BridgePoint Modeling – Exercises in Building Executable Models

Mentor Graphics Corporation

Published April 2013

1. Lab 1

1.1 Background

BehindTheTimes Inc. is planning a new range of sports watches incorporating GPS technology. The engineering group is researching availability of GPS technology - either integrated IP, or as a separate receiver. Past products have been late to market because of software delays; software testing has always had to wait for availability of prototype hardware. The product manager is eager to see the software designs prototyped and tested - which is a claim for the newly adopted model-driven development strategy.

1.2 What we want to accomplish

The purpose of this exercise is to:

- Become familiar with the graphical user interface
- Create and edit basic model elements
- Learn how to launch Verifier and how to feed stimuli into the model

1.3 Task

This instruction takes you through the construction of two UML components, one called 'Location' that will simulate the physical GPS and one called 'Tracking' which will encapsulate the core application.

Set up the project

1. Make sure the xtUML Modeling perspective is active:

- Window > Open perspective > xtUML Modeling
- Enable 'Link with Editor'

2. Create a project:

- Right click the Model Explorer view and select New > xtUML Project
- Name it 'Gps Watch'
- Leave default values and press Finish

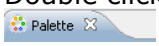
Structure the model

3. Create two models:

- Right click the project in Model Explorer and select New > Package
- Enter 'Tracking' in the name field
- Repeat the step in here to create another model called 'Location'

	We create these models to allow us to formalize them into components which can be interconnected via interfaces.

4. Add subsystems:

- Double click the Tracking model created in the step above to open the Domain Package Diagram
- Double click the Tracking Package from the Default Toolset under the 

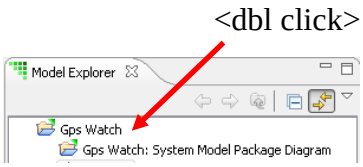
- Draw the package somewhere on the canvas and name it 'Tracking'
- Repeat the steps in here to create a similar subsystem in the Location component, name it 'Location'

5. Create two components:

- Double click the Tracking model created in the previous step to open the Package Diagram
- Activate the Component tool  in the 

and add two components to the diagram
- Name them 'Tracking' and 'Location'

6. Create a package diagram to hold System:



- Open the System Model Package Diagram by double clicking the project in Model Explorer
- Select the  Package from the Default Toolset, draw a new package on the diagram and name it 'System'
- Double click System to open our new component diagram

6. Create two component references:

- Activate the Component Reference tool  from the 

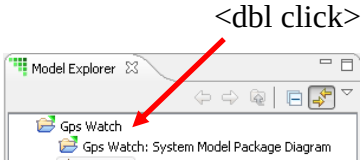
and add two component references to the diagram

7. Formalize the component references:

- Right click each component and select Assign Component...
- The models created in step 5 appears in the drop down list, select corresponding model and press Finish
- The reference names should now change to match the path of the assigned component in Model Explorer

Defining data types

8. Create a data type package:



- Open the System Model Package Diagram again (double click the project in Model Explorer)
- Add a new  Package to the canvas
- Name it 'LocationDataTypes'

