Implementing Design Patterns in Ada95
Tips, Tricks, and Idioms
SIGAda 2002

Matthew Heaney

mheaney@on2.com

matthewjheaney@earthlink.net

http://home.earthlink.net/~matthewjheaney/

Interpreter

What's An Interpreter?

- Interpret sentences in a language specified by a grammar.
- Each production in the grammar is implemented as a type.
- As a lexical expression is parsed, an object is created for each production in the sentence.

Boolean Expression Grammar

```
<exp> ::= <and exp> | <or exp> |<not exp> |
          <var exp> | <const exp>

<and exp> ::= <exp> and <exp>

<or exp> ::= <exp> or <exp>

<not exp> ::= not <exp>

<var exp> ::= <name>

<const exp> ::= true | false
```

```
package Bool_Exps is

   type Bool_Exp (<>) is
     abstract tagged limited private;

   type Bool_Exp_Class_Access is
      access all Bool_Exp'Class;

   type Exp_Context is limited private;

   function Eval
     (Exp     : access Bool_Exp;
      Context : in     Exp_Context)
      return Boolean is abstract;

...
```

```
   function Copy
    (Exp : access Bool_Exp)
     return Bool_Exp_Class_Access is abstract;

   procedure Free
     (Exp : in out Bool_Exp_Class_Access);

private

    type Bool_Exp is
      abstract tagged limited null record;

    procedure Do_Free
      (Exp : access Bool_Exp);

end Bool_Exps;
```

The Need For Indirection

- Some expressions (`<and>`, `<or>`, `<not>`) contain other expressions.
- We don't know the size of the expression component (of type `Bool_Exp'Class`, which is indefinite), so we must refer to it indirectly.
- "Containment by reference" instead of "containment by value."

Desiderata

- We want to prevent a client from creating or destroying instances directly, in order to allow each type to define its own storage management policy.
- We want to minimize syntactic overhead (don’t want to have to explicitly dereference a pointer).

Implementation

- We always refer to Bool_Exp objects indirectly, via an access type. (This doesn't necessary imply heap, as instances might be allocated statically.)
- The type is limited, to prevent (shallow) copies.
- The type is indefinite, to prevent direct allocation.
- Each specific type in the class declares a constructor, so clients can create instances. (A client never calls allocator `new` directly.)

- Primitive operations take access parameters. Therefore, no explicit dereferencing is necessary.
- Each type declares its own private deconstructor, which performs type-specific clean-up prior to actual deallocation.
- The client reclaims storage for an object by explicitly calling a public, class-wide deconstructor, which is implemented by calling the type's private deconstructor.

```
package Bool_Exps.And_Exps is

   type And_Exp (<>) is
      new Bool_Exp with private;

   function New_And
     (L, R : access Bool_Exp'Class)
      return Bool_Exp_Class_Access;

   function Eval
     (Exp     : access And_Exp;
      Context : in     Exp_Context)
      return Boolean;
...
```

```
...
private

   type And_Exp is
     new Bool_Exp with record
        L, R : Bool_Exp_Class_Access;
     end record;

   procedure Do_Free
     (Exp : access And_Exp);

end Bool_Exps.And_Exps;
```

((True and X) or (Y and (not X)))

```
declare
   Exp : Bool_Exp_Class_Access :=
     New_Or
       (New_And (New_Const (True),
                 New_Var ('X')),
        New_And (New_Var ('Y'),
                 New_Not (New_Var ('X'))));

   Exp_Value : constant Boolean :=
      Eval (Exp, Context);
begin
  Free (Exp);  --explicit Free is req'd
end;
```

```
package body Bool_Exps.And_Exps is
   ...
   function Eval
     (Exp     : access And_Exp;
      Context : in     Exp_Context)
      return Boolean is
   begin
      if Eval (Exp.L, Context) then
         return Eval (Exp.R, Context);
      else
         return False;
      end if;
   end Eval;
```

```
package body Bool_Exps.And_Exps is

   type And_Exp_Access is
      access all And_Exp;

    function New_And
     (L, R : access Bool_Exp'Class)
      return Bool_Exp_Class_Access is

      Exp : constant And_Exp_Access :=
         new And_Exp;
   begin
      Exp.L := Bool_Exp_Class_Access (L);
      Exp.R := Bool_Exp_Class_Access (R);

      return Bool_Exp_Class_Access (Exp);
   end New_And;
```

Deallocation

- Client manually calls a class-wide Free operation to deallocate an expression object.
- Free can’t deallocate the (class-wide) object directly, because type-specific clean-up may be required ("grant last wishes").
- Free internally calls the private operation Do_Free, which dispatches according to the object’s tag. The type itself does the clean-up and actual deallocation.
- Free is an example of a “template method.”

```
package body Bool_Exps is
   ...
   procedure Free
     (Exp : in out Bool_Exp_Class_Access) is
   begin
      if Exp /= null then

         Do_Free (Exp); -- dispatches

         Exp := null;

      end if;
   end Free;

end Bool_Exps;
```

```
package body Bool_Exps.And_Exps is
  ...
     procedure Do_Free (Exp : access And_Exp) is

        procedure Deallocate is
           new Ada.Unchecked_Deallocation
            (And_Exp, And_Exp_Access);

        EA : And_Exp_Access := And_Exp_Access (Exp);

   begin

        Do_Free (Exp.L);
        Do_Free (Exp.R);

        Deallocate (EA);

   end Do_Free;

end Bool_Exps.And_Exps;
```

