

CHAPTER III

Additional Capabilities, Support for Testing and Debugging, and Partial Instantiation

The previous chapter introduced principles for constructing a component's interface. It also introduced principles to be followed when implementing a component. This chapter discusses how to add various "extensions" to these components. Section 3.1 discusses how to add additional operations to a component. Section 3.2 discusses methods supporting testing and debugging that take advantage of the component's formal comments. Section 3.3 discusses how to make fully-parameterized components more convenient to use by using partial instantiation.

3.1 Adding Additional Capabilities to a Component

This section examines the issue of how to add additional operations (called *capabilities*) to a RESOLVE/Ada component. The need for additional capabilities arises when a client needs some operation from a component that is not directly exported by the component. There are two methods for providing the additional capability. One method is to "layer" the new operation on top of the component, using only its exported types and operations. The second method is to change the component so that it exports the new operation. This second method we call "directly implementing" the operation because the operation is implemented directly on the component's underlying representation. We discuss the benefits and drawbacks of these two approaches and provide examples illustrating each method.

Layering an additional capability on top of a component. This method requires the engineer to build the additional capability by using only the types and operations exported by the component. For example, suppose a client needs an equality testing operation for Queue; see Section 2.1.8 for a layered implementation of this operation. The benefits of layering are: that the layered operation can be used with other queue components whose concept is the same as Queue_15's, regardless of their implementation; and that the engineer reasons about the layered operation based on the component's abstraction rather than its underlying representation and implementation.

A drawback might be unacceptable efficiency. For example, by examining the implementation of Test_If_Equal (in Section 2.1.8), one sees that once the two queues

have been found to be not equal, `Test_If_Equal` cannot just quit; it must still handle each item of each queue. A directly implemented `Test_If_Equal` probably would not have to handle any more of the items after it has found the two queues to be not equal. This may or may not turn out to give unacceptable performance depending on how often `Test_If_Equal` is used, how often the compared queues turn out to be unequal, and how close to the front of the queues the inequality is first detected.

Directly implementing an additional capability. If the new operation does not use the component's exported types and operations, it must directly manipulate the exported type's internal representation. The benefit of this can be a more efficient operation. The drawbacks are: that the implementation cannot be used with any other component that implements the same concept, but has a different representation; that the engineer has to reason using the representation instead of using the exported type's abstract model; that the engineer has to understand the component's conventions (see Section 2.2.3) and guarantee that the new operation abides by those conventions; and that the component's interface changes, so all previous clients of the component have an outdated version of the component and will need to be recompiled and relinked.

[Hollingsworth 91] reports the results of an empirical pilot study on programmer productivity and code quality due to the effects of layering an operation instead of directly implementing it. The statistical results from that study suggest that there is potential for gain in productivity if additional capabilities are layered instead of directly implemented. Furthermore, a significant difference in code quality (based on the number of defects causing run-time errors that were found and fixed before testing revealed no more defects) was reported. The layered implementations had significantly fewer defects than the direct implementations. The most reasonable explanation for these results seems to be that the engineer of the directly implemented operation incurs additional cognitive load because it is necessary to deal with the representation's abstraction rather than relying solely on the component's abstraction. The representation's abstraction is one level farther removed from the problem domain of the engineer than is the component's abstraction. What's more, with the layered implementation the engineer does not have to be concerned with the component's internal conventions. In light of this study, we introduce the following principles:

Principle 34 — Add additional capabilities to a component by layering, when layering is possible.

Principle 35 — Directly implement a capability only if a) layering is not possible or b) the layered implementation has been shown (e.g., by program instrumentation) to exact an unacceptable performance penalty for a particular application.

Section 2.1.8 introduced a layered version of `Test_If_Equal`, but its interface does not follow the principles of the discipline. For example, it violates Principle 1, "Make generic packages the unit of modularity." The following specification for `Test_If_Equal` adheres to the discipline's principles:

```

--! concept Queue_Equal_Capability

    generic
--!   conceptual context

--!     uses
--!       Queue_Template

--!     conceptual parameters

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

--!   facility Queue_Facility is
--!     Queue_Template (Item)
      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 out Item);
      with procedure Dequeue (
        q: in out Queue;
        x: in out Item);
      with procedure Test_If_Empty (
        q: in out Queue;
        empty: in out Boolean);

--!   realization context

--!     realization parameters

      with procedure Test_If_Equal (
        x1: in out Item;      --! preserves
        x2: in out Item;      --! preserves
        equal: in out Boolean --! produces
      );
--!   ensures "equal iff (x1 = x2)"

```

```

package Queue_Equal_Capability is
--! interface

    procedure Initialize_Package;
    procedure Finalize_Package;

    procedure Test_If_Equal (
        q1: in out Queue;           --! preserves
        q2: in out Queue;           --! preserves
        equal: in out Boolean       --! produces
    );
--! ensures "equal iff (q1 = q2)"

end Queue_Equal_Capability;
--! end Queue_Equal_Capability

```

A possible implementation for the component Queue_Equal_Capability might be:

```

with Built_In_Types; use Built_In_Types;

```

```

package body Queue_Equal_Capability is

```

```

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

```

```

procedure Initialize_Package is
begin
    null;
end Initialize_Package;

```

```

-----
procedure Finalize_Package is
begin
    null;
end Finalize_Package;

```

```

-----
procedure Test_If_Equal (
    q1: in out Queue;
    q2: in out Queue;
    equal: in out Boolean
) is
    empty_q1: Boolean;
    empty_q2: Boolean;
    x1: Item;
    x2: Item;

