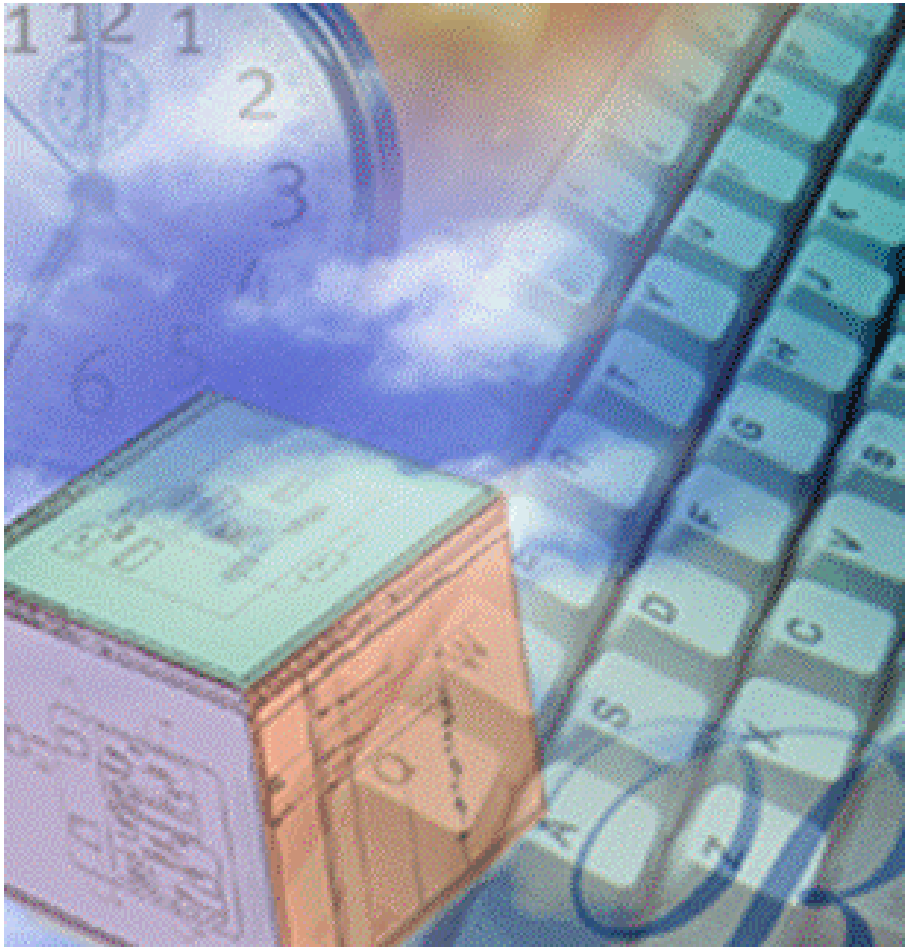
UML -> Rhapsody in Ada



An Example

Object Orientated Ada

- Ada is not fully OO
- Key Concepts:
  - Separate external and internal views
  - Encapsulation

  Ada 83

  - Classification
  - Inheritance
  - Dynamic Dispatching (Polymorphism)

  Ada 95

I-Logix

Code Generation

- RULES BASED APPROACH
  - The Auto-Code Generation is achieved through an external Code Generator
  - The Code Generator has a set of rules to map UML concepts to Ada Code
  - Every single token in the code is customisable
  - The User can maintain several rulesets (eg Ada95, Ada83, Spark)
  - The Rules Editor is Java-Based and WYSIWYG

I-Logix

The Transformation Engine

UNIFIED MODELING LANGUAGE

Code Generation

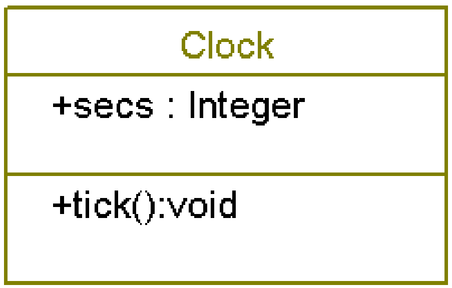
Roundtripping

Reverse Engineering

```
package Waveform is

   --Declared types ---------------------------
   type Waveform_t is abstract tagged null record;

   type Waveform_acc_t is access Waveform_t'Class;
```


I-Logix

UML to Ada → Classes




```
--++ class Clock
package Clock is

  --Declared types ---------------------------
  type Clock_t is tagged

  record

    -- Fields --
    secs : Integer; --++ attribute secs

  end record;

  type Clock_acc_t is access Clock_t;

  --Variables ---------------------------

  --Functions/Procedures section ------------
  --++ operation tick()
  procedure tick (this : in out Clock_t);
```

Class

Class Attributes

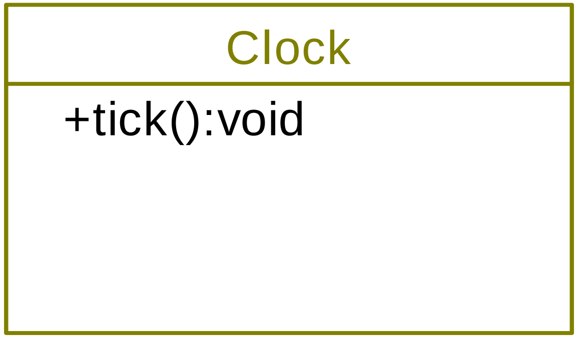
Class Pointer

'this' Pointer

For Ada83, the SW_t would be just a record – not a tagged type – so avoid Inheritance if you want the model to remain Ada83 compatible