Constant Expressions

- The type Const_Exp has only two values: True and False.
- Because objects are referred to indirectly, multiple clients can share the same object, thus avoiding allocation of duplicate values.
- This is an example of the Flyweight pattern.

```
package Bool_Exps.Const_Exps is

   pragma Elaborate_Body;

   type Const_Exp (<>) is
     new Bool_Exp with private;

   ...

   function New_Const
     (Value : Boolean)
      return Bool_Exp_Class_Access;

   ...
```

```
package body Bool_Exps.Const_Exps is

   type Const_Exp_Array is
      array (Boolean) of aliased Const_Exp;

   Const_Exps : Const_Exp_Array;

   function New_Const
     (Value : Boolean)
      return Bool_Exp_Class_Access is
   begin
      return Const_Exps (Value)'Access;
   end;

   ...
```

```
...

begin

   for Value in Const_Exps'Range loop
      Const_Exps (Value).Value := Value;
   end loop;

end Bool_Exps.Const_Exps;
```

Smart Pointers

Motivation

- One issue with the Interpreter example is that deallocation of expression objects must be done manually by the client, by explicitly calling Free.
- This is an obvious source of memory leaks and dangling references.
- Besides being prone to error, explicit deallocation also carries a fair amount of syntactic overhead.

```
Perform_Mental_Gymnastics:
declare
   Replacement : Bool_Exp_Class_Access :=
     New_Not (New_Var ('Z'));

   Rep_Exp : constant Bool_Exp_Class_Access :=
     Replace (Exp, 'Y', Replacement);
begin
   Free (Replacement);
   Free (Exp);
   Exp := Rep_Exp;
end Perform_Mental_Gymnastics;
```

Desiderata

- Low syntactic overhead. Manipulation of smart pointers should be similar to regular access objects.
- By-reference semantics implies a reference-counting scheme.
- No explicit deallocation is ever required. Implies use of a Controlled type.

```
package Bool_Exps is

   type Bool_Exp (<>) is
     abstract tagged limited private;

   type Bool_Exp_Class_Access is
      access all Bool_Exp'Class;

   type Exp_Handle is private;

   function "+"
     (Handle : Exp_Handle)
      return Bool_Exp_Class_Access;

   function Null_Handle return Exp_Handle;
   ...
```

```
private

   type Bool_Exp is
     abstract tagged limited record
       Count : Natural;
     end record;

   type Exp_Handle is
     new Controlled with record
        Exp : Bool_Exp_Class_Access;
     end record;

   procedure Adjust (Handle : in out Exp_Handle)

   procedure Finalize (Handle : in out Exp_Handle);
```

((True and X) or (Y and (not X)))

```
declare
   Exp : constant Exp_Handle :=
     New_Or
       (New_And (New_Const (True),
                 New_Var ('X')),
        New_And (New_Var ('Y'),
                 New_Not (New_Var ('X'))));

   Exp_Value : constant Boolean :=
     Eval (+Exp, Context);
begin
   null;  --no explicit Free is req'd
end;
```

Without Smart Pointers

```
declare
   Replacement : Bool_Exp_Class_Access :=
     New_Not (New_Var ('Z'));

   Rep_Exp : constant Bool_Exp_Class_Access :=
     Replace (Exp, 'Y', Replacement);
begin
   Free (Replacement);
   Free (Exp);
   Exp := Rep_Exp;
end;
```

With Smart Pointers

```
declare
  Replacement : constant Exp_Handle :=
    New_Not (New_Var ('Z'));
begin
  Exp := Replace (+Exp, 'Y', Replacement);
end;
```

```
function Eval
  (Exp     : access And_Exp;
   Context : in     Exp_Context)
   return Boolean is
begin
   if Eval (+Exp.L, Context) then
      return Eval (+Exp.R, Context);
   else
      return False;
   end if;
end Eval;
```

Implementation

- A “smart pointer” is a non-limited type that privately derives from Controlled, and has an access object as its only component.
- It uses unary plus “+” to return the value of the internal access object, which is used immediately as the actual parameter in subprogram calls.

- The type designated by the access type has a Count component to store the number of references.
- When the reference count drops to zero (meaning there are no more references to the object), the designated object is automatically returned to storage.