```

```

        catalyst_1: Queue;
        catalyst_2: Queue;
begin
        Initialize (empty_q1);
        Initialize (empty_q2);
        Initialize (x1);
        Initialize (x2);
        Initialize (catalyst_1);
        Initialize (catalyst_2);

        equal := true;

        loop
            Test_If_Empty (q1, empty_q1);
            Test_If_Empty (q2, empty_q2);
        exit when not equal or empty_q1 or empty_q2;
            Dequeue (q1, x1);
            Dequeue (q2, x2);
            Test_If_Equal(x1, x2, equal);
            Enqueue (catalyst_2, x2);
            Enqueue (catalyst_1, x1);
        end loop;

        if not empty_q1 then
            equal := false;
            loop
                Dequeue (q1, x1);
                Enqueue (catalyst_1, x1);
                Test_If_Empty (q1, empty_q1);
            exit when empty_q1;
            end loop;
        end if;

        if not empty_q2 then
            equal := false;
            loop
                Dequeue (q2, x2);
                Enqueue (catalyst_2, x2);
                Test_If_Empty (q2, empty_q2);
            exit when empty_q2;
            end loop;
        end if;

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

        Finalize (catalyst_2);
        Finalize (catalyst_1);
        Finalize (x2);
        Finalize (x1);
        Finalize (empty_q2);
        Finalize (empty_q1);
end Test_If_Equal;

```

```
end Queue_Equal_Capability;
```

The following two examples of how to directly implement Test_If_Equal illustrate some of the differences between a layered implementation and a direct implementation. One implementation of Test_If_Equal is based on Queue_15's implementation where there are no representation conventions to be observed. The other is based on Queue_15's implementation where the fence is always maintained at the start of the list. See Section 2.3.2 for implementations of Queue_15 without conventions and Queue_15 with this convention.

The following is a possible implementation of Test_If_Equal when no conventions are observed:

```
procedure Test_If_Equal (
    q1: in out Queue;
    q2: in out Queue;
    equal: in out Boolean
) is
    x1: Item;
    x2: Item;
    at_finish1: Boolean;
    at_finish2: Boolean;
begin
    Initialize (x1);
    Initialize (x2);
    Initialize (at_finish1);
    Initialize (at_finish2);

    equal := true;
    Move_To_Start (q1.rep);
    Move_To_Start (q2.rep);
    loop
        Test_If_At_Finish (q1.rep, at_finish1);
        Test_If_At_Finish (q2.rep, at_finish2);
    exit when not equal or at_finish1 or at_finish2;
        Remove_Right (q1.rep, x1);
        Remove_Right (q2.rep, x2);
        Test_If_Equal (x1, x2, equal);
        Add_Right (q1.rep, x1);
        Add_Right (q2.rep, x2);
        Advance (q1.rep);
        Advance (q2.rep);
    end loop;
    equal := equal and at_finish1 and at_finish2;

        Finalize (at_finish2);
        Finalize (at_finish1);
        Finalize (x2);
        Finalize (x1);
end Test_If_Equal;
```

Notice the first thing that must be done is to move the fence to the start of both lists, and when `Test_If_Equal` is finished, it can leave the fence at any location. The engineer who directly adds this operation to `Queue_15` would not need to know the conventions in order to figure out that both `q1.rep`'s fence and `q2.rep`'s fence need to be at the start. However, to know that the fence can be left at any location when `Test_If_Equal` is finished requires knowing the component's conventions. This information can be ascertained in two ways: the "easy way" by examining the formal comment stating the conventions, or if no conventions are stated, the "hard way" by examining each of the other operations.

The following is a possible implementation of `Test_If_Equal` when the fence is maintained at the start:

```

procedure Test_If_Equal (
    q1: in out Queue;
    q2: in out Queue;
    equal: in out Boolean
) is
    x1: Item;
    x2: Item;
    at_finish1: Boolean;
    at_finish2: Boolean;
begin
    Initialize (x1);
    Initialize (x2);
    Initialize (at_finish1);
    Initialize (at_finish2);

    equal := true;
    loop
        Test_If_At_Finish (q1.rep, at_finish1);
        Test_If_At_Finish (q2.rep, at_finish2);
    exit when not equal or at_finish1 or at_finish2;
        Remove_Right (q1.rep, x1);
        Remove_Right (q2.rep, x2);
        Test_If_Equal (x1, x2, equal);
        Add_Right (q1.rep, x1);
        Add_Right (q2.rep, x2);
        Advance (q1.rep);
        Advance (q2.rep);
    end loop;
    equal := equal and at_finish1 and at_finish2;
    Move_To_Start (q1.rep);
    Move_To_Start (q2.rep);

    Finalize (at_finish2);
    Finalize (at_finish1);
    Finalize (x2);
    Finalize (x1);
end Test_If_Equal;

```

In this version, the implementer of `Test_If_Equal` can depend on the fence for both `q1.rep` and `q2.rep` being at the start, so the implementation does not have to move the two fences at the outset. But the implementer must make sure that both `q1.rep`'s fence and `q2.rep`'s

fence are at the start when `Test_If_Equal` is finished. The fact that the implementer has to be concerned with the component's conventions points out the need for their specification and also reveals why there might be extra complexity involved when directly implementing additional capabilities.

3.2 Support for Testing and Debugging

Bentley [Bentley 88, page 27] defines software *scaffolding* as “temporary programs and data that give programmers access to system components.” He goes on to say that scaffolding “is indispensable during testing and debugging.” This section discusses how to provide scaffolding for RESOLVE/Ada components. Specifically, we discuss how to provide operations that display a variable's value and operations that check their own preconditions. We also discuss how these operations support testing and debugging, and when they are most valuable.

3.2.1 Display of the Abstract Model

Principle 16 (see Section 2.1.14) requires that we formally specify the abstract behavior of a component using a mathematical specification language. In this section we illustrate how a component's specification can be used to help develop a uniform method for displaying values of variables.

One possible approach for displaying a variable's value is to display it in terms of its abstract mathematical model. The basis for this is:

- All RESOLVE/Ada components are specified using a mathematical model specification language that is based on existing mathematical theories (e.g., integer theory, string theory, set theory, etc.).
- By convention we can choose to display math objects (that model our program objects) in a particular way (e.g., always display a set in the same manner).
- Given a variable declared from a component's exported type, we can convert the variable's value from its representation to its corresponding math model value [Hegazy 89] based on the correspondence formal comment (see Section 2.3.2).
- Once it is converted, we can then use our convention for displaying math objects to display the variable's math model value.