9. Create a new structured data type

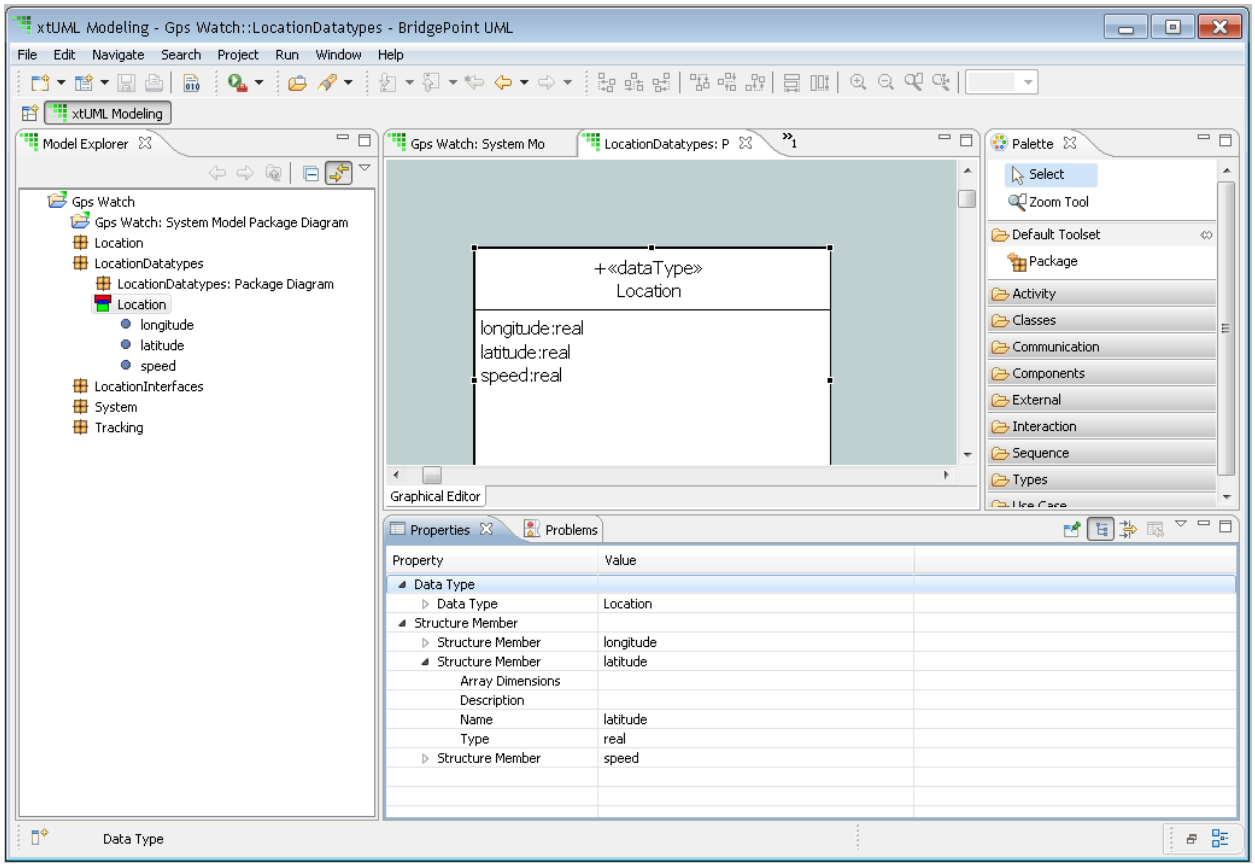
- Double click the package created in the previous step
- Select the New Structured Data Type tool from the Palette and draw the new data type on the canvas
- Name it 'Location'

10. Add members to the structured data type

- Right click the data type and select New > Member
- Select the data type on the canvas (left click)
- In the Properties view, expand the field called Structure Member
- Highlight the value of Type and change the data type to 'real'

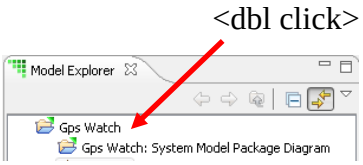
11. Repeat step 10 to create two additional members:

- 'latitude' of type real
- 'speed' of type real



Defining interfaces

12. Create an interface package:



- Open the System Model Package Diagram again (double click the project in Model Explorer)
- Right click the canvas and select New >  Package
- Name the package 'LocationInterfaces'

13. Define an interface:

- Double click the package created in the previous step
- Select the New Interface tool  from the 

and draw the new interface
- Name it 'LocationProvider'

14. Add a signal:

- Right click the interface and select New > Signal
- Select the interface on the canvas (left click)
- In the Properties view, expand the field called Interface Signal
- Change the message direction to 'From Provider'
- Name it 'locationUpdate'

15. Add a parameter to the 'locationUpdate' signal:

- In Model Explorer, expand: GpsWatch::LocationInterfaces::LocationProvider
- Right click the locationUpdate signal and select New > Parameter (This can also be done from the Outline view)
- Right click the parameter and Set type...
- Select the Location data type and click OK
- Name the parameter 'location'

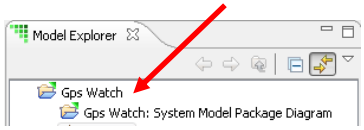
16. Repeat step 14 to create two additional signals:

- One named 'registerListener' with direction 'To Provider'
- And another named 'unregisterListener' with direction 'To Provider'
- None of these signals should carry data, i.e. no need to add parameters