Consequences

- Constructors now return Exp_Handle instead of Bool_Exp_Class_Access.
- Expression components (of `<and>`, `<or>`, and `<not>` expressions) are now of type Exp_Handle.
- Small syntactic penalty necessary to dereference handle object.

```
package Bool_Exps.And_Exps is

   type And_Exp (<>) is
      new Bool_Exp with private;

   function New_And
     (L, R : Exp_Handle)
      return Exp_Handle;

   function Eval
     (Exp     : access And_Exp;
      Context : in     Exp_Context)
      return Boolean;
   ...
```

```
   ...
private

   type And_Exp is
     new Bool_Exp with record
        L, R : Exp_Handle;
     end record;

   procedure Do_Free
     (Exp : access And_Exp);

end Bool_Exps.And_Exps;
```

Allocation

```
function New_And
   (L, R : Exp_Handle) return Exp_Handle is

    Exp : constant And_Exp_Access :=
      new And_Exp;
 begin
    Exp.Count := 1;
    Exp.L := L;
    Exp.R := R;

    return (Controlled with Exp.all'Access);
 end New_And;
```

Assignment

- During assignment, the private operation Adjust is called to increment the reference count of the object designated by the pointer.

```
package body Bool_Exps is

   ...

   procedure Adjust
     (Handle : in out Exp_Handle) is
   begin
      if Handle.Exp /= null then

         Handle.Exp.Count :=
            Handle.Exp.Count + 1;

      end if;
   end Adjust;

   ...
```

Deallocation

- When a smart pointer is assigned a new value, or goes out of scope, then the private operation Finalize is called.
- Finalize decrements the reference count of the designated object.
- If the reference count is zero, then Finalize also returns the object to storage, by calling (dispatching operation) Do_Free.

```
package body Bool_Exps is
   ...
   procedure Finalize
     (Handle : in out Exp_Handle) is
   begin
      if Handle.Exp /= null then

         Handle.Exp.Count :=
            Handle.Exp.Count - 1;

         if Handle.Exp.Count = 0 then
            Do_Free (Handle.Exp);
         end if;

      end if;
   end Finalize;
```

```
package body Bool_Exps.And_Exps is
   ...
   procedure Do_Free (Exp : access And_Exp) is

      EA : And_Exp_Access := And_Exp_Access (Exp);

      procedure Deallocate is
         new Ada.Unchecked_Deallocation
         (And_Exp, And_Exp_Access);
   begin
      pragma Assert (Exp.Count = 0);
      Exp.L := Null_Handle;
      Exp.R := Null_Handle;
      Deallocate (EA);
   end;

end Bool_Exps.And_Exps;
```

Dereferencing

- The “deference” operator (“+”) for the smart pointer has a trivial implementation: it simply returns the internal access value.
- A weakness of the whole approach is that it depends on clients never making a copy or otherwise manipulating the access value.
- Limited access types or garbage-collecting storage pools would be a helpful addition to the language.

```
package body Bool_Exps is
   ...
   function "+"
     (Handle : Exp_Handle)
      return Bool_Exp_Class_Access is
   begin
      return Handle.Exp;
   end;

   function Null_Handle
      return Exp_Handle is
   begin
      return (Controlled with null);
   end;
   ...
end Bool_Exps;
```

Multiple Views

```
package Persistence_Types is
   type Persistence_Type is
      abstract tagged limited null record;
   type Persistence_Class_Access is
      access all Persistence_Type'Class;

   procedure Write
     (Stream : in out Root_Stream_Type'Class;
      View   : access Persistence_Type) is abstract;
   procedure Read
     (Stream : in out Root_Stream_Type'Class;
      View   : access Persistence_Type) is abstract;
end Persistence_Types;
```

```
with Persistence_Types;  use Persistence_Types;

package Q is

   procedure Save
      (Persistence : access Persistence_Type'Class);

end Q;
```

```
package body Q is

   File : Ada.Streams.Stream_IO.File_Type;

   Stream : Root_Stream_Type'Class renames
      Stream_IO.Stream (File).all;

   procedure Save
     (Persistence : access Persistence_Type'Class) is
   begin
      Write (Stream, Persistence);
   end;

end Q;
```

```
with Persistence_Types;  use Persistence_Types;

package P is

   type T is limited private;

   function Persistence_View (O : access T)
     return Persistence_Class_Access;
   ...
private
   ...
```

```
 type Persistence_View_Type (O : access T) is
    new Persistence_Type with null record;

   type T is
      record
         Persistence : aliased Persist_View_Type (T'Access);
         ...
      end record;

   procedure Write
     (Stream : in out Root_Stream_Type'Class;
      View   : access Persistence_View_Type);
   procedure Read
     (Stream : in out Root_Stream_Type'Class;
      View   : access Persistence_View_Type);
end P;
```

```
with P;
with Q;

procedure Test_Persistence is
   O : aliased P.T;
begin
   Q.Save (Persistence_View (O'Access));
end;
```

Observer

Motivation

- It’s often the case that when the state changes in one object, another object needs to be notified of the change. There are a couple of ways of implementing this.
- The subject can know who its observers are by name, and tell them directly about the state change; or,