The ‘this’ Pointer

- Class_t is automatically added as the first argument to every operation (The ‘this’ pointer)

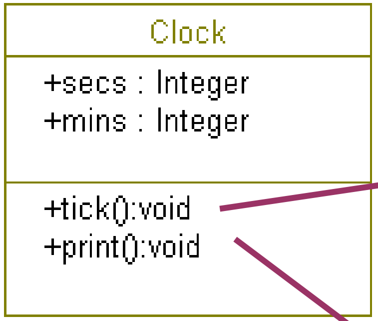



```
--Functions/Procedures section --------------------
--++ operation tick()
procedure tick (this : in out Clock_t);
```

A Simple Clock

- What has a clock got to do?
  - Maintain minutes, seconds
  - Implement timing algorithm (so we don't get 61 seconds)
  - Display the time

The Clock Class




```
procedure tick (this : in out Clock_t) is
begin
   --+[ operation tick()
   if this.secs = 59 then
       this.secs :=0;
       this.mins := this.mins +1;
   else
       this.secs := this.secs +1;
   end if;
   --+]
end tick;
```

```
procedure print (this : in out Clock_t) is
begin
   --+[ operation print()
   ada.text_io.put("Time is: ");
   ada.integer_text_io.put(this.mins);
   ada.text_io.put(":");
   ada.integer_text_io.put(this.secs);
   ada.text_io.new_line;
   --+]
end print;
```


Instantiating a Class

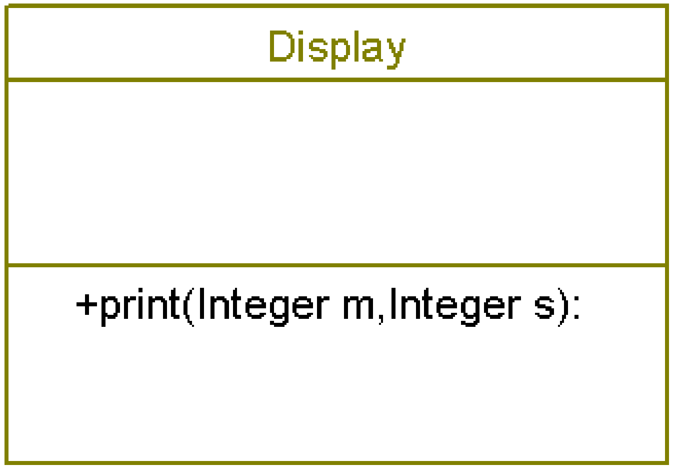
Clock

```
type Clock_acc_t is access Clock_t;
```

THE MAIN

```
procedure MainTest is
  p_Clock : Clock.Clock_acc_t;
begin
  -- Instance Initialization
  p_Clock := new Clock.Clock_t;

 for n in 1..100 loop
 delay(0.1);
 Clock.tick(p_Clock.all);
 Clock.print(p_Clock.all);
 end loop;
```

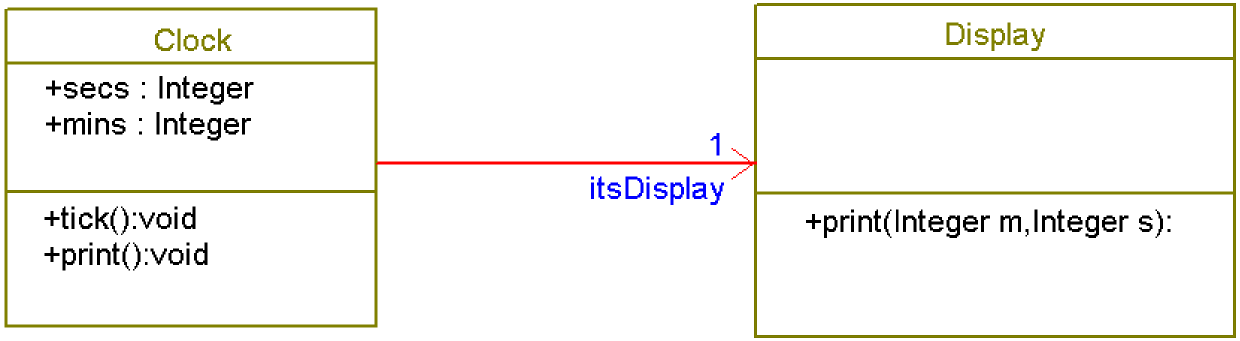
The Display Class




```
package Display is

   --Declared types -----------------------------
   type Display_t is tagged null record;

   type Display_acc_t is access Display_t;
```

```
--Functions/Procedures section -----------
procedure print (this : in out Display_t;
   m : in Integer;
   s : in Integer
) is
begin
   --+[ operation print(Integer, Integer)
   ada.text_io.put("Time is: ");
   ada.integer_text_io.put(m);
   ada.text_io.put(":");
   ada.integer_text_io.put(s);
   ada.text_io.new_line;
   --+]
end print;
```