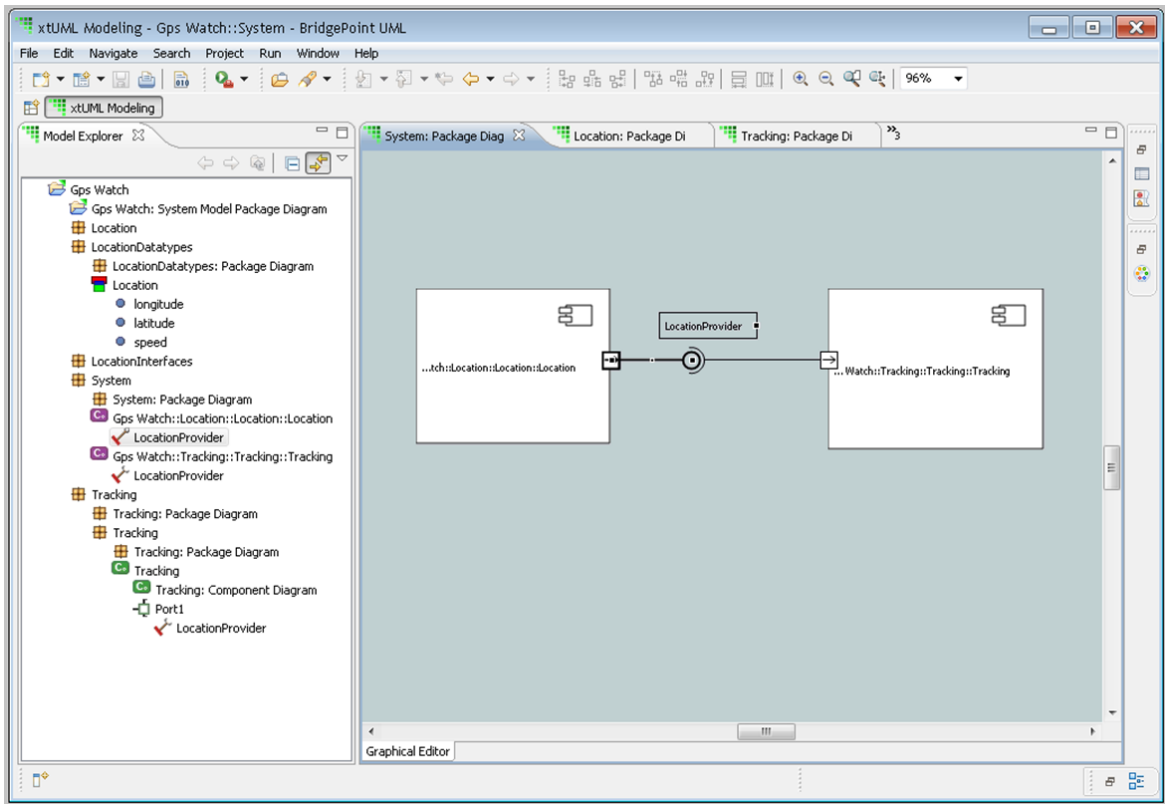
Build the system model

17. Connect the components:

<dbl click>



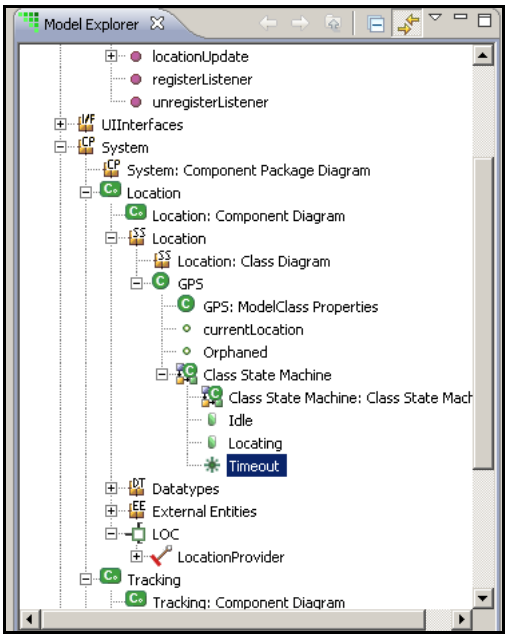
- Open the System Model Package Diagram again (double click the project in Model Explorer)
- Double click on the Location package
- Draw the provider end of the interface and attach it to the Location component, use the New Provided Interface Tool from the Palette
- Right click the interface and select Formalize...
- Locate the interface package and the interface defined in 12-13 and OK
- Return to the Model Explorer and Double click on the Tracking package
- Draw the requiring end of the interface and attach it to the Tracking component, use the New Required Interface Tool from the Palette
- Right click the interface and select Formalize...
- Locate the interface package and the interface defined in 12-13 and OK
- In Model Explorer, navigate to the ports of the recently created interface:
  - GpsWatch::Location::Location::Port1
  - GpsWatch::Tracking::Tracking::Port1
- Rename both of them to 'LOC'
- Return to the Model Explorer and Double click on the System package
- Connect the provided end to the required end (or vice versa)



Defining behavior

18. Provide a behavioral model for the Location component:

- Double click the Location component
- Double click the Location subsystem to open the class diagram
- Select the New Class tool and draw a new class
- Select it and set Name and Key Letters to 'GPS' over in the Properties view
- Right click the class to create a new attribute
- Use the Properties view to name it 'currentLocation' set type to Location
- Add another attribute called 'timer' of type inst_ref<Timer>