- The subject only knows that it’s being observed. It tells its observer that its state has changed, and then the observer queries the subject for the new state.
- A consequence of the former approach is that every time a new observer is added to the system, the subject must be modified to update yet another observer.
- The latter approach doesn’t suffer from this, because the observer just inserts itself into a list of anonymous observers.

```
package Subjects_And_Observers is

   type Subject is
     tagged limited private;

   procedure Notify
     (Sub : in out Subject'Class);

   type Observer is
     abstract tagged limited private;

   procedure Update
     (Obs : access Observer) is
      abstract;
   ...
```

Subjects_And_Observers

- Abstractions that wish to be observed derive from Subject. When the state changes, the abstraction calls Notify to let observers know about the change.
- Abstractions that wish to observe a subject derive from Observer, and Attach themselves to a subject. They must override Update, which is called by the subject during the notification.

```
   ...

   procedure Attach
     (Obs : access Observer'Class;
      To  : in out Subject);

   procedure Detach
     (Obs  : access Observer'Class;
      From : in out Subject);

private
   ...
end Subjects_And_Observers;
```

Subjects_And_Observers

- The subject type is implemented as a linked list of observers.
- When an observer wants to be notified of a state change in the subject, it places itself on the subject's list of observers.
- During a notification, the subject traverses the list, updating each observer in turn.

```
private

   type Observer_Class_Access is
     access all Observer'Class;

   type Subject is
     tagged limited record
        Head : Observer_Class_Access;
     end record;

   type Observer is
     abstract tagged limited record
        Next : Observer_Class_Access;
     end record;

end Subjects_And_Observers;
```

```
package body Subjects_And_Observers is

   procedure Notify
     (Sub : in out Subject'Class) is

      Obs : Observer_Class_Access := Sub.Head;

  begin

      while Obs /= null loop

         Update (Obs);

         Obs := Obs.Next;

      end loop;

   end Notify;
   ...
```

Clock_Timer Subject

- The subject Clock_Timer publicly derives from Subject, which allows it to be observed.
- Tick is the operation that updates the state of the clock timer, and then notifies any observers.
- Selector operations Get_Hour, Get_Minute, etc allow an observer to query the state.

```
package Clock_Timers is

   type Clock_Timer is
      new Subject with private;

   procedure Tick
     (Timer : in out Clock_Timer);

   subtype Hour_Number is
     Natural range 0 .. 23;

   function Get_Hour
     (Timer : Clock_Timer) return Hour_Number;

   ...
end Clock_Timers;
```

```
package body Clock_Timers is

   procedure Tick
     (Timer : in out Clock_Timer) is

   begin

      <update hour, min, sec attributes>

      Notify (Timer);  -- Update observers

   end Tick;
   ...
end Clock_Timers;
```

Alternate Technique

- Derive from Subject privately, and provide public operations to attach an observer.

```
package Clock_Timers is

   type Clock_Timer is private;

   procedure Attach
     (Obs : access Observer'Class;
      To  : in out Clock_Timer);
…
private

  type Clock_Timer is
    new Subject with record ...
```

Digital_Clock Observer

- The observer Digital_Clock derives (here, privately) from Observer type.
- The clock observer binds to its timer subject via an access discriminant. This guarantees that the (clock) subject lives at least as long as the observer, and therefore ensures that no dangling references from observer to subject can occur.

```
package Digital_Clocks is

   type Digital_Clock
     (Timer : access Clock_Timer'Class) is
       limited private;

   procedure Attach (Clock : access Digital_Clock);
   procedure Detach (Clock : access Digital_Clock);

private

  type Digital_Clock
    (Timer : access Clock_Timer'Class) is
      new Observer with null record;

   procedure Update
     (Clock : access Digital_Clock);

end Digital_Clocks;
```

- Attach (Detach) ensures that the Clock observer is properly attached to (detached from) the Timer subject to which it is bound via its access discriminant.
- Update is called by Notify, which is called by the Timer subject just after it (the subject) has changed its state.
- The Clock observer can “see” its Timer subject through its access discriminant. During the Update, the clock queries the state of the timer, and then displays the time in a format specific to that observer.

```
package body Digital_Clocks is

   procedure Attach (Clock : access Digital_Clock) is
   begin
      Attach (Clock, Clock.Timer.all);
   end;

   procedure Detach (Clock : access Digital_Clock) is
   begin
      Detach (Clock, Clock.Timer.all);
   end;
   ...
end Digital_Clocks;
```

```
package body Digital_Clocks is

   procedure Update
     (Clock : access Digital_Clock) is

      Hour : constant Hour_Number :=
         Get_Hour (Clock.Timer.all);

      Hour_Image : constant String :=
        Integer'Image (Hour);
      ...
      Clock_Image : constant String :=
        Hour_Image & ...;
   begin
      Put_Line (Clock_Image);
   end Update;
```

```
declare
   Timer : aliased Clock_Timer;
   Clock : aliased Digital_Clock (Timer'Access);
begin
   Attach (Clock’Access);
   Tick (Timer);
end;
```