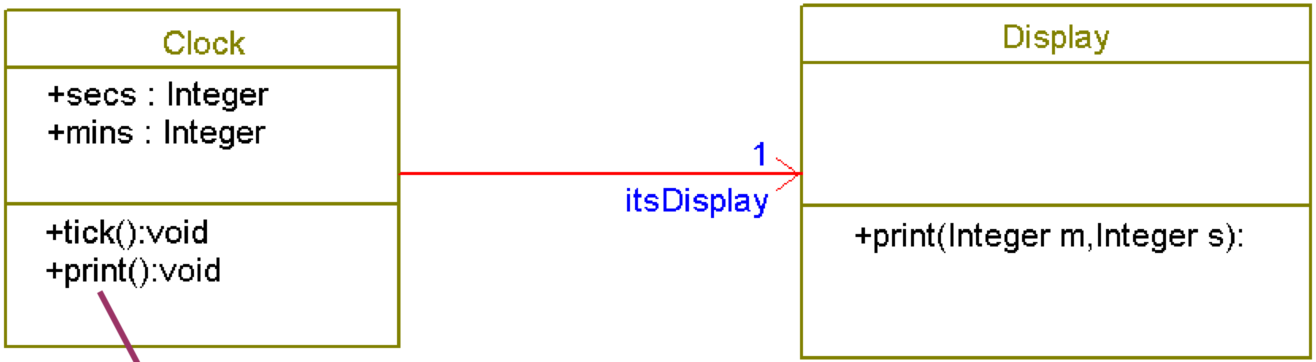
Relationships




```
type Display_acc_t is access Display_t;
```

```
--++ class Clock
package Clock is

  --Declared types ---------------------------
  type Clock_t is tagged

  record

    -- Relations --
    itsDisplay : Display.Display_acc_t := null;

    -- Fields --
    secs : Integer; --++ attribute secs
    mins : Integer; --++ attribute mins

  end record;
```

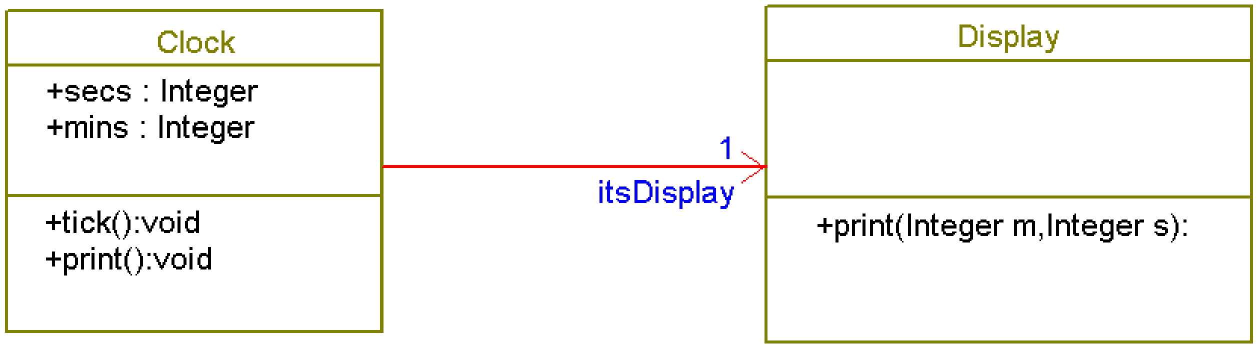
Creating a NEW Instance




```
procedure print (this : in out Clock_t) is
begin
   --+[ operation print()
   if this.itsDisplay = null then
       --No Instance - Create New One
       this.itsDisplay := new Display.Display_t;
       Display.print(this.itsDisplay.all,this.mins,this.secs);
   else
       Display.print(this.itsDisplay.all,this.mins,this.secs);
   end if;
   --+]
end print;
```

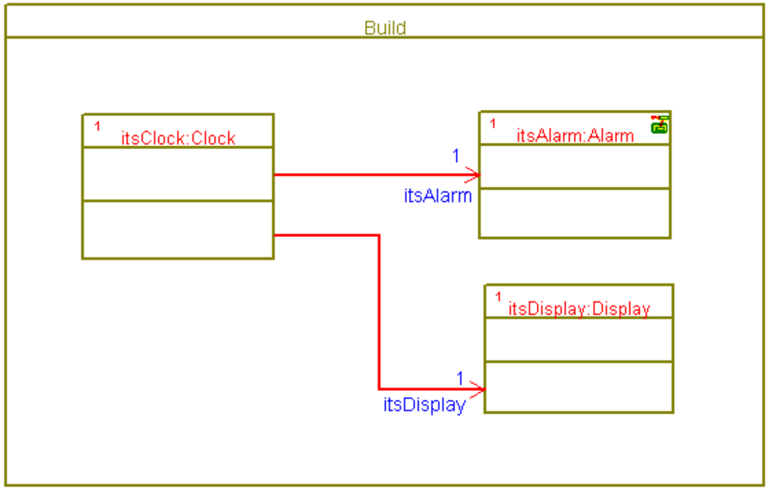

I-Logix

Accessors / Mutators




```
procedure get_itsDisplay(this : in out Clock_t; result : out Display.Display_acc_t) is
begin
  result := this.itsDisplay;
end get_itsDisplay;

procedure set_itsDisplay(this : in out Clock_t; value : in Display.Display_acc_t) is
begin
  this.itsDisplay := value;
end set_itsDisplay;
```