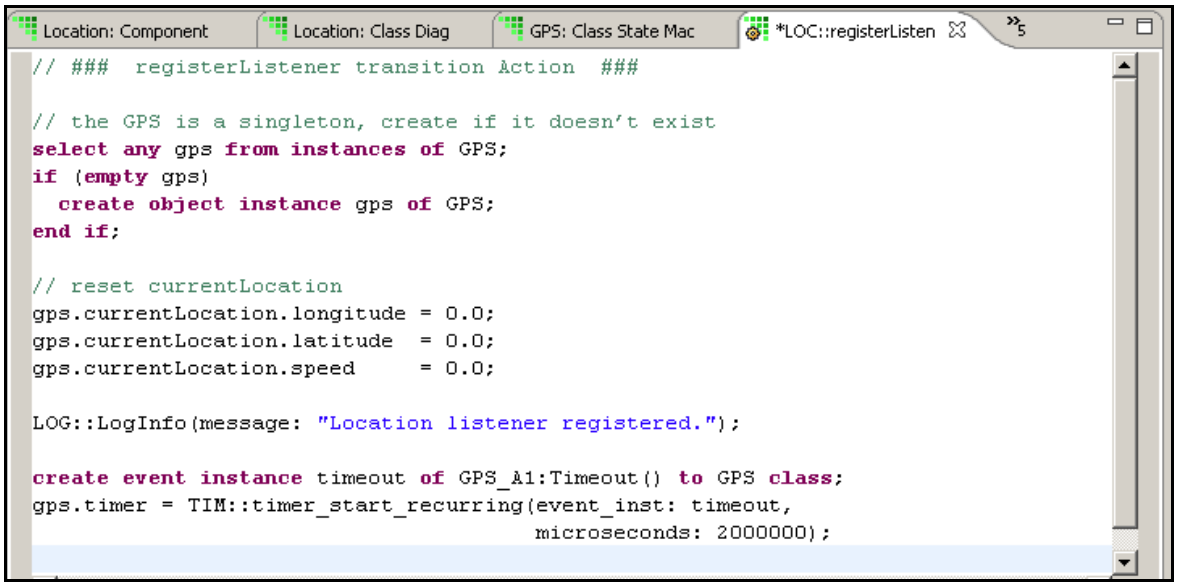


19. Define the life cycle of the GPS:

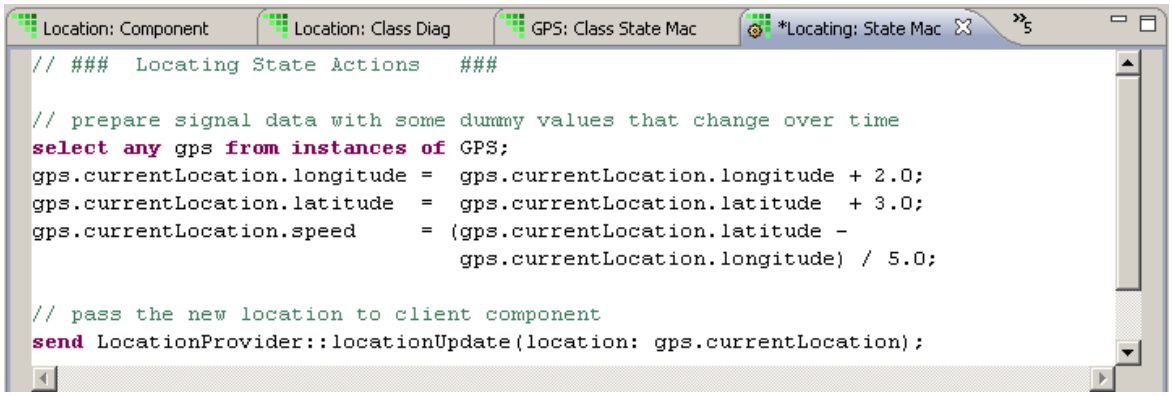
- Right click the GPS class and to create a new Class State Machine
- Locate the state machine in Model Explorer, right click and add new Event, rename the event to 'timeout'
- Double click the state machine to open its editor
- Select New State tool and draw two new states
- Rename state 1 to 'idle' and state 2 to 'locating'
- Select the New Transition tool and draw the following transitions:
  - From state idle to state locating
  - From state locating to state idle
  - From state locating back to itself
- Formalize transitions:
  - Right click transition idle -> location and Assign Signal registerListener
  - Right click transition location -> idle and Assign Signal unregisterListener
  - Right click transition location -> location and Assign Event timeout
- Open the State Event Matrix editor (a tab at the bottom of the state machine editor)
- In row 'idle', column 'GPS_A1:timeout' select 'Event Ignored' from the drop down list

20. Enter state machine actions:

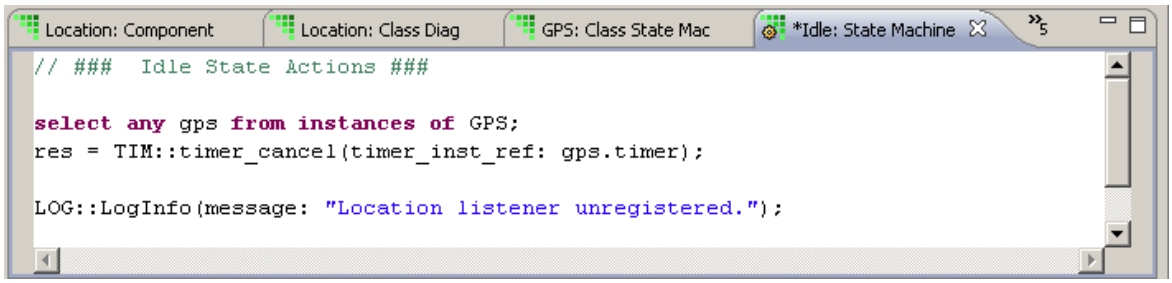
- Double click the registerListener transition to open the transition action editor
- From Appendix B, copy OAL_1 into the editor and save