Dynamic Observers

- You might have an application in which observers of a subject are added and removed dynamically.
- We need to automate calls to Attach and Detach, to ensure no dangling reference from subject to observer occurs.

```
declare
  Timer : aliased Clock_Timer;
begin
  …
  declare
     Clock : aliased Digital_Clock(Timer'Access);
  begin
     Attach (Clock'Access);
     Tick (Timer);
  end;  -- Oops!  Forget to Detach...

  Tick (Timer);  -- Notify non-existent observer!
end;
```

```
package Digital_Clocks is

   type Digital_Clock
    (Timer : access Clock_Timer'Class) is
     limited private;

private

   …

end Digital_Clocks;
```

Adding Controlled-ness

- We don’t really need to advertise that Digital_Clock derives from Observer, so we declare the partial view of the type as limited private, and implement the full view as a derivation.
- Controlled-ness is added as a component of the extension, because Ada doesn’t have multiple inheritance (and doesn’t need it).

```
private

   type Control_Type (Clock : access Digital_Clock) is
     new Limited_Controlled with null record;

   procedure Initialize (Control : in out Control_Type);

   procedure Finalize (Control : in out Control_Type);

   type Digital_Clock
       (Timer : access Clock_Timer'Class) is
     new Observer with record
        Control : Control_Type (Digital_Clock'Access);
     end record;

   procedure Update (Clock : access Digital_Clock);

end Digital_Clocks;
```

- During its initialization, the observer inserts itself on its subject's observer list.
- During its finalization, the observer removes itself from its subject's observer list. This guarantees that no dangling reference from subject to observer can occur, because removal is automatic when the lifetime of the observer ends.

```
package body Digital_Clocks is

   procedure Initialize (Control : in out Control_Type) is

      Clock : Digital_Clock renames Control.Clock.all;
      Timer : Clock_Timer renames Clock.Timer.all;
   begin
      Attach (Clock’Access, To => Timer);
   end;

   procedure Finalize (Control : in out Control_Type) is

      Clock : Digital_Clock renames Control.Clock.all;
      Timer : Clock_Timer renames Clock.Timer.all;
   begin
      Detach (Clock'Access, From => Timer);
   end;
…
```

```
declare
  Timer : aliased Clock_Timer;
begin
…
  declare
    Clock : Digital_Clock (Timer’Access);
    -- automatically Attach
  begin
    Tick (Timer);
  end; -- automatically Detach

  Tick (Timer);  -- OK
end;
```

Dynamic Observer Note

- This example was rather contrived, and was really designed to illustrate how to add Controlled-ness to an existing type hierarchy.
- Realistically, a dynamic observer would be declared on the heap. In that case, you could simply Attach in the constructor, and Detach in the deconstructor. A Controlled observer wouldn't be necessary.

```
package Digital_Clocks is

   type Digital_Clock(<>) is limited private;

   function New_Clock
     (Timer : access Clock_Timer)
     return Digital_Clock_Access;
...
private

   type Digital_Clock
     (Timer : access Clock_Timer) is
     new Observer with record ...;
```

```
package body Digital_Clocks is

   function New_Clock
     (Timer : access Clock_Timer)
     return Digital_Clock_Access is

     Clock : constant Digital_Clock_Access :=
        new Digital_Clock(Timer);
   begin

     Attach(Clock, To => Timer.all);

     return Clock;

   end;
```

Observers Observed

- We now introduce another variation of our original example, which allows an observer itself to be observed, by another observer.
- As before, a Digital_Clock observes a Clock_Timer. Here, we add another observer, a Clock_Watcher, to observe the Digital_Clock.

Digital_Clock

- The Digital_Clock must announce the fact that it can be observed, so it publicly derives from Subject.
- But it’s also an observer, so it binds to its Clock_Timer subject via an access discriminant.
- Like any subject, the Digital_Clock provides selector operations to allow its state to be queried by observers.

```
package Digital_Clocks is

   type Digital_Clock
     (Timer : access Clock_Timer'Class) is
     new Subject with private;

   type Meridian_Type is (AM, PM);

   function Get_Meridian (Clock : Digital_Clock)
     return Meridian_Type;

private
   …
```

Digital_Clock

- The Digital_Clock already derives from Subject, so in order to be an observer too it will have to have an Observer component.
- A helper type, `Timer_Obs_Type`, which derives from `Observer`, is used as the component.
- Here we also use a Controlled type to automatically Attach and Detach the observer. This wouldn't be necessary if you were to manually Attach to the subject.

```
   type Timer_Obs_Type (Clock : access Digital_Clock) is
     new Observer with null record;

   procedure Update (Timer_Obs : access Timer_Obs_Type);

   type Control_Type (Clock : access Digital_Clock) is
     new Limited_Controlled with record
        Timer_Obs : aliased Timer_Obs_Type (Clock);
     end record;

   procedure Initialize (Control : in out Control_Type);

   procedure Finalize (Control : in out Control_Type);

   type Digital_Clock (Timer : access Clock_Timer'Class) is
     new Subject with record
        Control  : Control_Type (Digital_Clock'Access);
        Meridian : Meridian_Type;
     end record;

end Digital_Clocks;
```