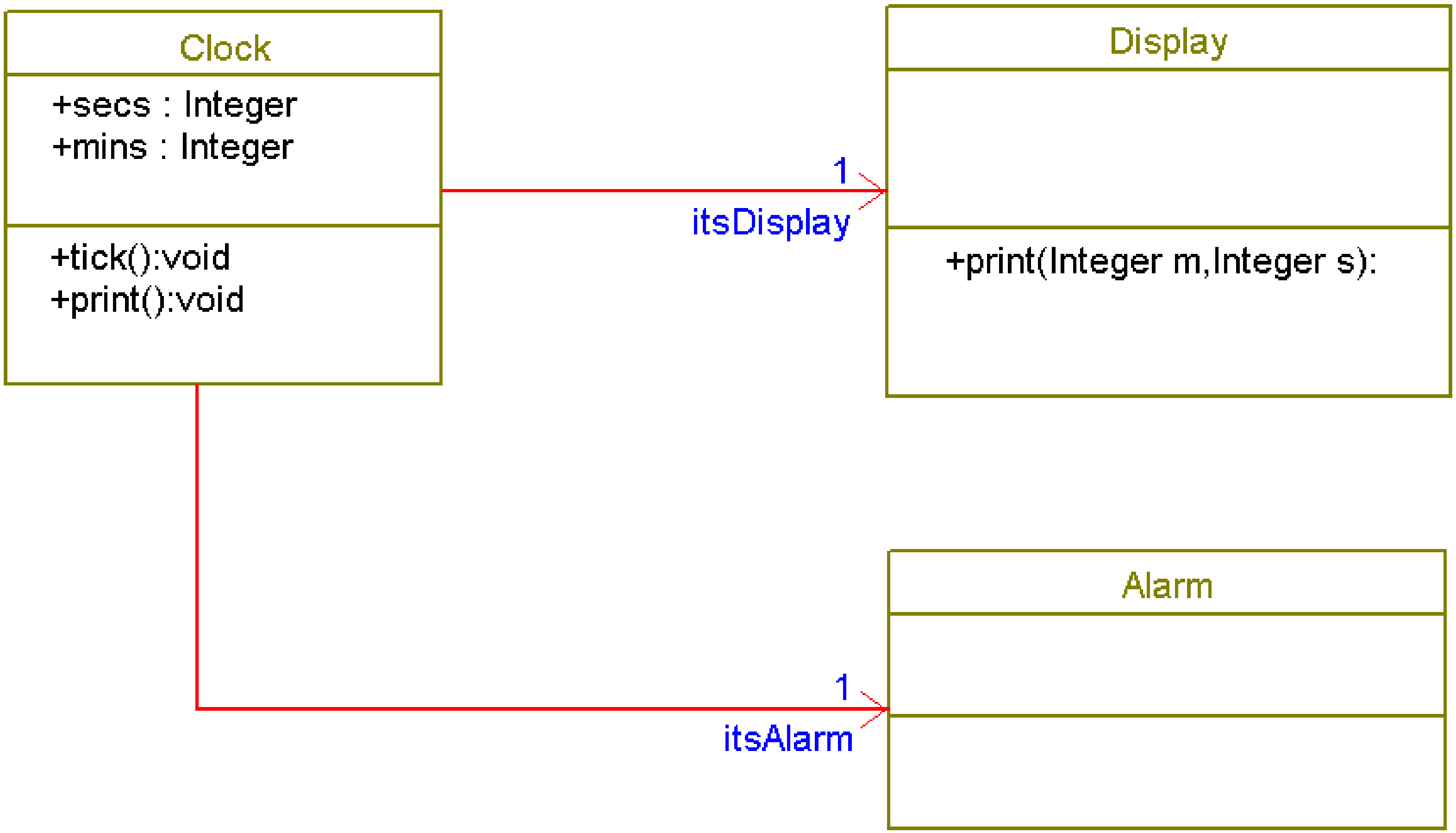
Composite Classes




```
package Build is

  --Declared types ----------------------------
  type Build_t is tagged

  record

     -- Relations --
     itsClock : Clock.Clock_acc_t := null;
     itsDisplay : Display.Display_acc_t := null;
     itsAlarm : Alarm.Alarm_acc_t := null;

  end record;
```

```
--++ class Build
package body Build is

  --Functions/Procedures section --------------------
  procedure Initialize (this : in out Build_t) is
  begin
        this.itsClock := new Clock.Clock_t;
        this.itsDisplay := new Display.Display_t;
        this.itsAlarm := new Alarm.Alarm_t;
        Clock.set_itsDisplay(this.itsClock.all, this.itsDisplay);
        Clock.set_itsAlarm(this.itsClock.all, this.itsAlarm);
```