For example, `Queue_15` is modeled as a mathematical string of items. A convention for displaying strings might be to display an opening symbol (e.g., “<”) followed by each item on a new line, and terminate the string by a closing symbol (e.g., “>”). The display operation for `Queue` can be a layered operation since we can obtain each item in the `Queue` by using the `Dequeue` operation. To display an item requires type `Item`'s display operation, so this operation will have to be supplied as a generic parameter. The following is a package specification for a `Queue` display capability:

```
--! concept Queue_Display_Capability
```

```

generic
--! conceptual context

--!
--!     uses
--!         Queue_Template

--!
--!     conceptual parameters

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

--!
--!     facility Queue_Facility is
--!         Queue_Template (Item)
--!         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 out Item);
--!         with procedure Dequeue (
--!             q: in out Queue;
--!             x: in out Item);
--!         with procedure Test_If_Empty (
--!             q: in out Queue;
--!             empty: in out Boolean);

--!
--!     realization context

--!
--!     realization parameters

--!         with procedure Display (
--!             x: in out Item;           --! preserves
--!             indent: in Integer       --! preserves
--!         );
--!     ensures "x's math model is output
--!         starting indent spaces right of the cursor
--!         position at the time of the call, leaving
--!         the cursor position at the end of the
--!         displayed value"

```

```

package Queue_Display_Capability is
--! interface

    procedure Initialize_Package;
    procedure Finalize_Package;

    procedure Display (
        q: in out Queue;           --! preserves
        indent: in Integer       --! preserves
    );
--! ensures "q is output as a string of items
--!          starting indent spaces right of the cursor
--!          position at the time of the call, leaving
--!          the cursor position at the end of the
--!          displayed value"

    end Queue_Display_Capability;
--! end Queue_Display_Capability

```

A possible implementation for Queue_Display_Capability might be:

```

with Text_Io;           use Text_Io;
with Built_In_Types;   use Built_In_Types;
package body Queue_Display_Capability

```

```

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

```

```

procedure Initialize_Package is
begin
    null;
end Initialize_Package;

```

```

-----

procedure Finalize_Package is
begin
    null;
end Finalize_Package;

```

```

-----

procedure Display (
    q: in out Queue;
    indent: in Integer
) is
    empty_q: Boolean;
    x: Item;
    catalyst: Queue;
begin

```

```

        Initialize (empty_q);
        Initialize (x);
        Initialize (catalyst);

    for i in 1..indent loop
        Put (" ");
    end loop;
    Put ("<");
    New_Line;

    loop
        Test_If_Empty (q, empty_q);
    exit when empty_q;
        Dequeue (q, x);
        Display (x, indent + 3);
        New_Line;
        Enqueue (catalyst, x);
    end loop;

    for i in 1..indent loop
        Put (" ");
    end loop;
    Put (">");

    Swap (q, catalyst);

        Finalize (catalyst);
        Finalize (empty_q);
        Finalize (x);
    end Display;
end Queue_Display_Capability;

```

Zweben suggests that it “appears reasonable to require that, at least for testing purposes, each module contain a display operation for any type that it provides”; see [Zweben 89, page 19]. So one use for `Queue_Display_Capability` is for testing any `Queue_Template` implementation. It works with any `Queue_Template` implementation because it is layered. An interactive driver program — a “command interpreter” that can execute the component’s individual operations on demand — can be created that allows a tester to create various configurations of Queue variables using `Initialize`, `Enqueue`, `Dequeue`, and `Swap`. By including `Display` among the driver’s commands, the tester can at any time visualize the abstract value of a Queue variable in its different configurations.

The goal of testing a component is to uncover defects in the implementation of the operations. Therefore, another benefit of being layered is that the `Display` operation uses the operations exported by the component. If any of the operations used by `Display` have defects, the tester might detect them just by using `Display`. It happens to be the case that `Queue_Display_Capability` uses all of `Queue_Template`’s exported operations (except `Initialize_Package` and `Finalize_Package`, which should be invoked by the driver program). Notice that using an operation of a component does not guarantee that a defect in its implementation will be detected. Certain operation combinations (e.g., `Initialize` followed by `Enqueue` followed by `Dequeue`, followed by `Enqueue`) might be required to uncover a defect [Zweben 89].

A drawback with layering `Display` (instead of directly implementing it) is that the layered version might make it difficult for the tester to determine which of the exported operations is defective. For example, suppose the tester uses the test driver to initialize and enqueue some values onto a `Queue` variable. At this point if the tester invokes the `Display` operation and the value displayed is incorrect, the tester might believe that `Enqueue` or `Initialize` is defective. But the defect could actually be located in `Dequeue` or `Test_If_Empty` since `Display` uses both of these operations. If `Display` were directly implemented (and known to be correct), then the tester in this example would know that the defect could not be in `Dequeue` or `Test_If_Empty` and had to be in `Initialize` or `Enqueue`. This is because a directly implemented `Display` would not use `Dequeue` or `Test_If_Empty`. For this reason, it might be beneficial to directly implement `Display`.

Another concern is that `Display` has to temporarily dismantle the `Queue`, and then reassemble it. If `Display` itself happens to have a defect, then the tester will probably spend time uncovering `Display`'s defects, or be misled into believing that the component's operations are defective. Even worse, a cancellation of errors might occur if both the component's operations and `Display` have defects. This could possibly prevent the tester from uncovering such defects. This drawback exists whether `Display` is layered or directly implemented. Fortunately, the implementation of `Display` usually turns out to be fairly straightforward, especially when it is layered. This is because `Display`'s implementer can reason in terms of the component's other abstract operations, instead of the operations of the internal representation type. In our experience, it is not hard to get `Display` "right."

Since `Display` is a useful operation to have for component testing, we introduce the following principles:

Principle 36 — For each type `T1`, ..., `Tn` exported by a component provide an extra capability called `Display` with the following interface:

```

procedure Display (
    x: in out Ti;           --! preserves
    indent: in Integer;    --! preserves
);
--! ensures "x's math model is output starting indent
--!          spaces right of the cursor position at the
--!          time of the call, leaving the cursor
--!          position at the end of the displayed value"

```

Principle 37 — Provide an interactive driver program — a "command interpreter" — for each *abstract* component. The driver should allow the user to invoke the component's `Display` operations and all the component's exported operations (except `Initialize_Package` and `Finalize_Package`) in any order. The driver program itself should invoke `Initialize_Package` and `Finalize_Package` at the appropriate times.

