing as a
formal comment. For example, here is Enqueue’s procedure header:

```ada
procedure Enqueue (q: in out Queue; x: in out Item

The mode is not listed for Initialize, Finalize, and Swap, or for generic formal
procedure parameters that are part of a facility being imported. This is purely for
brevity; these modes are evident without being stated explicitly in the specification.

Support for the properties is as follows:

- Correctness — The client programmer uses the abstraction presented in the formal
specification to reason about the component. Without the formal specification, the
client programmer has no other choice but to “pry” open the component to see how it
is implemented in order to determine its functionality. In other words, the client
programmer needs to see more than the fact that it is implemented using
One_Way_List (that can already be seen). He/she needs to see how One_Way_List is
being used, and if One_Way_List is not formally specified, then he/she will also need
to see how it is implemented, and so forth. This defeats our desire to reason locally
about components. Furthermore, with a formal specification of the component’s
behavior, we now have the opportunity to verify (either formally or rigorously
informally) that the implementation meets the specification. Without the specification,
the implementation “does what it does!”

- Reusability — Components that are formally specified (i.e., in a language other than
that of the code itself) are more likely to present an unambiguous description of their
functionality. Having this formal description available tends to make components
more reusable because it counteracts the attitude that “if I have to figure out what it
does by looking at the code, I would rather build my own.” Furthermore, one can
build plug-compatible implementations for a (reused) specification.

- Understandability — This is supported because client programmers are able to reason
at an abstract level about the component instead of being forced to reason at the
concrete level.
Because of Principle 1 (Section 2.1.1) and Principle 16 (Section 2.1.14) a component’s specification is physically and logically separated from its implementation and the specification includes implementation independent formal comments describing the component’s behavior. Recall that one of the benefits of this arrangement is that multiple implementations can be created for the same concept (Section 1.3.2). A benefit of this is that we can achieve a higher level of reusability by reusing the concept.

It is true that in Ada we can create multiple implementations for the same concept, e.g., for Queue_Template we have created Queue_15 based on One_Way_List_Template (Section 2.1.14), and in Section 4.3.1 we introduce Queue_16 based on One_Way.Nilpotent_Template. This can be represented by the diagram in Figure 11:

![Diagram](image)

**Figure 11 — Multiple Implementations of Queue_Template**

However, due to the nature of Ada packages, we must at least duplicate the package specification for each implementation and edit it to give it a unique name. So for Ada, it might be more realistic to represent multiple implementations for a single concept in diagram such as the one in Figure 12. (Note: There are proposals to change Ada to permit multiple package bodies for one package specification; see [Sitaraman 92]. However, at this time, these proposals are not being seriously considered for Ada 9X.)
When creating another implementation for the same concept, there might be other changes required to the package specification. There are ways to perform these changes so that the client programmer can “unplug” one implementation and “plug-in” another with no changes required to the executable code of the client; changes are only required in the declaration part. How to make these changes is covered by the following principle:

**Principle 17** — To support multiple implementations for the same concept that are plug compatible, make the package specification for each implementation differ only in the package name and the realization context.

Not only does this support reusability (because we use the same concept for different implementations) but it also supports composability. It does so because any component that is parameterized by a package instance of a concept (e.g., Queue_15 is parameterized by One_Way_List_Facility which is an instance of the concept One_Way_List_Template) can be composed with an instance of any implementation of that concept.

Queue_15 (Section 2.1.14) and Queue_16 (Section 4.3.1) are different implementations for Queue_Template concept. By comparing their package specifications one sees that they differ only in their name and in their realization context. Therefore, if a client program uses Queue_Template, a client programmer could change from one of these implementations to another simply by changing the client program’s declaration section. No changes would have to be made to the executable code.

2.1.16 **Naming and Formatting Conventions**

The introduction of formal comments into our components increases the number of different kinds of conceptual items in the component’s interface. For example, in
Queue_14 there are program variables, procedures, types and Ada keywords, etc. Added to these items in Queue_15 are mathematical theories, functions, variables and RESOLVE keywords, etc. Others have noted (e.g., [Booch 87] and [SPC 89]) that one way to reduce the conceptual load on all engineers that create and/or use a component is to follow conventions for naming and formatting. We agree with this point of view and introduce the following principle which supports understandability:

**Principle 18** — When designing and implementing a component, consistently follow a “reasonable” comprehensive set of naming and formatting conventions.

The conventions we have followed appear in Appendix C.

### 2.2 Client Implementation Principles

This section introduces principles that apply to client implementations (see Figure 13). A client is either a main program or another component. The principles introduced in this section apply to both. If the client is a main program, then Ada requires that it be a procedure. In this situation, the rectangle in Figure 13 labeled “Client Implementation” represents an Ada procedure. If the client is another component, then the rectangle represents the component’s package body. Additional principles specifically for component implementations are introduced in Section 2.3.

![Figure 13 — Client Implementation](image)

#### 2.2.1 Ada Constructs/Statements

The principles in Section 2.1 explicitly identify the Ada declarations/constructs (e.g., generic package, procedure, limited private type, etc.) to be used when designing a component interface. The following principle lists the Ada constructs/statements to be used
in implementations. The list mainly consists of the standard structured programming constructs. Some Ada constructs, e.g., “exception” and “function,” are ruled out because of principles introduced in Section 2.1. We omit the “task” construct because we do not have a formal system that supports reasoning about it, and the “goto” because it can reduce program understandability.

**Principle 19** — In a client, in addition to the Ada constructs permitted/prescribed by other principles, use only the following Ada constructs/statements:
- block
- case
- exit
- for loop
- if
- loop
- procedure call
- renames (procedures only)
- return
- while loop

### 2.2.2 Variable Declarations

Ada allows some variables to be declared using type constructors without first declaring a type. For example, Ada allows the following variable declaration:

```
a: array(1..80) of Boolean;
```

This declaration implicitly declares an abstract data type (i.e., a mapping from the Integers 1 to 80, to Boolean). Although this looks rather innocuous, the anonymous type does not conform to the component interface principles introduced in Section 2.1. For example, there is no explicit abstract model for describing the behavior of the allowed operations. Furthermore, it does not have the standard Initialize, Finalize or Swap operations. To prohibit an implicit type declaration through the use of type constructors in a variable declaration, we introduce the following principle:

**Principle 20** — Declare each variable using one of the built-in Ada scalar types (i.e., Boolean, Character, Float or Integer), or a type exported from a component that has been designed using the component interface principles of Section 2.1.

The astute reader will notice that the built-in types allowed by Principle 20 do not faithfully follow all of the component interface principles of Section 2.1. Since these types basically adhere to our intuitive notion of what they “should do” — and in principle this behavior could be formally specified [Ogden 91] — the most obvious problem with them is that they
do not have the operations Initialize, Finalize and Swap. The Built_In_Types package introduced in Section 4.1 provides Initialize, Finalize and Swap for each of the built-in scalar types. Section 2.2.4 discusses the use of the Built_In_Types package and its operations.

2.2.3 Eliminating Unwanted Aliasing

Implicitly included with each operation’s postcondition is a clause stating “and nothing else changes.” In other words, nothing other than the variables explicitly identified in the postcondition is affected by the execution of the operation. [Krone 88], [Harms 90], and [Ernst 91] have noted that when aliasing is present in a program it is hard to guarantee that nothing else changes. For example, suppose a client of an operation creates an alias (i.e., two variables, x1 and x2 both reference the same data object, which represents their common abstract value) and then calls the operation with x1. The operation’s postcondition should explicitly identify what happens to x1. If the operation changes the value referenced by x1, then the value of x2 will also be changed, and the condition “and nothing else changes” will not be true. If we permit aliasing of this kind in our programs, then locally reasoning about correctness is jeopardized. In this section we further discuss aliasing and introduce principles that eliminate unwanted aliasing from our programs.

There are two kinds of aliasing, explicit and implicit. Explicit aliasing occurs when we have pointer variables and copy addresses using the assignment operator. Principle 20 (see Section 2.2.1) eliminates this source of aliasing by requiring that all variables be declared using one of the built-in scalar types or a type exported from a component that has been designed using the principles of Section 2.1. This principle rules out the use of Ada’s access type, so explicit aliasing is eliminated. In Chapter 4, we introduce two components that encapsulate Ada’s access type constructor. There, we will see how aliasing can be controlled so that local reasoning about correctness is still possible.

Principle 20 also eliminates one form of implicit aliasing which can occur when using arrays. For example, in the following call to procedure P, if the indices i and j are equal, then P’s formal parameters name the same location in array “a”:

\[
P (a(i), a(j));
\]

Principle 20 prohibits variables from being declared using Ada’s built-in array constructor, so this form of aliasing is also eliminated.

The reader may be concerned that we have over-restricted the implementer by eliminating variables declared using Ada’s access type and array constructor, but we have not. Chapter 4 discusses encapsulations for both that are acceptable under the provisions of Principle 20.

The remainder of this section discusses two other forms of implicit aliasing that have not yet been ruled out and introduces two principles for doing so.

An alias to a variable can be created by using it more than once as an actual parameter in a procedure call. The following example illustrates how this can cause problems. Suppose a client of Queue decides to use the procedure Test_If_Equal (defined in Section 2.1.8; see also Section 3.1) in the following way:

\[
\text{Test}_\text{If}_\text{Equal} (q, q, \text{equal});
\]
Apparently this tests to see if q is equal to itself. Will this work? Only if q is empty does nothing bad happen. Any time q begins with an odd number of items, Dequeue’s requires clause is violated in the first loop. Any time q begins with an even number of items, it ends up with half as many items when Test If Equal returns; it is not preserved as demanded by the specification.

The problem is not that something is wrong with the implementation of Test If Equal; it is that an alias to q has been created, which we cannot allow. Therefore, we introduce the next principle:

**Principle 21** — In any procedure call, do not use any variable as an actual parameter more than once.

Another form of aliasing that must be prevented occurs in the following example. Suppose an integer variable x and procedure P are declared in the same scope, so that x is a global variable to P. If P is defined as follows, for example, then unexpected results can occur if P is called with x supplied as an actual parameter:

```plaintext
procedure P (y: in out Integer) is
begin
  x := x + 1;
  y := y + 2;
end P;
```

The problem is not in P’s implementation, it is that an alias to x has been created. To eliminate this form of aliasing, we introduce the following principle:

**Principle 22** — In any procedure call, do not supply as an actual parameter a variable that is globally known to the called procedure and used within the called procedure’s body.

We can be sure that no form of aliasing, explicit or implicit, can occur in a client program that follows all stated principles.

2.2.4 *Initialization, Finalization and Meeting Preconditions*

Client procedures must faithfully initialize and finalize package instances and declared variables at the correct time in order for the used components to operate correctly. The following principle prescribes the required behavior of a client:
**Principle 23** — In any client procedure, initialize and finalize all package instances and variables declared local in its scope. Specifically, do the following in this order:

1) Initialize all packages the procedure instantiates, in the order they are instantiated, by calling their respective Initialize_Package procedures.

2) Initialize all locally declared variables, in the order they are declared, by calling the appropriate Initialize procedures.

3) Do any other computations required to realize the specified behavior of the procedure.

4) Finalize all locally declared variables, in the reverse of the order they are declared, by calling the appropriate Finalize procedures.

5) Finalize all packages the procedure instantiates, in the reverse of the order they are instantiated, by calling their respective Finalize_Package procedures.

Note, only the main procedure may (but does not have to) omit steps 4 and 5. There is no point in wasting time finalizing variables and package instances if the program is about to terminate. Also, for steps 2 and 4, if the variable’s type is one of the built-in Ada scalar types (i.e., Boolean, Character, Float or Integer), then the client must “with” and “use” the Built_In_Types package (see Section 4.1) to obtain the built-in types’ Initialize, Finalize and Swap operations.

Of course client procedures must faithfully satisfy all preconditions of an operation, at the time the client calls the operation, in order for the operation to achieve its postcondition. The following obvious principle prescribes the required behavior of a client:

**Principle 24** — Call an operation only if its precondition is satisfied.

### 2.3 Component Implementation Principles

This section introduces the principles to be followed when implementing a layered component; refer to the rectangle labeled “Component Implementation” in Figure 14 below. These principles assume that the component interface principles introduced in Section 2.1 have been followed during the design of the package specification and that the client procedure implementation principles of Section 2.2 have been observed. Our examples continue to be based on the Queue package.
2.3.1 Private Part’s Type Declaration

This section discusses the declarations of a component’s exported types that appear in the private part of the package specification. There are two parts to this discussion: 1) the mechanics of how it is done; and 2) the source of the type used to represent the exported type.

The following principle describes how an exported type’s representation should be declared:

**Principle 25** — In the private part of component C’s generic package specification, declare each of C’s exported types as a record with one field (called rep) of the representation type.

Our example Queue component is fully parameterized, so the “rep” field type is List, which is a generic formal parameter. The following is the relevant portion of Queue_15’s package specification:

```haskell
generic
  ...
  type List is limited private;
  ...
package Queue_15 is
  ...
  type Queue is limited private;
  ...
private
  type Queue is record
    rep: List;
  end record;
end Queue_15;
```
An alternative to declaring the exported type as a record of one field is to use Ada’s derived types. This changes the private part of the package specification. For example, we might write:

```ada
generic
   ...
   type List is limited private;
   ...
package Queue_15 is
   ...
   type Queue is limited private;
   ...
private
   type Queue is new List;
end Queue_15;
```

We choose not to use this approach for the following reasons. First, if we use Ada’s overload resolution with a derived type, then in the package body an inconvenient type coercion is required everywhere where a call is made to a List operation whose name is the same as a Queue operation. For example, Queue’s Initialize operation calls List’s Initialize. When using the derived type approach the call is “Initialize (List (q)).” By declaring a record (see above), the call to List’s Initialize is “Initialize (q.rep).” (See Section 2.3.5 for the implementation of Initialize.)

Second, there are times when reasoning about an implementation is facilitated by introducing entities that are not explicitly implemented, but whose values can be derived from the values of entities that are implemented. Such unimplemented entities are known as adjunct variables [Krone 88]. Their (pseudo) computations appear only in formal comments in the package body (see Section 2.3.2). The record construct provides a convenient location for their declaration.

The discussion now turns to the source of the type used to represent a component’s exported type. One way to facilitate constructing components with leak-proof abstraction barriers is to use only leak-proof “building blocks” in the process. It is possible to build a component with a leak-proof abstraction barrier on top of other components that have leaky abstraction barriers. But this approach almost certainly requires the component implementer to understand specific implementation details of the lower-level leaky component, thwarting local certification of correctness and complicating understandability by increasing the cognitive load on the implementer. Our components are fully parameterized by the components they use and exactly which ones are supplied as actuals is not known until instantiation time. At instantiation time is too late for the component implementer to obtain the needed specific implementation details. Under these dynamic circumstances, the component implementer cannot possibly implement the component so that it presents a leak-proof abstraction barrier in all cases. Therefore, we introduce the following principle to control the source of the type used to represent a component’s exported type:
Principle 26 — For the representation type in the declaration of a component’s exported type, use one of the built-in Ada scalar types (i.e., Boolean, Character, Float or Integer), or a type exported from a component that has been designed using the component interface principles of Section 2.1.

Principle 26 supports correctness because the component implementer can reason locally about the components used in the implementation since they have been designed using the component interface principles of Section 2.1.

An implication of Principle 26 is that Ada’s built-in type constructors (i.e., record, array, and access types) cannot be directly used when declaring a representation type. We do not have to do without them. But before they can be used, each must be encapsulated in its own leak-proof abstraction that has been designed by following the component interface principles of Section 2.1. Chapter 4 discusses encapsulations of these built-in Ada type constructors.

2.3.2 Package Body Formal Comments

This section discusses the formal comments that appear in the package body as a direct result of Principle 16, introduced in Section 2.1.14.

The following is a list of the different kinds of formal comments that can appear in the package body:

- Conceptual context — This augments the conceptual context found in the package specification. It is used only when additional mathematical machinery is needed by the package body’s formal comments. These additional math theories and definitions appear here, rather than in the package specification, because they are needed by the package body’s formal comments but not by the package specification’s formal comments. The formal comments in Queue_15’s package body require no additional mathematical machinery. Note that Queue_15’s implementation (appearing below) is the same as Queue_14’s implementation (see Section 2.1.13).

- Conventions — This formal comment describes the implementation conventions that must be observed by each of the component’s exported operations in order that the representation of an exported type be interpretable as an abstract value of that type. For example, the linked list implementation of the Bounded_Queue, presented in Section 2.1.6, maintains a “dummy” node at the head of the linked list at all times. This convention (or package-level invariant) can be assumed by each of Bounded_Queue’s exported operations, and must be ensured by each of them in order for the component to operate correctly. Specifically, the conventions must be satisfied between all calls to the component’s exported operations. However, while an exported operation is executing, the conventions may temporarily be not satisfied. When the exported operation terminates, the conventions must be satisfied again. Thus, each exported operation (except Initialize) can depend on the conventions being satisfied when invoked, and each operation (except Finalize) must guarantee that they are again satisfied upon termination.
Furthermore, by specifying package-level conventions, the component implementer is able to reason locally about each exported operation. The implementer reasons using the operation’s requires and ensures clauses, and the module’s implementation conventions. Without the specification of the conventions, the implementer would have to understand the implementations of each of the other operations in order to confidently reason about the implementation of any particular exported operation. Queue_15 maintains no conventions.

- Correspondence — This formal comment specifies the correspondence between the representation’s mathematical model and the exported type’s mathematical model. With the correspondence the component implementer is able to logically convert a variable’s value from its representation’s mathematical model to the exported type’s mathematical model. For example, the correspondence for Queue_15 is: “q = q.rep.left \ o \ q.rep.right,” which states that Queue q (modeled by a string) can be obtained from q.rep (modeled by a tuple of two strings) by concatenating q.rep.left with q.rep.right.

- Specifications of local operations — If any local operations (known only within the package body) are declared they must be formally specified using requires and ensures clauses. Notice that implementation conventions need not hold before or after a local operation is executed. Queue_15’s package body does not have any local operations.

- Code to update adjunct variables — If any adjunct variables [Krone 88] are declared, then “adjunct code” must be supplied to update their values. The code appears as formal comments in the package body’s operations. Queue_15 has no adjunct variables. (Adjunct variables are used in the formal comments of an implementation of Queue_Template based on One_Way_Nilpotent_Template, see Section 4.3.2.)

The following is Queue_15 package body with formal comments added:

```plaintext
package body Queue_15 is

--- Declarations --------------------------------------

--!
type Queue
--!
correspondence
--!
"q = q.rep.left \ o \ q.rep.right"

--- Exported Operations -------------------------------

procedure Initialize_Package is
begin
  null;
end Initialize_Package;
```

procedure Finalize_Package is begin null; end Finalize_Package;

procedure Initialize ( q: in out Queue ) is begin Initialize (q.rep); end Initialize;

procedure Finalize ( q: in out Queue ) is begin Finalize (q.rep); end Finalize;

procedure Swap ( q1: in out Queue; q2: in out Queue ) is begin Swap (q1.rep, q2.rep); end Swap;

procedure Enqueue ( q: in out Queue; x: in out Item ) is begin Move_To_Finish (q.rep); Add_Right (q.rep, x); end Enqueue;

procedure Dequeue ( q: in out Queue; x: in out Item ) is begin Move_To_Start (q.rep); Remove_Right (q.rep, x); end Dequeue;
procedure Test_If_Empty (  
  q: in out Queue;  
  empty: in out Boolean  
) is  
begin  
  Test_If_At_Start (q.rep, empty);  
  if empty then  
    Test_If_At_Finish (q.rep, empty);  
  end if;  
end Test_If_Empty;

Abstractly, One_Way_List_Template maintains the items in the list in two strings called "left" and "right." Commonly, the separation provided by left and right is referred to as the "fence." Another possible implementation for Queue_15, also based on One_Way_List_Template, is to keep the List's fence at one end of the List. This is a convention. Currently Queue_15's implementation does not have a convention; therefore, Enqueue must first position the fence at the end of the list before adding an item and Dequeue must first position the fence at the beginning of the list before removing an item.

To illustrate the use of conventions, we provide another implementation of Queue_15 which maintains the fence at the start (front) of the List. This convention stated in a formal comment is "q.rep.left = EMPTY," i.e., the fence is always at the start. This changes the implementation for Enqueue, Dequeue and Test_If_Empty. Enqueue moves the fence to the finish, adds the item to the List, and then moves the fence back to the start. When there was not a convention, Enqueue did not have to move the fence back to the start when it was finished. Dequeue can now depend on the fence being at the start, so it does not have to move the fence before removing the front item. When there was not a convention, it could not depend on the location of the fence, and had to move it to the start prior to the removal. Notice that removing an item from the start of the List leaves the fence at the start, so Dequeue leaves the fence at the proper location. Test_If_Empty can also depend on the fence being at the start, so its only task is to ask if the fence is also at the finish. All other operations remain the same.

The following is the relevant portion of Queue_15's package body when the fence is maintained at the start of the list:

package body Queue_15_With_Conventions is

--- Declarators -----------------------------------------------

--! type Queue
--!  convention
--!    "q.rep.left = EMPTY"
--!  correspondence
--!    "q = q.rep.left  o  q.rep.right"
--- Exported Operations -------------------------------
---

procedure Enqueue (  
  q: in out Queue;  
  x: in out Item  
) is
begin  
  Move_To_Finish (q.rep);  
  Add_Right (q.rep, x);  
  Move_To_Start (q.rep);  
end Enqueue;

---

procedure Dequeue (  
  q: in out Queue;  
  x: in out Item  
) is
begin  
  Remove_Right (q.rep, x);  
end Dequeue;

---

procedure Test_If_Empty (  
  q: in out Queue;  
  empty: in out Boolean  
) is
begin  
  Test_If_At_Finish (q.rep, empty);  
end Test_If_Empty;

end Queue_15_With_Conventions;

2.3.3 Eliminating Package Body Incest

We need to make clear the distinction between a type T exported by component C, and T’s representation within C’s package body. External to C, the client programmer reasons about the exported type using the mathematical model for type T. Internally, i.e., within the package body, the component implementer generally reasons about T using its representation type. This makes sense from the point of view that the purpose of the package body is to implement the external interface presented by the package specification. Extending this point of view leads one to the conclusion that the external interface should not be available for use within its own package body.
Furthermore, recall that package-level conventions can temporarily be violated during the execution of an exported operation; they need be satisfied only at the conclusion of each operation. However, each exported operation can depend on the conventions being satisfied when called. If we allow the implementation of a component’s exported operation to call any of the component’s other exported operations then the possibility exists that the call will be made while the conventions are not satisfied. The following principle provides one way to guarantee that this does not happen:

**Principle 27** — When implementing any operation in a component’s package body, do not call any of the component’s exported operations.

Since Principle 27 prohibits an implementation of a package body’s operation from calling any of the exported operations, it makes no sense to declare variables in the package body of the exported type. A declaration of this kind is possible, but nothing can be done with the declared variable because Principle 27 prohibits calls to any of the operations that manipulate the variable. Therefore, we introduce the following (corollary) principle:

**Principle 28** — In the package body of a component, do not declare local variables of the component’s exported types at the package level, or in any of the component’s exported or local operations.

These two principles support understandability because there is a uniform, consistent, and clean cut between the external interface and its internal implementation. Externally the engineer reasons using the abstract type, and internally using the representation type.

### 2.3.4 Implementing Initialize_Package and Finalize_Package

Next we turn our attention to implementing the exported operations. We start with the Initialize_Package and Finalize_Package operations:
**Principle 29** — In the package body of component C, implement “Initialize_Package” as follows:

- First, initialize all packages that C instantiates, in the order they are instantiated, by calling their respective Initialize_Package operations.
- Second, initialize all package-level variables, in the order they are declared, by calling the appropriate Initialize operations.
- Third, perform all other required computations involving the package-level variables. These should make the package-level variables have the values that are required prior to the invocation of C’s other exported operations.

**Principle 30** — In the package body of component C, implement “Finalize_Package” as follows:

- First, finalize all package-level variables, in the reverse of the order they are declared, by calling the appropriate Finalize operations.
- Second, finalize all packages that C instantiates, in the reverse of the order they are instantiated, by calling their respective Finalize_Package operations.

For components that do not instantiate any packages, and do not have any package-level variables, Initialize_Package and Finalize_Package have nothing to do. This is often the case for fully parameterized components, e.g., Queue_15. Here are Queue_15’s Initialize_Package and Finalize_Package:

```plaintext
procedure Initialize_Package is
begin
    null;
end Initialize_Package;
```

```plaintext
procedure Finalize_Package is
begin
    null;
end Finalize_Package;
```

Examples of Initialize_Package and Finalize_Package with non-null bodies can be found in Section 3.3.
2.3.5 Implementing Initialize

For each type T exported by a component, an Initialize operation is exported whose job is to initialize variables of type T. The ensures clause for Initialize appears in the formal specification of the corresponding type (see Section 2.1.14). For example, Queue’s initialization ensures clause states “q = EMPTY.” Initialize must also guarantee that the component’s package-level representation conventions are satisfied (see Section 2.3.2), in order that the correspondence should be well-defined.

**Principle 31** — Implement “Initialize (x: in out T)” as follows:

- First, initialize the representation of x by calling the Initialize operation for x.rep.

- Second, initialize all local variables declared in Initialize, in the order that they are declared, by calling the appropriate Initialize operations.

- Third, perform all other computations required to create a proper abstract initial value for x, i.e., to satisfy type T’s representation conventions so the corresponding abstract value of x satisfies T’s initialization ensures clause.

- Fourth, finalize all local variables declared in Initialize, in the reverse of the order in which they are declared, by calling the appropriate Finalize operations.

The implementation of an Initialize operation often amounts to a call to the representation type’s Initialize operation. (Two_Way_List_Template’s Initialize implementation performs more than just a call to the representation type’s Initialize, see Section 4.4.2.) Here is Queue_15’s Initialize:

```plaintext
procedure Initialize (  
  q: in out Queue  
) is  
begin  
  Initialize (q.rep);  
  end Initialize;
```

2.3.6 Implementing Finalize

For each type T exported by a component, a Finalize operation is exported whose job is to finalize variables of type T. Finalize is invoked after a variable is no longer needed. Only if finalization of a variable changes the abstract state of a package is there a need for an ensures clause for Finalize. In such a case, an ensures clause appears in the formal specification of the exported type (see Section 2.1.14). (Note that Principle 2 suggests that there should be no package-level abstract state.) Finalize is always important, though,
because it provides the means for a component to reclaim dynamically allocated resources consumed by a variable’s representation (e.g., storage).

**Principle 32** — Implement “Finalize (x: in out T)” by calling the Finalize operation for x.rep.

Here is Queue_15’s Finalize:

```plaintext
procedure Finalize (  
    q: in out Queue  
) is  
begin  
    Finalize (q.rep);  
end Finalize;
```

2.3.7 Implementing Swap

For each type T exported by a component, a Swap operation is exported providing data movement for variables of type T.

**Principle 33** — Implement “Swap (x1: in out T; x2: in out T)” by calling the Swap operation for the representation type of T.

Here is Queue_15’s Swap:

```plaintext
procedure Swap (  
    q1: in out Queue;  
    q2: in out Queue  
) is  
begin  
    Swap (q1.rep, q2.rep);  
end Swap;
```

2.4 Local Certification of Composability, Correctness, Reusability and Understandability

In this section we discuss why each property (composability, correctness, reusability, understandability) is locally certifiable for components that are constructed by following Chapter 2’s principles.
2.4.1 Local Certification of Composability

Generally there are two methods of composition, by conceptual type and by facility, i.e., an instance of a component that includes all its exported types and operations (except Initialize_Package and Finalize_Package). To certify composability of a component C, we must certify that:

1) C composes properly with each imported conceptual type and C’s exported types compose properly with other components that import types;

2) C composes properly with each imported facility and instances of C compose properly with other components that import an instance of C.

To locally certify composability of component C, we must be able to certify item 1 above by examining only C’s package specification. To certify that C composes properly with other components that import an instance of C (from item 2) also requires only examining C’s package specification. Certifying that C composes properly with each imported facility (from item 2) requires examining C’s package specification and only one package specification of a component corresponding to each facility parameter.

For example, to locally certify composability for Queue_15, we should only have to examine Queue_15’s package specification and the package specification of any implementation of One_Way_List_Template. Figure 15 illustrates this by enclosing Queue_Template and One_Way_List_Template by dashed rounded boxes.

![Diagram showing composability certification](image)

Figure 15 — Items Examined to Locally Certify Composability

Why locally certifying item 1 (i.e., that C composes properly by conceptual type) is possible. Recall the following principles (some are paraphrased):
• Principle 3 — Export a Finalize operation for each exported type (Section 2.1.3);
• Principle 4 — Export an Initialize operation for each exported type (Section 2.1.4);
• Principle 7 — Export each type as limited private (Section 2.1.6);
• Principle 8 — Export a Swap operation for each exported type (Section 2.1.7);
• Principle 9 — Export and import all operations as procedures (Section 2.1.8);
• Principle 10 — Make all formal parameters have the mode “in out” (Section 2.1.8);
• Principle 14 — For each type parameter, import the type as limited private and import the type’s standard operations: Initialize, Finalize and Swap (Section 2.1.12).

*C composes properly with each imported conceptual type.*

Because of Principles 3, 4, 7, 8, 9 and 10 we are assured that any component that has been certified for composability exports its types as limited private and exports the standard operations Initialize, Finalize and Swap. Furthermore, because of these principles, the standard operations are exported as procedures and all their parameter modes are “in out.”

To see if Principle 9, 10 and 14 are observed when constructing C, we have to inspect C’s conceptual parameter section to see if all imported types are limited private and to see if each imported type’s standard operations are imported as procedures with mode “in out.”

*C’s exported types compose properly with other components that import types.*

Because of Principle 9, 10 and 14 we are assured that any component that has been certified for composability imports all types as limited private and imports each imported type’s standard operations as procedures with mode “in out.”

To see if Principles 3, 4, 7, 8, 9 and 10 are observed when constructing C, we have to inspect C’s interface section to see if all types are exported as limited private and to see if the standard operations Initialize, Finalize and Swap are exported for each type. Furthermore, we have to see if the standard operations are exported as procedures and if all their parameter modes are “in out.”

*Why locally certifying item 2 (i.e., C composes properly with each imported facility and instances of C compose properly with other components that import an instance of C) is possible.* Recall the following principles:

• Principle 15 — A component C that needs to use some other component should be parameterized by all of the used component’s types and operations in the same order and using the same names (Section 2.1.13);

• Principle 18 — Multiple implementations for the same concept should differ only in the package name and the realization context (Section 2.1.16);

*C composes properly with each imported facility.*

Because of Principle 18 we are assured that any component that has been certified for composability differs from other implementations of the same concept only in its name and
its realization context. Since the used component’s exported types and operations are in the interface section of the package specification and not in the realization context, we need only look at one corresponding component’s package specification. Other implementations of the same concept will have the same interface.

To see if Principle 15 is observed when constructing C, we have to inspect C’s package specification and the package specification of the used component to determine if C imports all of the used component’s exported types and operations.

**Instances of C compose properly with other components that import an instance of C.**

Because of Principle 15 we are assured that any component that has been certified for composability and uses a facility instance of component C is parameterized with all of C’s exported types and operations.

To see if Principle 18 is observed when constructing C, we have to inspect C’s package specification to determine if all realization-related information (i.e., implementation details such as the names of used facilities) are contained entirely in the realization context.

In summary, locally certifying composability is possible because the principles introduced in this chapter reduce certification to inspecting a fixed number of package specifications. The total number of package specifications requiring inspection always equals one (the package specification of the component under consideration) plus the number of facility parameters of the component under consideration.

### 2.4.2 Local Certification of Correctness

Recall from Section 1.3.3 that certifying correctness of a component C means we certify that C’s implementation meets its specification. Furthermore, local certification of correctness means that certifying correctness only requires examining C’s implementation, its specification, and the specifications of the concepts that it uses. It does not depend on any client of C, nor on the implementation of any concept that C’s implementation uses. The bounded area in Figure 16 illustrates the information required for local certification of correctness.
There appear to be two main enemies of local certification of correctness: *intermodule interference*, and *interprocedural interference*. First we discuss interprocedural interference, then intermodule interference.

**Eliminating interprocedural interference.** This occurs when a procedure is called with an alias and because of that, it produces an unspecified side-effect in some other part of the program. By permitting aliasing, there is always the potential for interprocedural interference. Because of this, we lose the ability to locally certify correctness. Why? Implicit in our specification of an operation is the clause “and nothing else changes.” When aliasing is present, we can guarantee this only by examining the client program to see if it is called with no aliased variables. But local certification of correctness of an operation means that the correctness of an operation can be certified without knowledge of the client program. This is how we lose local certification of correctness in the presence of aliasing.

One might think that if we could fully specify the behavior of an operation in the presence of aliases, then maybe aliasing would not be a problem. But in general, an operation’s unspecified side-effects (because of aliasing) cannot be “specified away” a priori. Why? First, it is difficult, if not impossible, to predict the behavior of the side-effect in the presence of genericity. Second, it is not usually known at the time the specification is written if the caller will always call with an aliased variable.

To eliminate the potential for interprocedural interference, we must eliminate aliasing. Section 2.2.3 discusses aliasing and identifies the principles — that when followed — prevent us from introducing aliasing into our programs, and eliminate the potential for interprocedural interference. The following briefly reviews the discussion of Section 2.2.3.

Recall the following principles:
• Principle 20 — Each variable must be declared using one of the built-in Ada scalar types (i.e., Boolean, Character, Float or Integer), or a type exported from a component that has been designed using the component interface principles of Section 2.1. No other variable declarations are allowed (Section 2.2.1).

• Principle 21 — In any procedure call, an actual parameter should not appear more than once (Section 2.2.3).

• Principle 22 — In any procedure call, the caller should not supply a variable that is globally known to the called procedure (Section 2.2.3).

Principle 20 eliminates explicit aliasing through the use of Ada’s access type by not allowing variables to be declared as access types. It also eliminates a form of implicit aliasing illustrated by the following procedure call:

\[ P(a(i), a(j)); \]

In the call to procedure P, if the indices i and j are equal then P’s formal parameters name the same location in array “a.” Principle 20 eliminates this form of aliasing because it prohibits variables from being declared using Ada’s built-in array constructor.

Principle 21 eliminates another form of implicit aliasing by not allowing calls of the form

\[ P(x, x); \]

Finally, Principle 22 eliminates procedure calls of the form:

\[ P(x); \]

where x is globally known to P.

Eliminating intermodule interference. Intermodule interference can occur when abstract state is maintained in a component instance. For example, Queue_2 (from Section 2.1.1) maintains abstract state in the component. The following is Queue_2’s package specification with formal comments added. Notice in the conceptual context section there appears a module state variable “q.” This variable is the manifestation of Queue_2’s abstract state, i.e., if Queue_2 did not maintain abstract state, this variable would not exist. With “q,” Enqueue and Dequeue’s abstract behavior are specified.