Alarm Function



I-Logix

Wake me up at 1.00

Clock

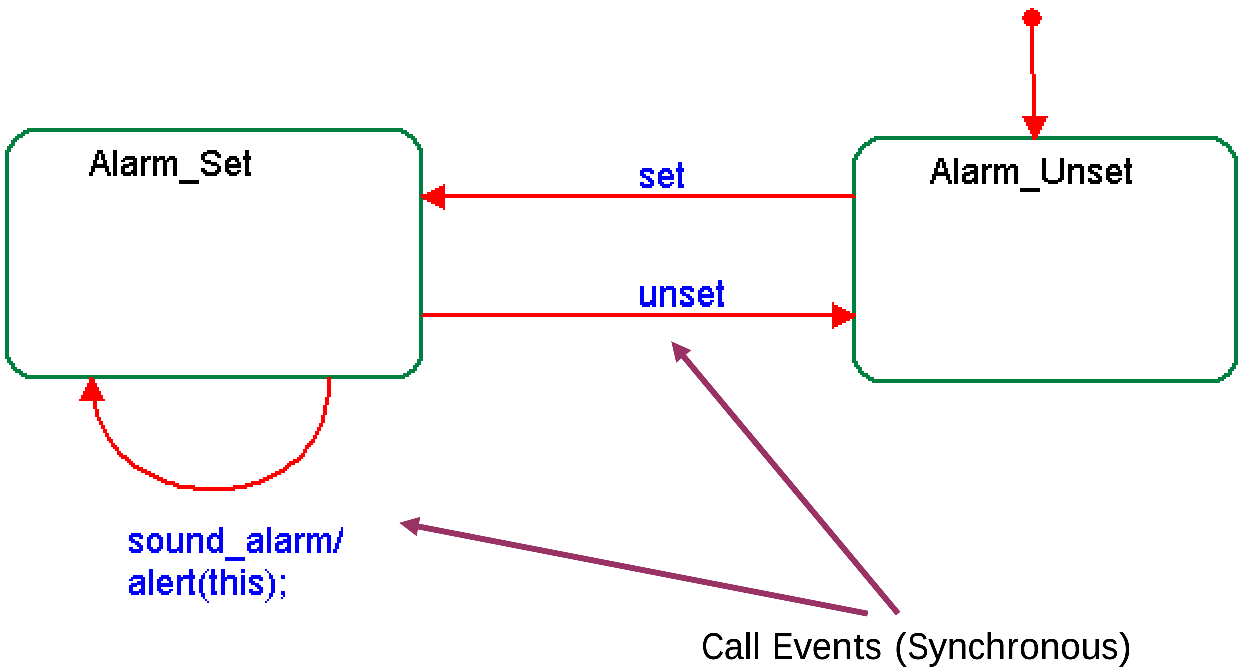
+secs : Integer
+mins : Integer

+tick():void
+print():void

```
--Functions/Procedures section ----------------------
procedure tick (this : in out Clock_t) is
begin
  --+[ operation tick()
  if this.secs = 59 then
      this.secs :=0;
      this.mins := this.mins +1;
  else
      this.secs := this.secs +1;
  end if;

  --Alarm?

  if this.mins = 1 and this.secs = 0 then
      Alarm.Alert(this.itsAlarm.all);
  end if;
  --+]
end tick;
```

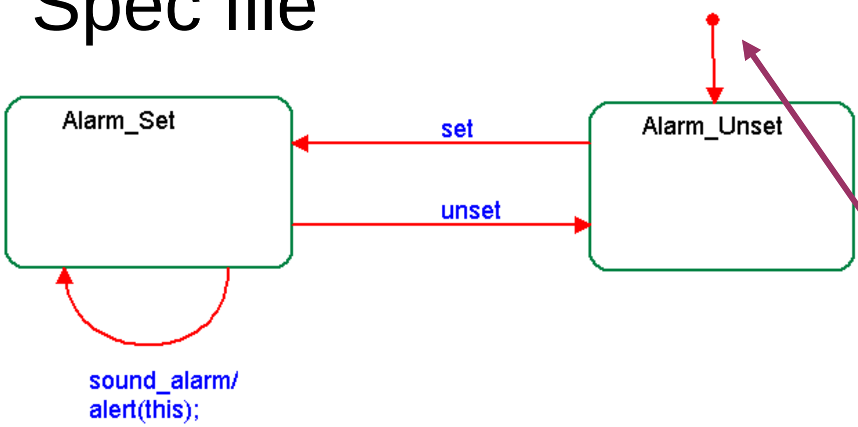
What if the Alarm is not set?

- Where do we store this?
  - An Attribute of the Clock
  - An Attribute of the Alarm
  - The Alarm State



Statechart Code - Defs

- Each State & Event has an ID defined in the Spec file



Default State

```
RiA_Non_State : constant integer := 0;

Alarm_Set : constant integer := 1;

Alarm_Unset : constant integer := 2;
```

States

Events

```
set_id : constant Integer := 1;

sound_alarm_id : constant Integer := 2;

unset_id : constant Integer := 3;
```

Statechart Code - Events

- Events are record Types with accessors
  - Events have Data
  - Event Data is also record type with accessor

Event

```
-- event record type
type Event is
record
  id : Integer := 0;
  data : Event_Data_acc := null;
end record;

type Event_acc is access Event;
```

Event Data

```
type Event_Data (event_id : integer) is
record
  case event_id is
    when set_id =>
      null;
    when sound_alarm_id =>
      null;
    when unset_id =>
      null;
    when others =>
      null;
  end case;
end record;

type Event_Data_acc is access Event_Data;
```