In the next section we illustrate how the Display operation can be utilized for component debugging.

3.2.2 Display of the Representation

In this section we introduce an operation that can help the component implementer debug the implementation during its development. In the previous section we discussed how displaying the abstract value of a variable is useful for uncovering defects. However, it may not be as useful for component debugging. This is because the abstract value displayed by Display factors out representational details that may be useful in finding the location of the defect. For example, in Queue_15, the representation type of a Queue variable is type List. Abstractly, List variables consist of two strings separated by a fence. It is this representational information that is (correctly) factored out by Display and is not seen by its user. However, it is this information that might help the component implementer find the location of a defect in Queue's package body. To help the implementer during component implementation, we propose the following principle:

Principle 38 — During the component implementation and debugging phase, for each type T1, ..., Tn exported by a component provide an operation called Display_Rep with the following interface:

```

procedure Display_Rep (
    x: in out Ti;           --! preserves
    indent: in Integer;    --! preserves
);
--! ensures "The math model of x's representation type
--! is output starting indent spaces right of
--! the cursor position at the time of the call,
--! leaving the cursor position at the end of
--! the displayed value"

```

At the completion of component implementation and debugging remove Display_Rep (and all related constructs) from the package specification and comment out its implementation (and all related constructs) in the package body.

Since Display_Rep must have access to the exported type's representation, it cannot be layered and must appear in the package specification. Furthermore, since Display_Rep displays its parameter in terms of the representation type's abstract value, it is necessary to add the representation type's Display operation to the generic parameter list. For example, the following is Queue_15's package specification after being modified to include Display_Rep:

```

--! concept Queue_Template

    generic
--! conceptual context

```

```

--!      uses
--!          STRING_THEORY_TEMPLATE

--!      conceptual parameters

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

--!      mathematics

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

--!      realization context

--!          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);

with procedure Display (
    s: in out List;           --! preserves
    indent: in Integer       --! preserves
);
--! ensures "s's math model is output
--!          starting indent spaces right of the cursor
--!          position at the time of the call, leaving
--!          the cursor position at the end of the
--!          displayed value"

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;           --! alters
        x: in out Item             --! consumes
    );
--! ensures "q = APPEND (#q, #x)"

    procedure Dequeue (
        q: in out Queue;           --! alters
        x: in out Item             --! produces
    );
--! requires "q /= EMPTY"
--! ensures "PREPEND (q, x) = #q"

```

```

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

procedure Display_Rep (
    q: in out Queue;           --! preserves
    indent: in Integer        --! preserves
);
--! ensures    "q's representation type's abstract
--!             value is output"

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

```

An implementation of Queue's Display_Rep operation might be:

```

procedure Display_Rep (
    q: in out Queue;
    indent: in Integer
) is
begin
    Display (q.rep, indent);
end Display_Rep;

```

Finally, at the end of the last section we mentioned that the Display operation can be utilized for component debugging. This section points out that this is an indirect use. Clearly, a component's Display operation is needed by its client's Display_Rep operation when the client programmer is developing the client implementation.

3.2.3 Display Model Components

In Section 3.2.1 we stated that we need to choose a particular method for displaying math objects, i.e., we need to establish conventions for how they are displayed. For example, for a string of items we might use the following convention: display a "<" (opening symbol) followed by each item on a new line, followed by a ">" (closing symbol). We need conventions so that all Display operations display their arguments in a uniform and consistent manner. However, if we rely on the implementer of each Display operation to follow the conventions, then there is a greater likelihood that they will not work uniformly and consistently. Furthermore, we will miss an opportunity for reuse.

For example, a Queue can be modeled as a string of items, and so can a Stack. By Principle 36, the implementer of each of these components must provide a Display capability for their respective types. Since both types are modeled by a string of items, they should both be displayed in the same manner. Here is where inconsistency can occur, and also where more reuse is possible. In this section, we discuss how to decouple the specific details of displaying a particular abstract mathematical object from the implementation of a Display operation. By decoupling these details, we can encapsulate the display conventions for a particular mathematical object in one place, and guarantee uniformity and consistency when they are displayed.

One possible way to encapsulate specific display conventions for a particular mathematical object is to embed those details in a component. For example, in Queue_15 we have modeled a queue as string of items, so we might create a component for constructing strings of items and for displaying them. Such a component might look like the following:

```

--! concept String_Model_Template

      generic
--!   conceptual context

--!     uses
--!       STRING_THEORY_TEMPLATE

--!     conceptual parameters

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

--!     mathematics

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

--!     realization context

--!     uses
--!       Two_Way_List_Template

--!     realization parameters

--!     facility Two_Way_List_Facility is
--!       Two_Way_List_Template (Item)
      type List is limited private;
      with procedure Initialize (
        l: in out List);
      with procedure Finalize (

```

```

        l: in out List);
with procedure Swap (
    l1: in out List;
    l2: in out List);
with procedure Move_To_Start (
    l: in out List);
with procedure Move_To_Finish (
    l: in out List);
with procedure Advance (
    l: in out List);
with procedure Retreat (
    l: in out List);
with procedure Add_Right (
    l: in out List;
    x: in out Item);
with procedure Remove_Right (
    l: in out List;
    x: in out Item);
with procedure Swap_Rights (
    l1: in out List;
    l2: in out List);
with procedure Test_If_At_Start (
    l: in out List;
    at_start: in out Boolean);
with procedure Test_If_At_Finish (
    l: in out List;
    at_finish: in out Boolean);

with procedure Display (
    x: in out Item;           --! preserves
    indent: in Integer       --! preserves
);
--! ensures "x's math model is output
--!           starting indent spaces right of the cursor
--!           position at the time of the call, leaving
--!           the cursor position at the end of the
--!           displayed value"

package String_Model_Template is
--! interface

    procedure Initialize_Package;
    procedure Finalize_Package;

--! type String_Model is modeled by STRING
--! exemplar s
--! initialization
--! ensures "s = EMPTY"
    type String_Model is limited private;
    procedure Initialize (
        s: in out String_Model);
    procedure Finalize (