- The Control_Type can see its enclosing record (Digital_Clock) via its access discriminant.
- The Digital_Clock observer can see its Clock_Timer subject via its access discriminant.
- Together, these allow the Control_Type to Attach its Timer_Obs component to the Timer subject during Initialize, and Detach it during Finalize.

```
package body Digital_Clocks is

   procedure Initialize
     (Control : in out Control_Type) is
   begin
      Attach
        (Obs => Control.Timer_Obs'Access,
         To  => Control.Clock.Timer.all);
   end;

   procedure Finalize
     (Control : in out Control_Type) is
   begin
      Detach
        (Obs  => Control.Timer_Obs'Access,
         From => Control.Clock.Timer.all);
   end;
```

- As an observer, the Clock_Timer (really, the Timer_Obs_Type) must provide an implementation of Update.
- Update displays the new time (plays its observer role), then updates its own state and Notify's its own observers (plays its subject role).
- This organization has the effect of propagating a signal all the way back from the ultimate subject to the ultimate observer.

```
   procedure Update
      (Timer_Obs : access Timer_Obs_Type) is
   begin

      <get time from Timer_Obs.Clock.Timer.all>
      <display new time>

      if Hour < 12 then
         Timer_Obs.Clock.Meridian := AM;
      else
         Timer_Obs.Clock.Meridian := PM;
      end if;

      Notify (Timer_Obs.Clock.all);

   end Update;

end Digital_Clocks;
```

Clock_Watcher

- A very simple observer that observes a Digital_Clock.
- Per the idiom, it binds to its subject via an access discriminant.
- Here we manually Attach and Detach to the subject, instead of using Controlled-ness to do it automatically.

```
package Clock_Watchers is

   type Clock_Watcher
      (Clock : access Digital_Clock'Class) is
     limited private;

   procedure Start_Watching_Clock
     (Watcher : access Clock_Watcher);

   procedure Stop_Watching_Clock
    (Watcher : access Clock_Watcher);

private

   type Clock_Watcher
      (Clock : access Digital_Clock'Class) is
     new Observer with null record;

   procedure Update (Watcher : access Clock_Watcher);

end Clock_Watchers;
```

```
package body Clock_Watchers is

   procedure Start_Watching_Clock
     (Watcher : access Clock_Watcher) is
   begin
      Attach (Watcher, To => Watcher.Clock.all);
   end;


   procedure Stop_Watching_Clock
    (Watcher : access Clock_Watcher) is
   begin
      Detach (Watcher, From => Watcher.Clock.all);
   end;

   ...
```

```
   ...
   procedure Update
     (Watcher : access Clock_Watcher) is
   begin
      case Get_Meridian (Watcher.Clock.all) is
         when AM =>
            Put_Line ("It's still morning.");
         when PM =>
            Put_Line ("It's afternoon.");
      end case;
   end Update;

end Clock_Watchers;
```

```
declare
   Timer : aliased Clock_Timer;
   Clock : aliased Digital_Clock (Timer'Access);
   Watcher : aliased Clock_Watcher (Clock'Access);
begin
   Start_Watching_Clock (Watcher);
   Tick (Timer);
end;
```

Observable-Observer Note

- There is another way to allow an observer to be both an observer and a subject.
- Simply change the declaration of Observer type in package Subjects_And_Observers so that it derives from Subject.
- Implementing the observing subject with an observer component isn’t necessary, because the type is already an observer.

```
package Subjects_And_Observers is

   type Subject is tagged limited private;

   ...

   type Observer is
     abstract new Subject with private;

   ...

end Subjects_And_Observers;
```

```
package Digital_Clocks is

   type Digital_Clock (Timer : access C_Timer'Class) is
     new Subject with private;
   ...
private

   type Control_Type (Clock : access Digital_Clock) is
     new Limited_Controlled with null record;
   ...
   type Digital_Clock (Timer : access C_Timer'Class) is
     new Observer with record
        Control  : Control_Type (D_Clock'Access);
        Meridian : Meridian_Type;
     end record;

   procedure Update (Clock : access Digital_Clock);

end Digital_Clocks;
```

Observing Multiple Subjects

- We introduce yet another variation of the observer pattern, this time allowing an observer to observe multiple subjects.
- Now the digital clock simultaneously observes both a timer and a battery. The battery subject notifies its observer when it is drained or charged.

```
package Batteries is

   type Battery_Type is
     new Subject with private;

   procedure Charge (...);

   procedure Drain (...);

   function Is_Low (...) return Boolean;

private

   type Battery_Type is
     new Subject with record
        State : Positive := 1;
     end record;
```

Battery Subject (spec)