- Double click the locating state and enter OAL_2

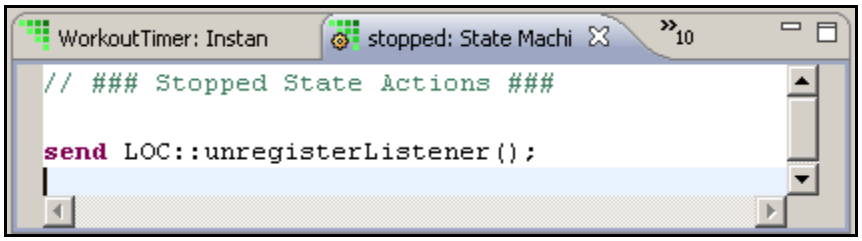


- Double click the idle state and enter OAL_3

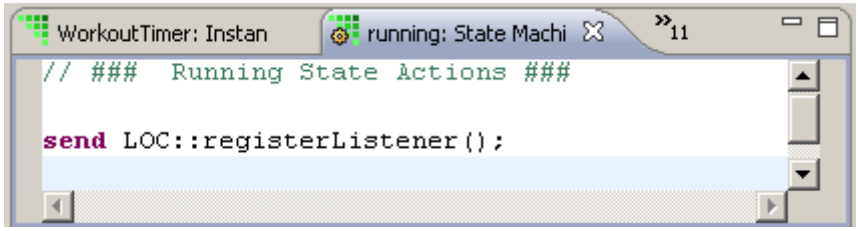


21. Take the first step to model the Tracking component:

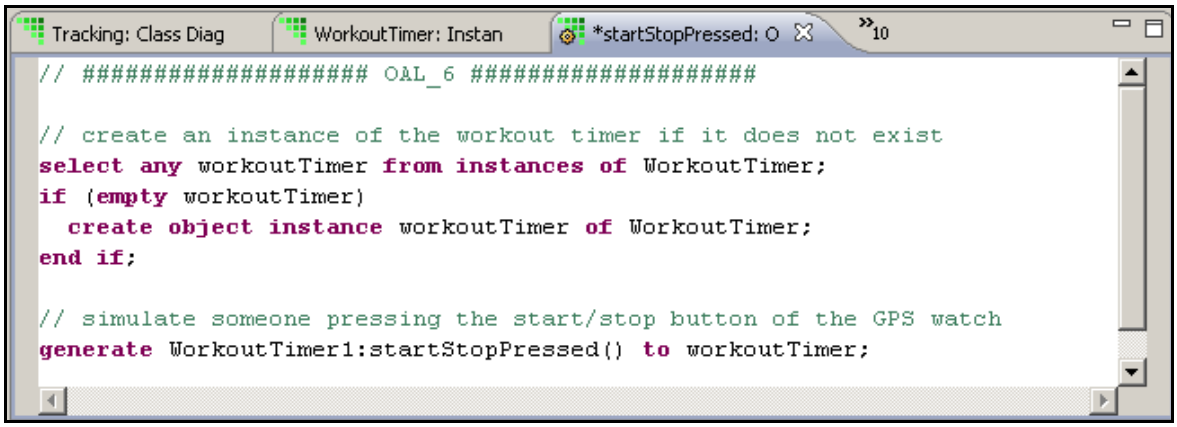
- In the Tracking subsystem, add a class called 'WorkoutTimer'
- Set Key Letters to 'WorkoutTimer' as well
- Add an Instance State Machine
- Add two states, 'stopped' and 'running'
- Draw two transition, one stopped > running and one running > stopped
- Create an event called startStopPressed and assign both transitions created in the previous step to it
- In the stopped state enter OAL_4