```

```

        s: in out String_Model);
    procedure Swap (
        s1: in out String_Model;
        s2: in out String_Model);

    procedure Add_To_Left_End (
        s: in out String_Model;    --! alters
        x: in out Item             --! consumes
    );
--! ensures      "s = PREPEND (#s, #x)"

    procedure Add_To_Right_End (
        s: in out String_Model;    --! alters
        x: in out Item             --! consumes
    );
--! ensures      "s = APPEND (#s, #x)"

    procedure Remove_From_Left_End (
        s: in out String_Model;    --! alters
        x: in out Item             --! produces
    );
--! requires     "s /= EMPTY"
--! ensures      "#s = PREPEND (s, x)"

    procedure Remove_From_Right_End (
        s: in out String_Model;    --! alters
        x: in out Item             --! produces
    );
--! requires     "s /= EMPTY"
--! ensures      "#s = APPEND (s, x)"

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

    procedure Display (
        s: in out String_Model;    --! preserves
        indent: in Integer         --! preserves
    );
--! ensures      "s is output as a string of Items
--!              <...> starting indent spaces right of
--!              the cursor position at the time of the
--!              call, leaving the cursor position at
--!              the end of the displayed value"

private
    type String_Model is record

```

```

        rep: List;
    end record;
end String_Model_Template;
--! end String_Model_Template

```

Notice the following about `String_Model_Template`:

- It exports operations for constructing strings (`Add_To_Left_End` and `Add_To_Right_End`) and dismantling strings (`Remove_From_Left_End`, `Remove_From_Right_End`, and `Test_If_Empty`).
- It exports `Display`, for displaying the constructed strings of items.
- It is parameterized by type `Item`'s `Display` operation (which is used by the implementation of `String_Model_Template`'s `Display`).

The implementer of any `Display` operation that requires displaying strings of items (e.g., `Queue_Display_Capability`) can use `String_Model_Template` to construct and display its strings without knowing the details of how strings are displayed. For example, `Queue_Display_Capability` might be implemented as follows:

```

with Text_Io;          use Text_Io;
with Built_In_Types;  use Built_In_Types;
with Two_Way_List_Template;
with String_Model_Template;
package body Queue_Display_Capability is

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

package Two_Way_List_Facility is new
    Two_Way_List_Template (
        Item,
        Initialize,
        Finalize,
        Swap
    );

package String_Model_Facility is new
    String_Model_Template (
        Item,
        Initialize,
        Finalize,
        Swap,
        Two_Way_List_Facility.List,
        Two_Way_List_Facility.Initialize,
        Two_Way_List_Facility.Finalize,
        Two_Way_List_Facility.Swap,
        Two_Way_List_Facility.Move_To_Start,
        Two_Way_List_Facility.Move_To_Finish,

```

```

        Two_Way_List_Facility.Advance,
        Two_Way_List_Facility.Retreat,
        Two_Way_List_Facility.Add_Right,
        Two_Way_List_Facility.Remove_Right,
        Two_Way_List_Facility.Swap_Rights,
        Two_Way_List_Facility.Test_If_At_Start,
        Two_Way_List_Facility.Test_If_At_Finish,
        Display
    );

use String_Model_Facility;

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

procedure Convert_To_Model (
    q: in out Queue;                                --! consumes
    q_model: in out String_Model                    --! produces
) is
--!    ensures "q_model = #q"

    garbage: String_Model;
    empty: Boolean;
    x: Item;

begin
    Initialize (garbage);
    Initialize (empty);
    Initialize (x);

    Swap (q_model, garbage);
    loop
        Test_If_Empty (q, empty);
    exit when empty;
        Dequeue (q, x);
        Add_To_Right_End (q_model, x);
    end loop;

        Finalize (x);
        Finalize (empty);
        Finalize (garbage);
end Convert_To_Model;

-----

procedure Convert_To_Rep (
    q: in out Queue;                                --! produces
    q_model: in out String_Model                    --! consumes
) is
--!    ensures "q = #q_model"

    garbage: Queue;
    empty: Boolean;

```

```

        x: Item;
begin
    Initialize (garbage);
    Initialize (empty);
    Initialize (x);

    Swap (q, garbage);
loop
    Test_If_Empty (q_model, empty);
exit when empty;
    Remove_From_Left_End (q_model, x);
    Enqueue (q, x);
end loop;

    Finalize (x);
    Finalize (empty);
    Finalize (garbage);
end Convert_To_Model;

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

procedure Initialize_Package is
begin
    Two_Way_List_Facility.Initialize_Package;
    String_Model_Facility.Initialize_Package;
end Initialize_Package;

-----

procedure Finalize_Package is
begin
    String_Model_Facility.Finalize_Package;
    Two_Way_List_Facility.Finalize_Package;
end Finalize_Package;

-----

procedure Display (
    q: in out Queue;
    indent: in Integer
) is
    q_model: String_Model;
begin
    Initialize (q_model);

    Convert_To_Model (q, q_model);
    Display (q_model, indent);
    Convert_To_Rep (q, q_model);

    Finalize (q_model);
end Display;

```

```
end Queue_Display_Capability;
```

Other components for displaying mathematical model values include `Set_Model_Template`, `Tuple2_Model_Template`, `Tuple3_Model_Template`, etc. (Appendix A gives the package specification for each of these.)

Displaying mathematical functions presents a problem because in our specifications they are total; it might be difficult to display a function with an infinite domain. One convention for displaying a function is to use the `Set_Model_Template` and only “display” interesting domain-range pairs. However, this will not work for functions that have a very large domain and where all domain-range pairs are interesting, e.g., the identity function for integers. Currently, constructing the `Display` operation for this type of function has to be handled on a case-by-case basis, and is based on the particular function and what part of it needs to be seen by the user.