- The battery is observable, asserting this by publicly deriving from Subject.
- Modifier operations Charge and Drain adjust the available energy, and then Notify any observers.
- A selector operation, Is_Low, queries whether there is any energy remaining.

```
package body Batteries is

   procedure Charge
     (Battery : in out Battery_Type) is
   begin
      Battery.State := 1;
      Notify (Battery);
   end;

   procedure Drain
     (Battery : in out Battery_Type) is
   begin
      Battery.State := Battery.State + 1;
      Notify (Battery);
   end;

   function Is_Low
     (Battery : in Battery_Type) return Boolean is
   begin
      return Battery.State > 3;
   end;
```

Digital_Clock

- The public part of the observer type Digital_Clock has been modified to accept two access discriminants, one for each subject it observes.

```
package Digital_Clocks is

   type Digital_Clock
     (Timer   : access Clock_Timer'Class;
      Battery : access Battery_Type'Class) is
     limited private;

private
   ...
end Digital_Clocks;
```

Digital_Clock

- There has to be some type that derives from Observer and overrides Update to process Clock_Timer notifications.
- There has to be some type that derives from Observer and overrides Update to process Battery_Type notifications.
- The same type can’t do both, because we don’t have multiple inheritance in Ada95. No problem, we just use the “multiple views” idiom.

```
private

   type Timer_Obs_Type
     (Clock : access Digital_Clock) is
     new Observer with null record;

   procedure Update
     (Observer : access Timer_Obs_Type);

   type Battery_Obs_Type
     (Clock   : access Digital_Clock) is
     new Observer with null record;

   procedure Update
     (Observer : access Battery_Obs_Type);

   ...
```

```
   ...
    type Digital_Clock
      (Timer   : access Clock_Timer'Class;
       Battery : access Battery_Type'Class) is
      new Limited_Controlled with record

         Timer_Obs :
            aliased Timer_Obs_Type (D_Clock'Access);

         Battery_Obs :
            aliased Battery_Obs_Type (D_Clock'Access);

      end record;

    procedure Initialize (Clock : in out Digital_Clock);

    procedure Finalize (Clock : in out Digital_Clock);

end Digital_Clocks;
```

- An internal type, Timer_Obs_Type, observes just the Clock_Timer.
- Another internal type, Battery_Obs_Type, observes just the Battery_Type.
- Each type is bound to its enclosing record, the Digital_Clock, via an access discriminant.
- These internal types will be used to declare the observer components of the Digital_Clock type, which itself already derives from Limited_Controlled.

```
package body Digital_Clocks is

   procedure Update
     (Observer : access Timer_Obs_Type) is …;

   procedure Update
     (Observer : access Battery_Obs_Type) is

      Clock : Digital_Clock renames
        Observer.Clock.all;

      Battery : Battery_Type'Class renames
        Clock.Battery.all;
   begin
      if Is_Low (Battery) then ...
   end Update;
```

```
procedure Initialize
  (Clock : in out Digital_Clock) is

begin
   Attach (Obs => Clock.Timer_Obs'Access,
           To  => Clock.Timer.all);

   Attach (Obs => Clock.Battery_Obs'Access,
           To  => Clock.Battery.all);
end Initialize;
```

Observing Multiple Attributes

- One issue is that when a subject notifies an observer that a state change has occurred, the observer has no way of knowing which specific attribute has changed.
- This may require the observer to redo all its processing (say, redraw a window), which may be inefficient.

- One solution is to make observation more fine-grained; that is, to be able to observe individual attributes of a object, instead of just one monolithic object.
- When an object being observed changes the value of an attribute, it can notify the observers of that one attribute.
- It’s analogous to observing multiple subjects, but here, all the subjects are part of a single object.

```
package Clock_Timers is

   type Clock_Timer is limited private;
   ...
   subtype Hour_Number is
     Natural range 0 .. 23;

   function Get_Hour
     (Timer : access Clock_Timer)
      return Hour_Number;

   function Get_Hour_Subject
     (Timer : access Clock_Timer)
      return Subject_Class_Access;
```

```
   ...
private

   type Clock_Timer is
     limited record
        Hour           : Integer := -1;
        Hour_Subject   : aliased Subject;
        Minute         : Integer := -1;
        Minute_Subject : aliased Subject;
        Second         : Integer := -1;
        Second_Subject : aliased Subject;
     end record;

end Clock_Timers;
```

```
package body Clock_Timers is

   procedure Tick
     (Timer : in out Clock_Timer) is
   begin

      <update time>

      if Timer.Hour /= Hour then

         Timer.Hour := Hour;

         Notify (Timer.Hour_Subject);

      end if;
      ...
   end Tick;
```

```
function Get_Hour_Subject
  (Timer : access Clock_Timer)
   return Subject_Class_Access is
begin
   return Timer.Hour_Subject'Access;
end;
```

```
package Digital_Clocks is

   type Digital_Clock (Timer : access Clock_Timer) is
      limited private;

private

   type H_Obs_Type (Timer : access Clock_Timer) is
     new Observer with null record;

   procedure Update
     (H_Obs : access H_Obs_Type);
   ...
   type Digital_Clock (Timer : access Clock_Timer) is
     new Limited_Controlled with record
         H_Obs : aliased H_Obs_Type (Timer);
         ...
     end record;

   ...
```

```
package body Digital_Clocks is

   procedure Update
     (H_Obs : access H_Obs_Type) is

      Image : constant String :=
        Integer'Image (Get_Hour (H_Obs.Timer) + 100);
   begin
      <display hour>
   end;


   procedure Initialize
      (Clock : in out Digital_Clock) is
   begin

      Attach
        (Obs => Clock.H_Obs'Access,
         To  => Get_Hour_Subject (Clock.Timer));
```