- In the running state enter OAL_5

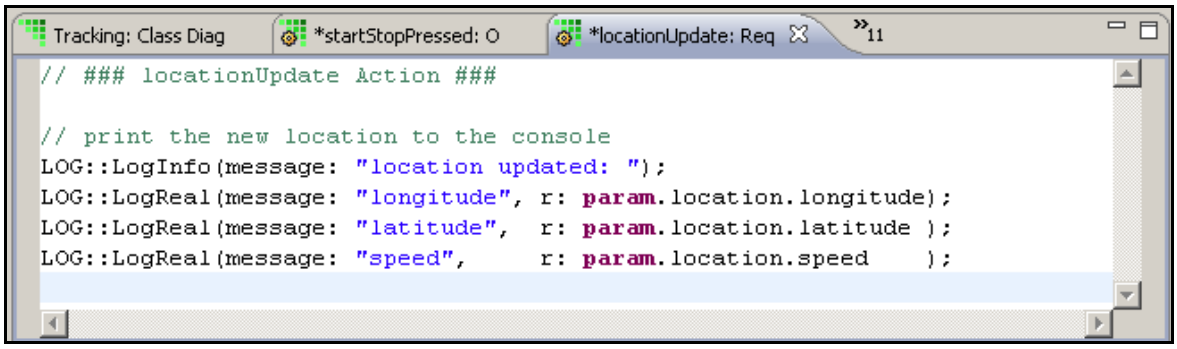


- Add an operation to this class called 'startStopPressed'
- Select the class to get access to the properties of the class
- In the Properties view, locate the operation and change its Instance Based Indicator to Class Based
- Still in the Properties view, open the Action Semantics Field and enter OAL_6 (This can also be done by double clicking the operation in Model Explorer)



22. Terminate the locationUpdate signal:

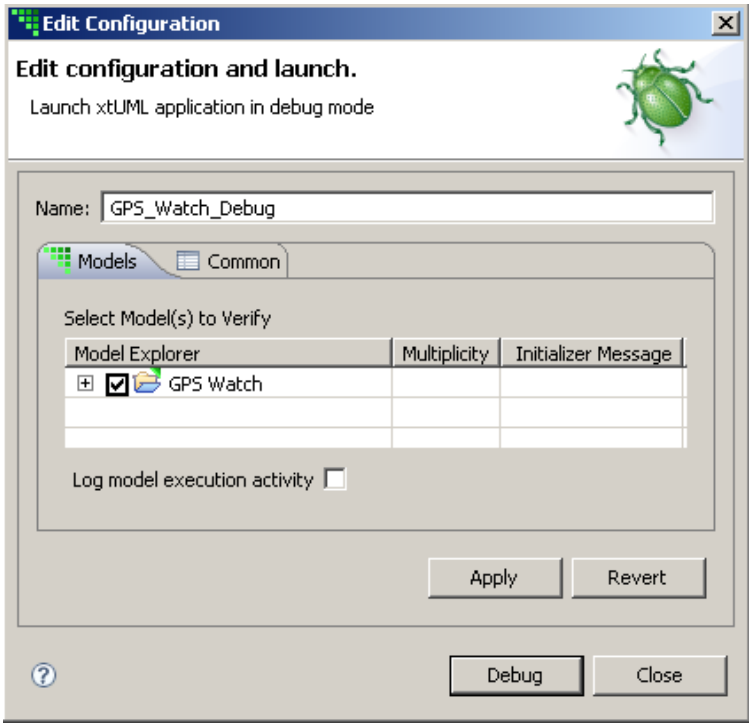
- In Model Explorer, navigate to: GpsWatch::System::Tracking::LOC::LocationProvider::locationUpdate
- Double click the signal and enter OAL_7



Verify the model

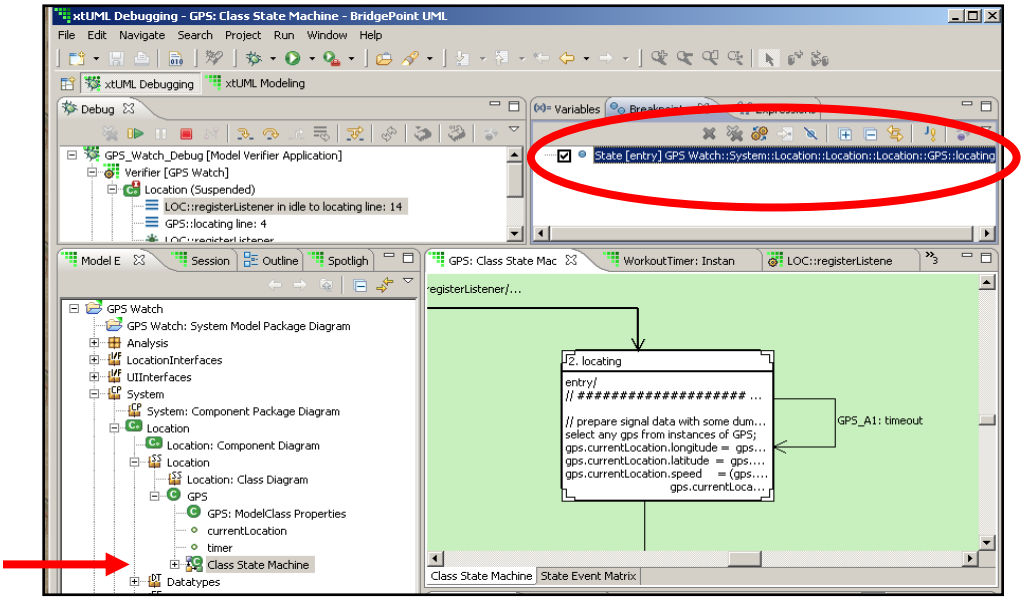
23. Test your model in Verifier:

- Switch to the xtUML Debugging perspective (Window > Open Perspective > xtUML Debugging)
- From the menu, go to Run > Debug Configurations..
- Right click Model Verifier Application to create a new executable
- Name it 'GpsWatch'
- Select your project to include all components in this executable
- Click Debug



- Launching the debugger should automatically switch to the xtUML Debugging perspective. If not, switch to that perspective (**Window > Open Perspective > xtUML Debugging**).