Finally, we present the following principle concerning the implementation of `Display` and `Display_Rep` operations:

Principle 39 — Whenever the implementation of `Display` or `Display_Rep` needs to display a mathematical model value, use existing model-displaying components (e.g., `String_Model_Template`) whenever feasible.

3.2.4 *Checking Component*

The behavior of an operation is not known or described when the conditions of the `requires` clause are not met, i.e., the `ensures` clause describes the behavior only when the conditions of the `requires` clause are met. An operation may exhibit benign behavior for a particular client if the client violates the operation’s `requires` clause (e.g., `Dequeue` might do nothing). The client in this situation is not correct, but if the violation of the `requires` clause does not cause at least some significant event to occur, the defect may go unnoticed during development and testing. For this reason, during testing of a client program, it is beneficial for the called operations to check their own `requires` clauses. This is not done as a means to protect the called operation from the client, but as a means to reveal when (if ever) a particular client is violating a called operation’s `requires` clause. This section describes how to create what is called a *checking component*. This is a layered version of a component whose operations check their own `requires` clauses and produce notification when there is a violation. It is generally to be used only during testing.

The main idea behind a checking component is that its exported syntactic and behavioral interface is exactly like the non-checking component that it is layered on, except when a client violates a `requires` clause of one of its operations. When a violation occurs, the checking component’s behavior, by convention, is defined and produces a notification (the non-checking version’s behavior is undefined). By having the same exported interface, the checking component can be substituted in the client for its non-checking counterpart, and vice versa, with no functional effect on a correct client. The checking component is substituted during testing of the client, and removed after testing has been finished. The tester is assured that the functional behavior of a correct client observed during testing

(using the checking version) is the same as it would have been had the non-checking component been used.

Fortunately, the construction of the checking component is quite easy. Therefore, it is usually easy to convince ourselves that its behavior is exactly the same as the non-checking version (when no requires clause is violated). Construction is easy because each component must export operations sufficient for a client to test all preconditions; see Principle 5, Section 2.1.5. Thus a checking component's implementation of an operation P (with a precondition) simply has to use the operations supplied by the non-checking version to check P's precondition. If the precondition is satisfied, then P just makes a call-through to the non-checking version's P. If the precondition is violated, then P produces a notification.

We can illustrate the design of a checking component using our Queue example since Queue_15's Dequeue has a precondition. The package specification might look like the following:

```

--! concept Queue_Template

    generic
    conceptual context

--!
    uses
--
        STRING_THEORY_TEMPLATE

--!
    conceptual parameters

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

--!
    mathematics

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

--!
    realization context

--!
    uses
--!
        Queue_Template

--!
    realization parameters

--!
    facility Queue_Facility is

```

```

--!
    Queue_Template (Item)
    type Queue1 is limited private;
    with procedure Initialize (
        q: in out Queue1);
    with procedure Finalize (
        q: in out Queue1);
    with procedure Swap (
        q1: in out Queue1;
        q2: in out Queue1);
    with procedure Enqueue (
        q: in out Queue1;
        x: in out Item);
    with procedure Dequeue (
        q: in out Queue1;
        x: in out Item);
    with procedure Test_If_Empty (
        q: in out Queue1;
        empty: in out Boolean);

--!
package Queue_Checking 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;           --! alters
        x: in out Item             --! consumes
    );
--!
    ensures "q = APPEND (#q, #x)"

    procedure Dequeue (
        q: in out Queue;           --! alters
        x: in out Item             --! produces
    );
--!
    requires "q /= EMPTY"
--!
    ensures "PREPEND (q, x) = #q"

```

```

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

private
    type Queue is record
        rep: Queue1;
    end record;
end Queue_Checking ;
--! end Queue_Template

```

Queue_Checking is just another implementation for the Queue_Template concept. Therefore, the only differences between Queue_15 and Queue_Checking occur in the realization context and in their names.

The following is an implementation of Queue_Checking. Notice that Initialize, Finalize, Swap, Enqueue and Test_If_Empty are implemented by a simple call-through to their respective Queue_15 operations. The most involved implementation is Dequeue, and even it is straightforward.

```

with Text_Io;           use Text_Io;
with Built_In_Types;   use Built_In_Types;
package body Queue_Checking is

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

    procedure Assert (
        b: Boolean;
        location: String;
        condition: String
    ) is
--! ensures "if b is False, an error message is displayed
--! indicating the 'location' of the violation and the
--! 'condition' that was violated (i.e., b); then the
--! built-in exception Program_Error is raised"

    begin
        if not b then
            New_Line;
            Put_Line ("*?!*?!*?!*?!*?!*?!*?!*?!*?!*?!*?!*?!*?!*?!*?!*?!");
            New_Line;
            Put (" ASSERTION VIOLATION AT: ");
            Put_Line (location);
            Put (" CONDITION VIOLATED: ");
            Put_Line (condition);
            New_Line;
            Put_Line ("*?!*?!*?!*?!*?!*?!*?!*?!*?!*?!*?!*?!*?!*?!*?!*?!");
            New_Line;
            raise Program_Error;
        end if;
    end;

```

```
    end if;  
end Assert;
```

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

```

    Enqueue (q.rep, x);
end Enqueue;

-----

procedure Dequeue (
    q: in out Queue;
    x: in out Item
) is
    empty: Boolean;
begin
    Initialize (empty);

    Test_If_Empty (q.rep, empty);
    Assert (not empty, "Dequeue", "q /= EMPTY");
    Dequeue (q.rep, x);

    Finalize (empty);
end Dequeue;

-----

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

end Queue_Checking;

```

Notice that procedure `Assert` is reusable among all checking components, and should be exported by a separate component. We recommend this approach but do not illustrate it here for brevity.

Instead of providing a checking component to be used by client programs during testing, we could leave checking of preconditions to each client program. There are at least two problems with this approach. First, an opportunity for reuse is missed when more than one client uses the component during its lifetime; the client programmer of each client would have to write code to check preconditions. Second, this approach requires that the client program be temporarily modified (during testing) at the point of the procedure call to check the called procedure's preconditions. Making these modifications and then removing them after testing is finished provides an opportunity for defects to be introduced into the client program. The checking component approach is less susceptible to this problem because the only change required to use it occurs where the client instantiates components. For these reasons, we introduce the following principle:

Principle 40 — Provide a checking component for each non-checking component that has an operation with a non-trivial precondition.