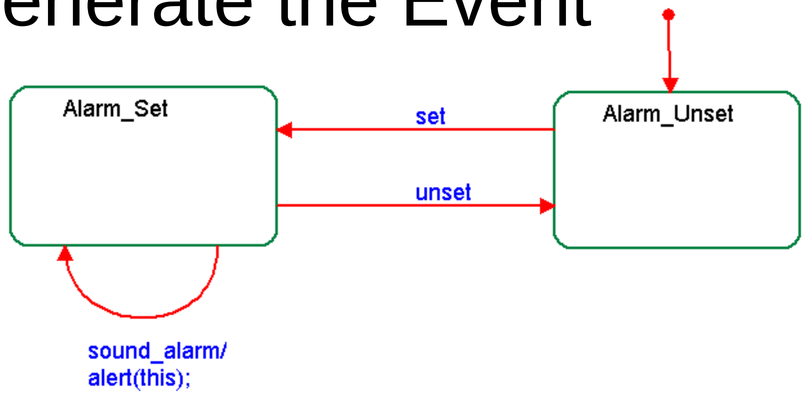
Class Code - Data

- The Class holds extra attributes for the statechart
  - Including a pointer to the current event being processed

```
type Alarm_t is tagged

record
  ------------------------------------------------------------
  --      Generated Statechart Fields for Reactive Class     --
  ------------------------------------------------------------
  ria_behavior_started : Boolean := false;
  ria_behavior_terminated : Boolean := false;
  ria_null_transitions_count : Integer := 0;
  ria_state_machine_busy : Boolean := false;
  ria_current_event : Event;
  ria_events_available_signal : RiA_Event_Flag_Ada83.RiA_Event_Flag_Ada83_acc_t := null;
  ria_queue_guard : RiA_Mutex_Ada83.RiA_Mutex_Ada83_acc_t := null;
  root_state_active : Integer := RiA_Non_State;
  root_state_sub_state : Integer := RiA_Non_State;
  ------------------------------------------------------------
  ------------------------------------------------------------

end record;
```

Statechart Code – Generating Events

- Each Event has a corresponding operation allowing a client to generate the Event




```
--++ operation set()
procedure set (this : in out Alarm_t);
```

```
--++ operation unset()
procedure unset (this : in out Alarm_t);
```

```
--++ operation sound_alarm()
procedure sound_alarm (this : in out Alarm_t);
```


I-Logix

Statechart Code for Events

- For Each Event:
  - The class has an operation that the client can call to generate the event into the statechart
    - Synchronous: Operation is name of event
    - aSynchronous: Operation is named take_eventname
      - Or gen_eventname
  - The Class has a ‘state_take_eventname’ for each state
    - Allows each state to handle an event independently

I-Logix

Event Operation Code

```
procedure set (this : in out Alarm_t) is
   ev : Event;
   ria_consume_result : RiA_Types.Consume_event_status;
begin
   initialize_event(ev, set_id, null);
   take_event(this, ev, ria_consume_result);
end set;
```

Creates the Event

Calls the Root of the Statechart with the new Event

Event Initialisation Code

```
procedure initialize_event (ev : in out Event; event_id : in Integer;
 data : in Event_Data_acc := null) is
begin
  ev.id := event_id;
  ev.data := data;
end initialize_event;
```


I-Logix

Statechart Structure

- The Statechart is one large switch statement, broken up into smaller switch statements
- Take_Event checks overall statechart behaviour (eg is it started yet?)
  - Take_Event calls Dispatch Event to pass the event onto the current state
  - Each state has its own take_event operation as a root
    - This dispatches any event received to :
    - Each state then has its own take_eventname to actually handle the event

Take_Event

Check If Statechart is still active

Set current Event

Dispatch Event

```
procedure take_event (this : in out Alarm_t;
 ev : in Event;
 result : out RiA_Types.Consume_event_status) is
  event_consume_status : RiA_Types.Consume_event_status
    := RiA_Types.event_not_consumed;
begin
  if not is_non_event(ev) then
      if is_behavior_terminated (this) then
            event_consume_status := RiA_Types.reached_terminate;
      elsif is_behavior_started (this) then
          -- consume the event
          if not is_state_machine_busy (this) then
              -- get busy : not thread safe
              do_state_machine_busy (this);
              -- set the current event
              this.ria_current_event := ev;
              -- consume the event
              dispatch_event(this, ev.id, event_consume_status);
              -- complete consumption of null transitions
              if should_complete_event_consumption (this) then
                  consume_null_transitions (this);
              end if;
              -- reset the current event
              set_non_event(this.ria_current_event);
              -- ready for the next event
              undo_state_machine_busy (this);
          end if;
      end if;
      -- let the user handle unconsumed events
      if not is_behavior_terminated (this) and
         not RiA_Types.is_event_consumed(event_consume_status)
      then
         handle_unconsumed_event(this, ev);
      end if;
  end if;
  result := event_consume_status;
end take_event;
```


I-Logix

Dispatch_Event

- Dispatch Event identifies the current State and calls the ‘root’ take event operation of that state

```
procedure dispatch_event(this : in out Alarm_t;
  event_id : in Integer;
  result : out RiA_Types.Consume_event_status) is
begin
  case this.root_state_active is
  when Alarm_Set =>
    alarm_set_take_event(this, event_id, result);
  when Alarm_Unset =>
    alarm_unset_take_event(this, event_id, result);
  when others =>
    result := RiA_Types.event_not_consumed;
  end case;
end dispatch_event;
```

The Alarm Set State

- The State_Take_Event operation identifies the event and calls the appropriate take_eventname for that state
  – Analagous to individual ‘dispatch_event’ operations for each state

```
procedure alarm_set_take_event (this : in out Alarm_t;
  event_id : in Integer; result : out RiA_Types.Consume_event_status) is
  temp_result : RiA_Types.Consume_event_status := RiA_Types.event_not_consumed;
begin
  case event_id is
  when sound_alarm_id =>
      alarm_set_take_sound_alarm(this, temp_result);
  when unset_id =>
      alarm_set_take_unset(this, temp_result);
  when others =>
      null;
  end case;
  result := temp_result;
end alarm_set_take_event;
```

Alarm_Set_Take_Sound_Alarm

Leaving the State

Action On Transition

Entering the next state

```
procedure alarm_set_take_sound_alarm (this : in out Alarm_t;
 result : out RiA_Types.Consume_event_status) is
begin
  result := RiA_Types.event_not_consumed;
  alarm_set_exit (this);
  alert(this);
  alarm_set_enter (this);
  result := RiA_Types.event_consumed;
end alarm_set_take_sound_alarm;
```