```ada
--! concept Queue_Object

--! generic

--! conceptual context

--! uses

--! STRING_THEORY_TEMPLATE

--! mathematics

--! math facilities

--! STRING_THEORY is

STRING_THEORY TEMPLATE (integer)

--! variables
initially "q = EMPTY"

package Queue_2 is

interface

procedure Enqueue ( x: in Integer ; --! preserves

--! other affected variables
--! alters q
--! ensures "q = APPEND (#q, #x)"

procedure Dequeue ( x: out Integer ; --! produces

--! other affected variables
--! alters q
--! requires "q /= EMPTY"
--! ensures "PREPEND (q, x) = #q"

Underflow: exception;

end Queue_2;
end Queue_Object;

To see how having abstract module-level state can lead to intermodule interference, we construct the following hypothetical situation:

- Queue_Object is used to implement another component called Priority_Queue_Object. Conceptually, Priority_Queue_Object is modeled as a string of items, like Queue_Object. The client can add items with varying priorities to the string and remove them one at a time, always receiving the highest priority item from the string.

- Internally, Priority_Queue_Object is implemented using Queue_Object, and maintains the convention that the items are always in priority order.

- Priority_Queue_Object is parameterized by Queue_Object, so when a client program uses Priority_Queue_Object, it must first instantiate Queue_Object.

Figure 17 illustrates the the dependence of the client program and Priority_Queue_Object on Queue_Object.
Unfortunately, intermodule interference can occur between Priority_Queue_Object (which is a client of Queue_Object) and any other client of Queue_Object (e.g., the client program). This is because all clients of Queue_Object have Queue_Object’s abstract module-level state in common, i.e., Queue_Object’s module state variable q. In our example, if the client program enqueues some items onto Queue_Object’s q, these items would also be in the priority queue provided by Priority_Queue_Object. Since Priority_Queue_Object maintains a convention that all items are maintained in priority order, and since these items were put there by Queue_Object’s Enqueue (not by Priority_Queue_Object’s Enqueue), they probably will not be in priority order. This is an example of intermodule interference. Intermodule interference makes local certification of correctness impossible because certifying correctness of a component such as Priority_Queue_Object requires knowledge of Priority_Queue_Object’s client; this is shown by the bounded area in Figure 18.
Components such as Priority.Queue.Object can be certified correct only when all other clients of Queue.Object are known, and it has been shown that none of the other clients interfere with Priority.Queue.Object through Queue.Object’s abstract state. If we permit components to maintain abstract module-level state (i.e., similar to Queue.Object), we lose local certification of correctness. Therefore, we need Principle 2 (Section 2.1.2) which states, “Export a type so that abstract state is maintained in variables of that type, not in package instances.” By maintaining abstract state in variables, and not in the component, we eliminate the potential for intermodule interference.

By eliminating interprocedural interference and intermodule interference, we achieve the potential for local certification of correctness of our components. Formal details of local correctness proofs (semantics, proof rules, etc.) can be found in [Krone 88]; see also [Ernst 91].
2.4.3  Local Certification of Reusability

To certify reusability of a fully-parameterized component C, we must certify the items in the following list. Each item identifies different aspects of a component that must be certified and what is required to perform the certification.

- C is parameterized by each ADT it manipulates but does not export.

  Recall Principle 13: Parameterize a component by each ADT that it manipulates but does not export (Section 2.1.11). Certification requires examining only C’s package specification because the ADTs that C manipulates must show up as parameters in C’s operations. For example, Queue_Template is parameterized by the items that can be enqueue and dequeued. If it were not parameterized, then a specific type would appear in Enqueue’s and Dequeue’s parameter list, e.g., Integer.

- C is parameterized by each component used by its implementation.

  Recall Principle 15: A component C that needs to use some other component should be parameterized by all of the used component’s types and operations (Section 2.1.13). There are two ways to certify this. First, one could examine C’s package specification and body for any components that have been “with”ed and then instantiated. If any are found then the component cannot be certified for the reusability property. Second, one could list all components used by the package body and then examine the generic parameter list to see that they are all supplied as facility parameters. If not, then the component cannot be certified for the reusability property. Either certification method requires examining only C’s package specification and body.

- C’s operations are designed so that they are not forced to check their preconditions.

  Recall Principle 6: Do not export any exceptions, and design the exported operations so that they do not raise any exceptions (Section 2.1.5). Certification requires examining the package specification to see that no exceptions are exported. It also requires examination of the package body to see that no operation checks its precondition and raises an exception. If this is not the case, then the component cannot be certified for the reusability property.