The main program that follows demonstrates the instantiation of Queue’s checking component and display capability (see Section 3.2.1). The following lists the changes required to make Queue_15_Test_Program use Queue_Facility instead of Queue_Checking_Facility:

- remove “with Queue_Checking,” marked by the line number “-- 1” in the right margin;
- remove Queue_Checking’s instantiation, marked by “--2”;
- remove “Queue_Checking_Facility.Initialize_Package,” marked by “--3”;
- remove “Queue_Checking_Facility.Finalize_Package,” marked by “--4”;
- perform a global search for “Queue_Checking_Facility” and replace it with “Queue_Facility.”

After this has been done, Queue_15_Test_Program would then use Queue_Facility everywhere.

```
with Built_In_Types;    use Built_In_Types;
with One_Way_List_Template_Fixed;
with Queue_15;
with Queue_Checking;           -- 1
with Queue_Display_Capability;
```

```
-----
procedure Queue_15_Test_Program is

  package One_Way_List_Facility is new
    One_Way_List_Template_Fixed (
      Integer,
      Initialize,
      Finalize,
      Swap
    );

  package Queue_Facility is new
    Queue_15 (
      Integer,
      Initialize,
      Finalize,
      Swap,
      One_Way_List_Facility.List,
      One_Way_List_Facility.Initialize,
      One_Way_List_Facility.Finalize,
```

```

        One_Way_List_Facility.Swap,
        One_Way_List_Facility.Move_To_Start,
        One_Way_List_Facility.Move_To_Finish,
        One_Way_List_Facility.Advance,
        One_Way_List_Facility.Add_Right,
        One_Way_List_Facility.Remove_Right,
        One_Way_List_Facility.Swap_Rights,
        One_Way_List_Facility.Test_If_At_Start,
        One_Way_List_Facility.Test_If_At_Finish
    );

package Queue_Checking_Facility is new -- 2
    Queue_Checking (
        Integer,
        Initialize,
        Finalize,
        Swap,
        Queue_Facility.Queue,
        Queue_Facility.Initialize,
        Queue_Facility.Finalize,
        Queue_Facility.Swap,
        Queue_Facility.Enqueue,
        Queue_Facility.Dequeue,
        Queue_Facility.Test_If_Empty
    );

package Queue_Display_Facility is new
    Queue_Display_Capability (
        Integer,
        Initialize,
        Finalize,
        Swap,
        Queue_Checking_Facility.Queue,
        Queue_Checking_Facility.Initialize,
        Queue_Checking_Facility.Finalize,
        Queue_Checking_Facility.Swap,
        Queue_Checking_Facility.Enqueue,
        Queue_Checking_Facility.Dequeue,
        Queue_Checking_Facility.Test_If_Empty,
        Display
    );

use Queue_Checking_Facility;
use Queue_Display_Facility;

b: Boolean
x: Integer;
q1, q2:Queue;

```

begin

```

One_Way_List_Facility.Initialize_Package;
Queue_Facility.Initialize_Package;
Queue_Checking_Facility.Initialize_Package;    -- 3
Queue_Display_Facility.Initialize_Package;

...

Queue_Display_Facility.Finalize_Package;
Queue_Checking_Facility.Finalize_Package;    -- 4
Queue_Facility.Finalize_Package;
One_Way_List_Facility.Finalize_Package;
end Queue_15_Test_Program;

```

3.3 Partial Instantiation

At the end of the previous section, we provided `Queue_15_Test_Program` to demonstrate the instantiation of `Queue_Display_Capability` and `Queue_Checking`. Another thing demonstrated is that fully-parameterized RESOLVE/Ada components require a lot of work by the client programmer to get them instantiated. For example, to instantiate `Queue_Checking`, we need an instance of `Queue_15`, but `Queue_15` is fully parameterized by `One_Way_List_Template`, so we have to instantiate it first.

At times it is convenient for client programmers to obtain and use components that are not fully parameterized. `Queue_15_Test_Program` also demonstrates this. Notice that the `One_Way_List_Template` that is used in `Queue_15_Test_Program` has the name `One_Way_List_Template_Fixed`. The only generic parameters required to instantiate it are for the conceptual type (i.e., Integer). `One_Way_List_Template_Fixed` is a *partial instantiation* of the fully-parameterized `One_Way_List_Template`. Think of a partial instantiation as a wrapper around a fully-parameterized component whose function is to supply some (or possibly all) of the parameters to the fully-parameterized component. In this example, `One_Way_List_Template_Fixed` has *fixed* (i.e., supplied) a realization parameter to `One_Way_List_Template`, but has not fixed the conceptual parameter.

In general, a component can be parameterized by both conceptual and realization generic parameters. There are three basic ways that a partial instantiation might fix these parameters: 1) fix some or all of the realization parameters but do not fix any of the conceptual parameters (`One_Way_List_Template_Fixed` is an example of this); 2) fix some or all of the conceptual parameters but do not fix any of the realization parameters; and 3) fix some or all of both the realization and conceptual parameters. One can imagine situations where each of these alternatives provides some convenience. It appears that in the situation where performance is not a key concern (e.g., prototyping), the first alternative (fix realization parameters) might prove to be the most beneficial. The third alternative (both fixed) might also be useful (e.g., a partial instantiation of `Queue_15` with Integer and a reasonable `One_Way_List_Template`, to obtain a Queue of Integers).

Each fully-parameterized component might have many partial instantiations. Just which kind (as discussed above) probably depends on the client base and the component. We imagine that after a fully-parameterized component's initial release, the set of partial instantiations will grow for some time and then slow after the most reasonable ones have been created and added to the component library.

For convenience, and based on the observation that client programmers in a prototyping phase might be the biggest customers of partial instantiations, we introduce the following principle:

Principle 41 — For each fully-parameterized component that has generic realization parameters, provide “reasonable” partial instantiations that fix some or all of these parameters. Add additional partial instantiations that are found to be convenient for particular components and client bases.