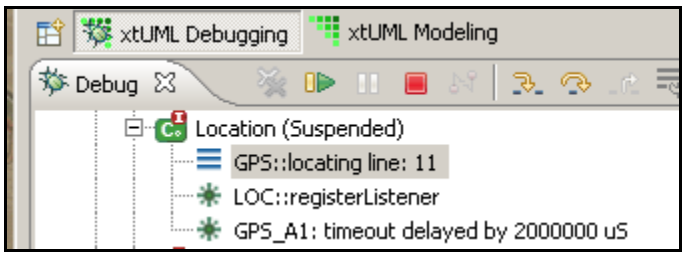
- Set a breakpoint. In Model Explorer, open the GPS Class State Machine. Navigate to *GPS Watch::System::Location::Location::GPS*. Right-click on the *Locating* state and select **Set Breakpoint**. You can see the breakpoint added in the Breakpoint window. You can also edit breakpoint properties, or disable breakpoints from this window. By default we create a breakpoint on entering the state.



- In the Debug Perspective, we can see the Session Explorer which offers a dynamic view of the running system, with created instances and structure. We use this view, rather than Model Explorer, when interacting with a running program. In Session Explorer, expand **GPS Watch > Tracking > Tracking > WorkoutTimer**. Right click startStopPressed and select **Execute**.

- Note that the verifier runs to our breakpoint, and displays our current location in the OAL code. Bring up the variables window by clicking on the tab next to the Breakpoint window. Expand the structures of the variables to see current values.

- The controls for the debugger are in the debug window. Controls exist for **Run**, **Pause**, **Stop**, **Step into**, **Step over**, etc. Step through the code and watch the variables change.



- You can toggle breakpoints from the OAL editor by double-clicking in the left margin. Set some breakpoints and use the **Run** and **Step** commands to become familiar with debugger operations.

- Find more information about model execution in the Help section:
  Help > Help Contents Then navigate to:

  BridgePoint UML Suite Help > Getting Started > Model Verification

2. Lab 2

2.1 Background

One of the main features of the GPS watch is the collection of coordinates that will track the user during his/her exercise. Before data can be collected there has to be a data model representing such track. There is a requirement to maintain the order in which coordinates are created to allow the track to be plotted on a map.

2.2 What we want to accomplish

The purpose of this exercise is to:

- Learn how object-oriented analysis can precisely capture the data
- Understand classes, attributes, associations with conditionality and multiplicity

2.3 Task

- In the Tracking component, build a class diagram representing a track (use the existing subsystem in this component)
- Identify the classes that will represent the track
- Identify and add attributes that belong to the classes you decide to build
- Establish necessary associations between classes
- Carefully choose multiplicity and conditionality of your associations

3. Lab 3

3.1 Background

Currently we have only captured the data modeling but data is not processed and stored as it arrives.

3.2 What we want to accomplish

The purpose of this exercise is to learn:

- Learn the object action language, OAL
- Create operations
- Work with breakpoints, stepping and variables in Verifier

3.3 Task

- Each time new location data arrives, make sure the class model representing the track is instantiated accordingly
- Verify that the model behaves as it should using Verifier

4. Lab 4

4.1 Background

After some industrial espionage at a local sports store there is a decision to add lap counting: hitting a button marks the end time of a lap - but the timer keeps running. With some persuasion, engineering agreed to add the button for this.

At our requirements review, someone pointed out that this was a one-shot watch; there is no way to reset the timer value, or to discard an old track and start fresh. Once refreshed, a succeeding press of the start/stop button will begin gathering data for a new track. Reset should only occur if the workout timer is not running. For this reason it was concluded that the lap button should also be used to reset the watch, making it the lap/reset button.

4.2 What we want to accomplish

After this exercise you will know how to work with:

- Lifecycles
- States
- Transitions
- Events
- Spotlight

4.3 Task

- Update the state machine in the WorkoutTimer class to cater for signals generated by the lap/reset button
- Extend your class model to include the notion of a lap

5. Lab 5

5.1 Background

A sports watch is not worth its name if it doesn’t keep track of time.

5.2 What we want to accomplish

The purpose of this exercise is to learn:

- Start and cancel timers

5.3 Task

- When recording, make sure time is measured, use a 1 sec. timer for this purpose.