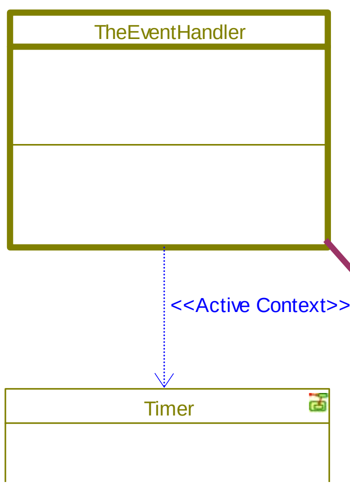
aSynchronous Statecharts

- Require an Event Handler – ‘Active Context’

To Be Continued ….

Creating an Event Driven Statechart

- Create the Statechart as usual.
- Add an EventHandler Class & make its Class Visibility Private
  - Make it Active
  - Add a Dependency from it to any Reactive Class and stereotype that to <<Active Context>>



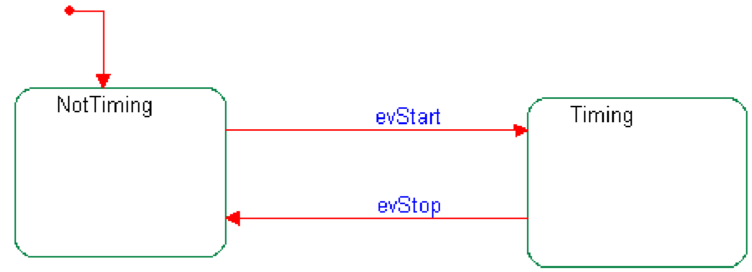
The combination of these two things causes extra operations to be generated in the EventHandler

```
procedure register_context_Timer (this : in out TheEventHandler_t;  reactive : in out Timer.Timer_t);
```


I-Logix

Statechart Code

- An Example Statechart



Events are defined at Package Level


```
evstop_id : constant Integer := 1;

evstart_id : constant Integer := 2;
```

States are defined in the Statechart Code for the Class

```
RiA_Non_State : constant integer := 0;

NotTiming : constant integer := 1;

Timing : constant integer := 2;
```

Statechart Event Dispatching Code

- Important Operations Generated for a Statechart:

procedure start_behavior	Kicks off the Statechart
Procedure take_*eventname* One Procedure is generated per Event EG: procedure take_evStop	Creates an Event and passes it to it's Active Context to be placed on a queue.
procedure take_event (Event)	Actually Consumes the Event and causes a State Transition. Called by the Active Context
procedure Initialize procedure Finalize	Constructor/Destructor

```
procedure take_evstop (this : in out Timer_t) is
  ev : Event_acc;
begin
  check_connected_to_active_context (this);
  ev := create_event(Default.evstop_id, null);
  Event_Queue.put(this.ria_event_queue, ev);
  signal_event_ready (this);
end take_evstop;
```


I-Logix

Instantiating a Reactive Object

- Create the Event Handler and Initialise it

```
-- create active object
    active_object := new TheEventHandler.TheEventHandler_t;
    TheEventHandler.Initialize(active_object.all);
```

- Create the Reactive Object and Initialise it

```
-- create reactive object
TheTimer := new Timer.Timer_t;
Timer.Initialize(TheTimer.all);
```

- Register the Reactive object with the Event Handler

```
-- register the reactive object on its active
TheEventHandler.register_context_Timer(active_object.all, theTimer.all);
```

- Start the Event Handler Task
  - Causes it to spawn a thread and process its object queue repeatedly

```
-- start the active instance task
TheEventHandler.start(active_object.all);
```

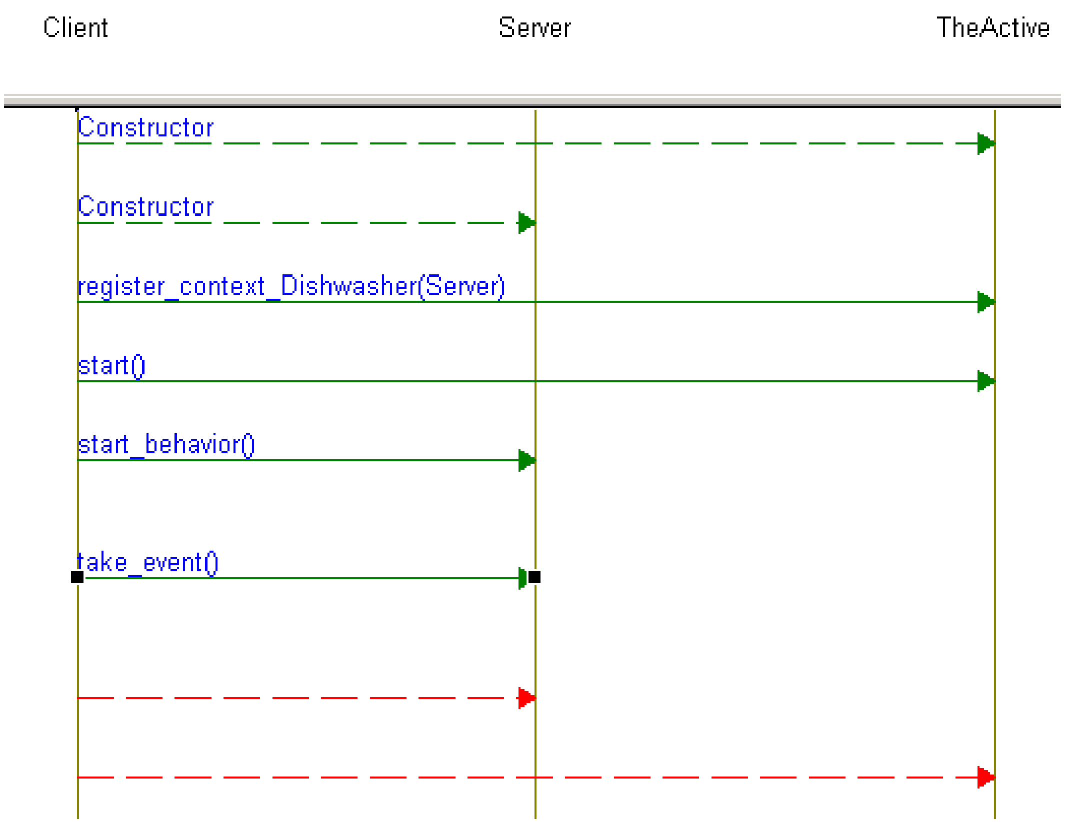
- Start the Reactive Object's Statemachine

```
-- start the reactive object behavior
Timer.start_behavior(TheTimer.all,timer_status);
```