Factory Method

Motivation

- Suppose we have a family of stack types, and we want to provide a class-wide operation to print a stack.
- We plan on using an active iterator to implement the operation. Each type in the class has its own iterator.
- Here's the problem: if the stack parameter has a class-wide type, then how do we get an iterator that works for *this* stack object?

```
procedure Stacks.Put
  (Stack : in Root_Stack_Type'Class) is

   Iterator : <what's its type?> :=
     <how do we get one for Stack's type?>
begin
   while not Is_Done (Iterator) loop
   ...
```

What's A Factory Method?

- If you need an iterator for this type of stack, then just *ask the stack* for one.
- A “factory method” is a constructor that dispatches on one type, and returns a value of some other type.
- In Ada95, the return type has to be class–wide, since an operation can only be primitive for one type.

```
generic
   type Item_Type is private;
package Stacks is

  type Root_Stack_Type is
     abstract tagged limited null record;

  type Root_Iterator_Type is
    abstract tagged null record;

  -- Here's the factory method:
  --
  function Start_At_Top
    (Stack : Root_Stack_Type)
     return Root_Iterator_Type'Class is
     abstract;
```

```
procedure Stacks.Put
  (Stack : in Root_Stack_Type'Class) is

   Iterator : Root_Iterator_Type'Class :=
     Start_At_Top (Stack);
begin
   while not Is_Done (Iterator) loop

     … Get_Item (Iterator) …

     Advance (Iterator);

   end loop;

   New_Line;
end Stacks.Put;
```

```
generic

   Max_Depth : in Positive;

package Stacks.Bounded_G is

   type Stack_Type is
     new Root_Stack_Type with private;

   type Iterator_Type is
      new Root_Iterator_Type with private;

   function Start_At_Top
     (Stack : Stack_Type)
      return Root_Iterator_Type'Class;
```

Copying A Stack

- Requires care, because it's easy to populate the target stack in reverse order.
- You can either (1) traverse the items in the source stack in bottom-to-top order, and populate the target stack in the normal way (using Push); or,
- You can (2) traverse the items in the source stack in top-to-bottom order, and populate the target stack in reverse order, using a special operation (like Copy).

```
procedure Copy_That_Does_Not_Work
  (From : in     Root_Stack_Type'Class;
   To   : in out Root_Stack_Type'Class) is

   Iter : Root_Iterator_Type'Class :=
      Start_At_Top (From);
begin
   if <From and To are the same stack> then
      return;
   end if;

   Clear (To);
   while not Is_Done (Iter) loop
      Push (Get_Item (Iter), On => To);
      Advance (Iter);
   end loop;
end Copy_That_Does_Not_Work;
```

```
procedure Stacks.Copy  -- technique (1)
  (From : in     Root_Stack_Type'Class;
   To   : in out Root_Stack_Type'Class) is

   Iterator : Root_Iterator_Type'Class :=
      Start_At_Bottom (From);
begin
   if From'Address = To'Address then
     -- per RM95 3.10 (9) and 13.3 (16)
     return;
   end if;

   Clear (To);
   while not Is_Done (Iterator) loop
      Push (Get_Item (Iterator), On => To);
      Backup (Iterator);
   end loop;
end Stacks.Copy;
```

```
package body Stacks.Bounded_G is

   procedure Copy -- technique (2)
      (From : in     Root_Stack_Type'Class;
       To   : in out Stack_Type) is

      Depth : constant Natural :=
        Get_Depth (From);

      Iterator : Root_Iterator_Type'Class :=
        Start_At_Top (From);

      use type System.Address;
   begin
      ...
```

```
   ...
   if From'Address = To'Address then
      return;
   end if;

   if Depth > Max_Depth then
      raise Storage_Error;
   end if;

   To.Top := Depth;

   for I in reverse 1 .. Depth loop
      To.Items (I) := Get_Item (Iterator);
      Advance (Iterator);
   end loop;
end Copy;
```

Summary of Stack Copying

- Technique (1) requires that you be able to traverse stacks in reverse order.
- Technique (1) can be implemented as a class-wide operation, or as a primitive operation with a default implementation.
- Technique (2) must be implemented as a primitive operation, for each type, because it needs to know the type's representation.