iUML Tutorial

iUML & iUMLite 2.1 NT

Manual Revision 2
About this Tutorial

An introduction to the iUML Tutorial

This tutorial provides you with an opportunity to explore the executable UML (xUML) formalism and how it is supported by iUML. In the Tutorial you will develop a model based on a case study.

This tutorial has been designed specifically for those conducting an iUML evaluation but will be equally useful to new users of iUML. It will give an overview of the features of iUML using a real xUML model.

Developing with xUML

xUML is a subset of the Unified Modelling Language incorporating a complete Action Language that allows system developers to build executable system models and then use these models to produce high quality code for their target systems.

It is based upon the principle of building a set of precise, testable analysis models of the system to be developed, executing defined tests on these models, and defining a systematic strategy by which the models will be used to produce code for the desired target system.

To find out more about xUML read the xUML Tutorial.

The Petrol Station Control System case study

The Petrol Station Control System is a case study, which will guide you through the entire system development process, enabling you to be involved in a ‘real’ xUML project from analysis to simulation.

Note that the following requirements deliberately embody the usual mix of untestable statements, ambiguities and contradictions that are found in most requirement specifications!
Mission Statement

This Petrol Station Control system manages the dispensing of fuel, customer payments and tank levels.

A computer-based system is required to control the dispensing of fuel, to handle customer payment and to monitor tank levels. The system must be “best of breed”, easy to use, reliable, fast and easy to modify to incorporate requirements yet to be conceived.

- **Enabling the Pump & Delivering Petrol**
  Before a customer can use the self-service pumps, the pump must be enabled by the attendant. When a pump is enabled, the pump motor is started, if it is not already on, with the pump clutch free. When the trigger in the gun is depressed, closing a micro switch, the clutch is engaged and fuel pumped. When it is released, the clutch is freed. There is also a micro switch on the holster in which the gun is kept which prevents fuel being pumped until the gun is taken out. Once the gun is replaced in the holster, the delivery is deemed to be completed and the pump disabled. Further depressions of the trigger in the gun cannot dispense more fuel. After a short stand-by period, the pump motor will be turned off unless the pump is re-enabled.

- **Measuring the Delivery**
  A metering device in the fuel line sends a pulse to the system for each hundreath of a litre dispensed. The cost of the fuel is calculated using the amount delivered and unit cost which is displayed on the pump.

- **Payment**
  Transactions are stored until the customer pays. Customers sometimes abscond without paying and the operator must annotate the transaction with any available information, the vehicle’s registration number for example. At the end of the day, transactions are archived and may be used for ad-hoc enquiries on sales.

- **Fuel**
  At present, two grades of fuel are dispensed from three pumps on the forecourt. Each pump takes its supply from one of two tanks, one tank for each grade. The tank level must not drop below 4% of the tank’s capacity. If this happens, the pumps serviced by that tank cannot be enabled to dispense fuel.

On a real project the requirements would be significantly more complex but for the purposes of this tutorial they have been deliberately kept concise in order to bring out the key aspects of both the method and the tool.
Conventions used in the Tutorial

These are the conventions used in the tutorial to describe your interaction with the user interface:

<table>
<thead>
<tr>
<th>Styles Used in the Tutorial</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>User Entry</strong></td>
</tr>
<tr>
<td><em>Bold Courier</em> is used for anything you are required to type in.</td>
</tr>
<tr>
<td><strong>Tool Text</strong></td>
</tr>
<tr>
<td>Text and Menu items in iUML are shown in Courier. Courier is also used for information contained in tables which you are required to enter.</td>
</tr>
<tr>
<td><strong>Method Explanations</strong></td>
</tr>
<tr>
<td>Explanations of the xUML method are given throughout the tutorial. They can be found at the bottom of the relevant pages.</td>
</tr>
<tr>
<td><strong>Tool Items</strong></td>
</tr>
<tr>
<td><em>Helvetica Narrow</em> is for items you are required to click or select (such as menu items or buttons).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mouse Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RMB and MMB</strong></td>
</tr>
<tr>
<td>The buttons on the mouse will be referred to as MMB (middle mouse button) and RMB (right mouse button). If not specified use the left mouse button.</td>
</tr>
<tr>
<td><strong>Hold down</strong></td>
</tr>
<tr>
<td>press a mouse button and <em>do not release.</em></td>
</tr>
<tr>
<td><strong>Pop-up</strong></td>
</tr>
<tr>
<td>From an item (textual or graphical) RMB to ‘pop-up’ a menu item.</td>
</tr>
<tr>
<td><strong>Drag</strong></td>
</tr>
<tr>
<td>drag graphical items by moving the mouse cursor <em>while depressing</em> the mouse button.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Graphical Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>To add a waypoint to a line</strong></td>
</tr>
<tr>
<td>Use RMB while drawing a line.</td>
</tr>
<tr>
<td><strong>To draw only horizontal and vertical lines</strong></td>
</tr>
<tr>
<td>Hold the Shift key while drawing the line.</td>
</tr>
<tr>
<td><strong>To cancel a graphical operation</strong></td>
</tr>
<tr>
<td>Press Escape</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>The Database Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>To expand/collapse the tree</strong></td>
</tr>
<tr>
<td>Double click the item to expand or single click the + icon.</td>
</tr>
<tr>
<td><strong>To show the Primary view of an item</strong></td>
</tr>
<tr>
<td>Single click the item to show the Diagram or Details frame associated with it.</td>
</tr>
</tbody>
</table>
2 Managing your Database
Creating and Editing an iUML Database

Ensure that you have installed the iUML toolset following the instructions in the installation guide and are able to invoke the iUML tool.