The following is the package specification for a partial instantiation for `Queue_15` where the realization parameter (`One_Way_List_Template`) is fixed. The partial instantiation should be viewed as another implementation of the concept provided by the fully-parameterized component. Therefore, each partial instantiation must provide the exact same exported interface as the fully-parameterized component on which it is layered. The only difference between a fully-parameterized component and one of its partial instantiations is in their realization context and their package names. Because of this, a client programmer can substitute a fully-parameterized component for a partial instantiation in the client program, or vice versa, simply by changing what is instantiated.

```
--! concept Queue_Template
--!
--!   realization context
--!
--!     uses
--!       One_Way_List_Template, Queue_Template
--!
--!     with One_Way_List_Template_Fixed;
--!     with Queue_15;
--!
--!   generic
--!   conceptual context
--!
--!     uses
--!       STRING_THEORY_TEMPLATE
--!
--!     conceptual parameters
--!
--!
--!   type Item
--!     type Item is limited private;
--!     with procedure Initialize (
--!       x: in out Item);
--!     with procedure Finalize (
--!       x: in out Item);
--!     with procedure Swap (
--!       x1: in out Item;
--!       x2: in out Item);
--!
--!   mathematics
```

```

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

package Queue_15_Fixed 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;           --! alters
  x: in out Item           --! consumes
);
--! ensures "q = APPEND (#q, #x)"

procedure Dequeue (
  q: in out Queue;           --! alters
  x: in out Item           --! produces
);
--! requires "q /= EMPTY"
--! ensures "PREPEND (q, x) = #q"

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

private
package List_Facility is new
  One_Way_List_Template_Fixed (
    Item,
    Initialize,
    Finalize,

```

```

        Swap
    );

    package Queue_Facility is new
        Queue_15 (
            Item,
            Initialize,
            Finalize,
            Swap,
            List_Facility.List,
            List_Facility.Initialize,
            List_Facility.Finalize,
            List_Facility.Swap,
            List_Facility.Move_To_Start,
            List_Facility.Move_To_Finish,
            List_Facility.Advance,
            List_Facility.Add_Right,
            List_Facility.Remove_Right,
            List_Facility.Swap_Rights,
            List_Facility.Test_If_At_Start,
            List_Facility.Test_If_At_Finish
        );

    type Queue is record
        rep: Queue_Facility.Queue;
    end record;
end Queue_15_Fixed;
--! end Queue_Template

```

The implementation of a partial instantiation is simple, using call-through for each of the operations. Here is Queue_15_Fixed's implementation:

```

package body Queue_15_Fixed is

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

use Queue_Facility;

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

procedure Initialize_Package is
begin
    List_Facility.Initialize_Package;
    Queue_Facility.Initialize_Package;
end Initialize_Package;

-----

procedure Finalize_Package is

```

```
begin
    Queue_Facility.Finalize_Package;
    List_Facility.Finalize_Package;
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
    Enqueue (q.rep, x);
end Enqueue;
```

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

```

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

end Queue_15_Fixed;

```

3.4 Summary

In Chapter 2 we introduced 33 principles for constructing the “standard” RESOLVE/Ada component. One might expect that often more than just the standard component is needed, e.g., we might need additional capabilities added to the standard component, or we might need a version for testing, etc. This chapter introduced eight new principles for constructing these various additions and versions.

Section 3.1 discussed the advantages and disadvantages of adding a new capability (an additional operation) to a component by layering or by direct implementation. An empirical pilot study reported in [Hollingsworth 91] suggests that there is potential for gain in productivity and quality if additional capabilities are layered instead of directly implemented. Therefore, the principles introduced in this section show a preference for layering over direct implementation, but do not rule out direct implementation. This section also pointed out how having the module-level conventions specified (discussed in Section 2.3.2) benefits the implementer of a directly implemented capability.

Section 3.2 discussed support for testing and debugging. During component testing, the tester needs to be able to easily determine the effects of executing a component’s exported operations. Since the component’s abstract behavior is specified in terms of a mathematical model, a good approach to communicating the effects of an operation is by displaying the values of the variables that the operation modifies in terms of their mathematical models. Furthermore, to facilitate testing, the tester needs to have available an interactive command interpreter which permits the tester to invoke all of the component’s exported operations and to see the effects of those operations. Section 3.2.1 introduced two principles requiring the component designer to supply both a Display operation and a command interpreter for each new component.

To further support testing, Section 3.2.4 introduced the checking component. For each standard RESOLVE/Ada component (i.e., those components whose operations do not check their preconditions) we ask that the component designer provide a checking version which does check its preconditions. The checking version’s behavior is exactly the same as its non-checking counterpart except when a precondition is violated. When a precondition is violated, the checking version produces a notification, while the non-checking version’s behavior is undefined. This checking version can be substituted for a non-checking version (and vice versa) with no effect on a correct client. The checking version aids testing of a client program by producing a notification detectable by the tester when the client program violates a precondition of a used component’s operation.

Fortunately, both the Display operation and checking component can be used during component implementation and debugging. When implementing and debugging a new

component which uses other components in its implementation, we can utilize the used components' Display operations and checking components. Section 3.2.2 introduced the Display_Rep operation which display the value of a variable in terms of its representation. The implementation of Display_Rep uses the Display operation of the representation type. If during implementation and debugging, the new component's implementation violates the preconditions of a used component's operation, a checking version of the used component will detect the violation and notify the new component's implementer.

Standard RESOLVE/Ada components are fully parameterized by conceptual types and by instances of the components used in their implementation. This often leads to components with many generic parameters, thereby making component instantiation tedious for the client programmer. Parameterizing a component by a facility used for its implementation gives the client more control of the component's performance. But when prototyping the functionality of a new system, for example, the client usually is not worried about performance issues. Therefore, it is often convenient for a component designer to provide a partial instantiation of a RESOLVE/Ada component whose realization parameters are fixed. Section 3.3 discussed partial instantiation of RESOLVE/Ada components from this standpoint.