- C permits clients to efficiently move values of variables declared from C’s exported types.

  Recall Principle 8: Export an operation (called Swap) for each exported type, which exchanges the values of its arguments (Section 2.1.7). Certification requires examining the package specification to determine if Swap is exported for each of C’s exported types.

- C permits clients to reclaim resources dynamically allocated as part of the exported type’s implementation (e.g., dynamically allocated storage).

  Recall Principle 3: For each exported type, export a finalization operation (called Finalize) with a single parameter of mode “in out.” Certification requires examining the package specification to determine if Finalize is exported for each of C’s exported types.
Since each of these items can be certified by examining either the component’s package specification or its package body, certification can be done locally. For example, to locally certify reusability for Queue_15, we should only have to examine Queue_15’s package specification and its package body. Figure 19 illustrates this by enclosing Queue_Template and Queue_15 Implementation by a dashed rounded box.

![Figure 19 — Items Examined to Locally Certify Reusability and Understandability](image)

2.4.4 Local Certification of Understandability

In trying to achieve understandability in our components we have focused on two areas:

- reducing the cognitive load on the component’s client programmer and the component’s designer; and

- improved communication of the abstract behavior of the component to the client programmer and the component implementer.

Locally certifying understandability can be achieved because it requires only examining the component’s package specification and body (see Figure 19), and a fixed set of mathematical theories on which the component’s specification is based.

In pursuit of reducing the cognitive load, we have focused on reducing complexity by insisting on uniformity and consistency. An implication of this is that the number of choices at any particular design step is often dramatically reduced. For example, Principle 3 states: For each exported type, export a finalization operation (called Finalize) with a single parameter of mode “in out.” Just this one principle reduces the choices to one for the following design decisions:
• There should be a finalization operation for each exported type.
• Its name should be Finalize.
• It should only have one parameter.
• Its one parameter should have mode “in out.”

Because the principles are specific, each principle in one way or another promotes uniformity and consistency. Therefore, certifying uniformity and consistency requires determining if all of the principles have been followed during component design and implementation.

There are two principles that directly support improved communication of the abstract behavior of the component to the client programmer. Recall Principles 16 and 17: Principle 16 — Formally specify the behavior of each component using a mathematical specification language (Section 2.1.14); and Principle 17 — When designing and implementing a component, consistently follow a “reasonable” comprehensive set of naming and formatting conventions. Certifying that both of these principles have been followed again requires examination of only the component’s package specification.

2.5 Summary

Chapter 1 introduced the definition of a discipline for constructing software components (Section 1.2). Furthermore, it formulated a language-independent, component-type-independent discipline for constructing high-quality software components (Section 1.3). This formulation introduced a goal (i.e., high quality) and properties (composability, correctness, reusability, and understandability). Only one principle was introduced because principles must provide specific details on how to construct the software component, and those details depend on the exact nature of the programming language and component type.

In this chapter we continued the formulation of the discipline (for constructing high-quality components) begun in Chapter 1 by fixing the programming language to be Ada and the component type to be abstract data types, and by introducing 33 specific principles for constructing components in Ada.

Section 2.1 introduced 18 principles for constructing a component’s package specification. The purpose behind these principles is to specifically identify the steps that are required in order to construct components so that their interface does not make it impossible for them to possess the composability, correctness, reusability and understandability properties.

The primary mission of many of the principles is to promote the correctness property. One aspect of this property is that the component must support abstract reasoning about its behavior. This implies a requirement for components to have an abstraction barrier with absolutely no implementation leaks. One principle which directly supports this is Principle 7 (Section 2.1.6). The example provided in Section 2.1.6 demonstrates that without this principle, we can easily construct components that have implementation leaks.

Section 2.2 introduced six principles for implementing clients of components that have been constructed using the interface principles of Section 2.1. A client is either a main program or another component so these client implementation principles also apply to
components whose implementations use other components. Of these, Principles 20, 21 and 22, which eliminate unwanted aliasing, are the most important for assuring local certifiability of correctness.

Section 2.3 introduced nine principles for implementing components that have been constructed using the interface principles of Section 2.1. One of the more interesting aspects of Section 2.3 appears in Section 2.3.2 which discusses the formal comments that appear in the package body (because of Principle 16 introduced in Section 2.1.14). Among other things, this section discusses the need for, and the practical value of specifying, the component’s correspondence and conventions.

Section 2.4 demonstrated why each of the four high-quality properties is locally certifiable. By being locally certifiable, we mean there is always a fixed number of items in the “local area” of the component that must be examined in order to certify its possession of a particular property. For example, certification of correctness requires examining the component’s package specification, package body, and the package specifications of the components used by the component’s implementation. It does not depend on any of these used components’ package bodies, nor on any client program.

Finally, the precise formulation of these principles did not happen by accident. Rather, they were chosen because they identify a restricted subset of Ada constructs that correspond to similar constructs in RESOLVE — which has been demonstrated to have a sound and relatively complete proof system [Krone 88], [Harms 90]. As discussed in Section 2.4, the constructs from this restricted set can be used to build “safe” components, i.e., components that can be locally certified for the high-quality properties. Furthermore, the formal comments used to specify the component’s functional behavior (see Sections 2.1.14 and 2.3.2) are based on RESOLVE’s formal specification language. Our Ada components share many qualities with RESOLVE components because we use a restricted set of Ada constructs (that correspond to RESOLVE constructs) and because we use RESOLVE’s formal specification language in our formal comments. Therefore, we refer to our Ada components as RESOLVE/Ada components. In the next chapter we continue our discussion of RESOLVE/Ada components by introducing methods for adding additional capabilities, for testing and debugging, and for creating partial instantiations.