Framework Use Sequence

- A sequence of the prior collaboration is shown here.

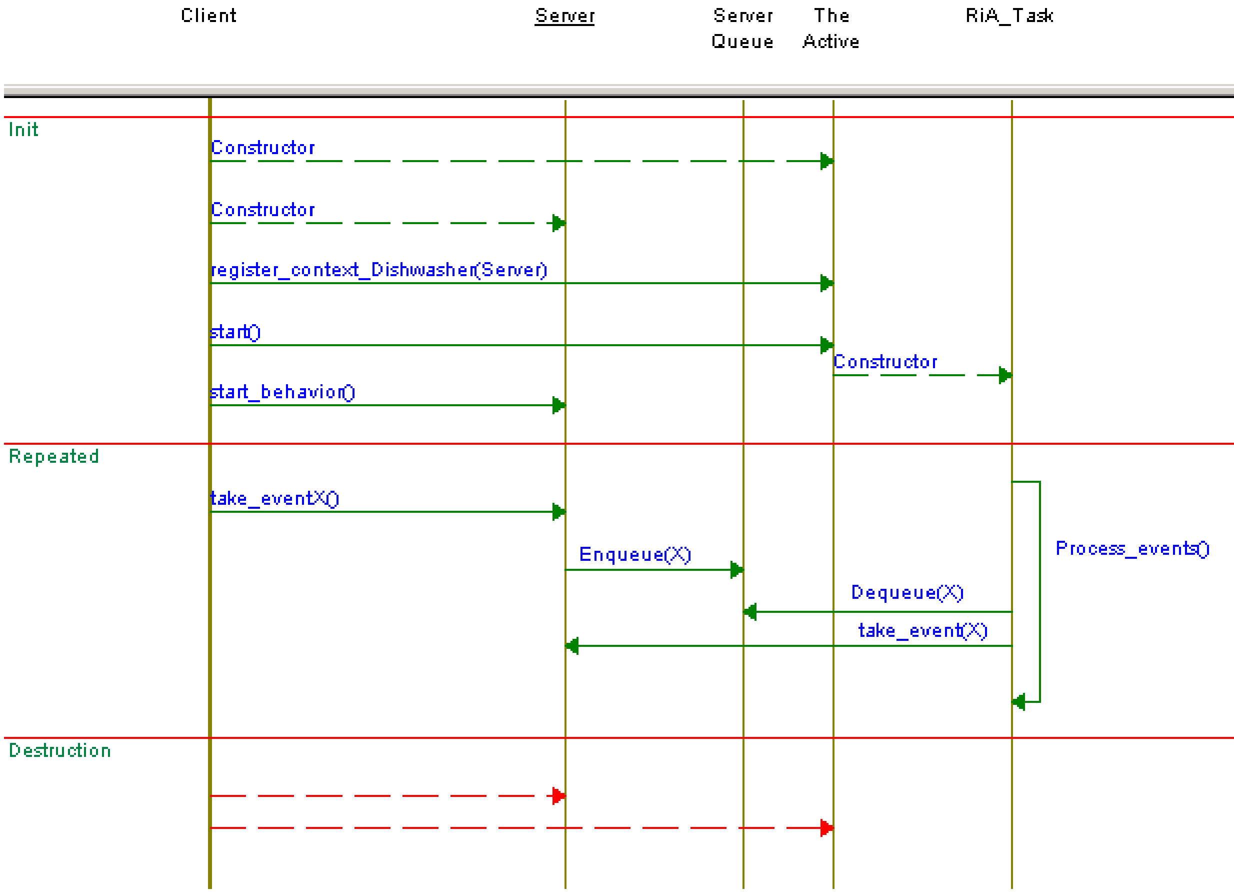
- The *client* does the creation, starting, message sending and deletion in an orderly fashion.



I-Logix

Framework Internal Behavior

- If we look at the *start* operation for an active object more closely we see there is much going on.
- A task is spawned and the reactive object queue is processed repeatedly to cause event consumption in the reactive state chart.



I-Logix