6. Exercise 6

6.1 Background

An optional heart rate monitor will provide updated beats-per-minute values which we would like to add to the track data. These updates are transmitted only when the measured heart rate changes - i.e. not at any predictable intervals.

6.2 What we want to accomplish

The purpose of this exercise is to:

- Build component and interface packages
- Understand how interfaces can be defined independently of components
- Required and Provided interfaces; how components are 'wired' together

6.3 Task

- Add a new component that will act as hart rate monitor. Connect the new component to the Tracking component over an interface containing register/unregister signals as well as a signal for the actual heart rate update. The heart rate signal should carry an integer with a beats-per-minute value.
- Update the tracking component to register and unregister for this service and take appropriate action when heart rate data is delivered.

7. Exercise 7

7.1 Background

The display of the watch is not big enough to show all metrics that the runner is typically interested in. For this reason the engineering team has decided to add yet another button called “mode”. Pressing the mode button will switch between the available modes:

- Distance
- Speed
- Pace
- Heart rate
- Lap count

7.2 Background

The purpose of this exercise is to:

- Understand different means of signal termination
- Benefit from derived attributes
- Work with cardinalities
- Understand class based state machines

7.3 Task

Build a class based state model that models the current mode. The current mode will determine the type of value being passed to the UI. Calculate these values when necessary and pass the correct one to the UI using the setData signal. Consider the option of making pace a derived value of speed.

8. Exercise 8

8.1 Background

BehindTheTimes have consulted with a fitness expert who believes in regular exercise regimes. He suggests defining workout 'plans' which can be run on a regular basis. There is no requirement, as yet, to interactively edit these workout plan specifications; they are created on a PC (we do not have to be concerned about how) and can be downloaded to the device. A user interaction does allow a downloaded workout to be designated as the selected plan; in this case, pressing the 'set target' button causes the watch not only to record a track, but to notify the runner as the activity steps are completed.

A workout plan can have any number of exercise steps. Each step may specify either duration or distance; the runner will be notified as the step completes. If neither duration nor distance is specified, the step ends when the lap button is pressed. Each step may additionally specify either a goal running pace or a goal heart rate; if specified, the runner will be alerted if he deviates significantly from the goal. (Let us assume some fixed percentage above or below the goal)

The sequence of steps in a workout may contain repetitions - i.e. after completing any given step, the workout may repeat from any designated previous step, up to some specified number of repetitions. Such repeated sequences can be nested.

8.2 What we want to accomplish

The purpose of this exercise is to:

- Learn to think about capturing specifications in passive classes
- Discover a use for generalization/specialization
- Partitioning into subsystems

8.3 Task

Create a new subsytem to hold workout specifications that meet the stated requirements. Consider how such a specification might be used to notify or alert a runner during an actual workout session.

Appendix A – OAL

```
// ##################### OAL_1 ####################

// the GPS is a singleton, create if it doesn't exist
select any gps from instances of GPS;
if (empty gps)
  create object instance gps of GPS;
end if;

// reset currentLocation
gps.currentLocation.longitude = 0.0;
gps.currentLocation.latitude  = 0.0;
gps.currentLocation.speed     = 0.0;

LOG::LogInfo(message: "Location listener registered.");

create event instance timeout of GPS_A1:timeout() to GPS class;
gps.timer = TIM::timer_start_recurring(event_inst: timeout,
                                       microseconds: 2000000);


// ##################### OAL_2 ####################

// prepare signal data with some dummy values that change over time
select any gps from instances of GPS;
gps.currentLocation.longitude =  gps.currentLocation.longitude + 2.0;
gps.currentLocation.latitude  =  gps.currentLocation.latitude  + 3.0;
gps.currentLocation.speed     = (gps.currentLocation.latitude -
                                 gps.currentLocation.longitude) / 5.0;

// pass the new location to client component
send LocationProvider::locationUpdate(location: gps.currentLocation);



// ##################### OAL_3 ####################

select any gps from instances of GPS;
res = TIM::timer_cancel(timer_inst_ref: gps.timer);

LOG::LogInfo(message: "Location listener unregistered.");


// ##################### OAL_4 ####################

send LOC::registerListener();



// ##################### OAL_5 ####################

send LOC::unregisterListener();
```

```
// ##################### OAL_6 ####################

// create an instance of the workout timer if it does not exist
select any workoutTimer from instances of WorkoutTimer;
if (empty workoutTimer)
  create object instance workoutTimer of WorkoutTimer;
end if;

// simulate someone pressing the start/stop button of the GPS watch
generate WorkoutTimer1:startStopPressed() to workoutTimer;



// ##################### OAL_7 ####################

// print the new location to the console
LOG::LogInfo(message: "location updated: ");
LOG::LogReal(message: "longitude", r: param.location.longitude);
LOG::LogReal(message: "latitude",  r: param.location.latitude );
LOG::LogReal(message: "speed",     r: param.location.speed    );
```