1. Creating an iUML Database

iUML stores the information you enter in a database. The iUML database is a set of folders and files which are held in the directory name you specify.

a. Pull-down File > New Repository ...

b. Navigate to the area you wish to store your database in.

c. Enter the name of the database
   Petrol_Station

d. Click OK.

e. Enter the following fields in the Database Details dialog:
   Pressing the Tab key will cycle through the text entry boxes.

<table>
<thead>
<tr>
<th>In field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Administrator</td>
<td>&lt;Your name&gt;</td>
</tr>
<tr>
<td>Database</td>
<td>Petrol Station</td>
</tr>
<tr>
<td>Description</td>
<td>Petrol Station case study for the iUML tutorial</td>
</tr>
</tbody>
</table>

f. Click OK.

You have now created a database in the directory you specified.

2. Acquiring the model for edit

Before you can edit the items in the database, you must acquire a lock. When you exit the database and re-enter you will need to acquire the model for edit each time.

To acquire the Top Level Lock for the database:
   From the tree RMB pull-down Database name > Acquire Lock > Top Level Lock
Opening, Exiting and Saving Models

In order to work through the tutorial at your own pace, you will need to save your work, exit and re-enter iUML Modeller.

To Save and Exit and iUML Database

a. Pull-down File > Exit.
b. Click Save.
c. You can enter an comment to describe your additions to the database.
d. Click Publish.

Confirm your exit of iUML by clicking OK.

To Open an iUML Modeller Database

e. Start iUML Modeller
f. Pull-down File > Open.
g. Select the location of the iUML Database.
h. Click OK.

To Save an iUML Database

To save changes and continue to work through the tutorial:

a. Pull-down File > Save.
b. You can enter an comment to describe your additions to the database.
c. Click Publish.

Note that after saving you will need to acquire a lock to continue editing.

To Acquire a Lock to Edit

Each area of the iUML database requires a lock so it can be edited. There are a number of locks available, each allowing different levels of access to the iUML model. When you enter the database and need write access to an item:

From the Item Name, RMB pop-up Acquire Lock > Lock Name.

The first item on the list of locks available will give you the correct level of access to perform the edit.
3

From Requirements to Domains

*How to use the iUML tool to construct and modify xUML models.*

When developing large complex software systems, the problem being studied will often include a number of distinctly different and unrelated *subject matters*. For example, in the Petrol Station case study, the subject matters of concern might include:

- **Petrol Station Control**
  This deals with the subject matter of the management and dispensing of fuel, customer payments and assignment of different fuel grades to tanks.

- **Attendant Interface**
  This deals with the subject matter of user interaction with respect to the forecourt attendant.

- **Pump Interface**
  The subject matter of the detailed elements of the Pump hardware including the cost display and motor and clutch control are specified here.

- **Operating System (OS)**
  The run time services provided by the operating system.

In xUML, each subject matter is abstracted as a separate Domain. These Domains can then be analysed independently and may make use of the services provided by other Domains.

A good technique for gaining greater understanding of the system requirements in order to effectively partition the system into Domains is Use Case analysis.

This section of the tutorial explores both iUML’s support for both Use Case modelling and Domain Partitioning.
In this Section you will:

• Create a Project for the study.
• Construct a set of Use Cases for the case study.
• Partition the Petrol Station Control system into a number of subject matter areas (Domains) and represent them on a Domain model.
Projects

Projects are used in iUML to organise the development of applications. To begin analysis of our tutorial’s requirements you will need to create a Project.

The mission statement of a project typically outlines the project requirements and the individuals performing key roles on the project. Each project has one or more versions, typically corresponding to review points.

Add a Project

You can now create the Petrol Station Project.

a. Acquire a lock if necessary,
b. From the Tree RMB pull-down Projects > Add Project > <New>.
c. Fill in the details of the Project as follows:

<table>
<thead>
<tr>
<th>Number</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Petrol Station Project</td>
</tr>
<tr>
<td>Key Letter</td>
<td>PS</td>
</tr>
<tr>
<td>Short Description</td>
<td>Petrol Station Case Study</td>
</tr>
<tr>
<td>Mission Statement</td>
<td>To develop a Petrol Station control model</td>
</tr>
<tr>
<td>Build Area</td>
<td>Leave blank for now</td>
</tr>
</tbody>
</table>

d. Click OK

Expand the tree for Projects to see the Project you have created, click on the project name to see its details frame.

Projects in iUML

Project

A key aspect to any development process is Configuration Management.
iUML allows you to specify the project you are going to develop. You are able to enter any number of projects within the database.

Project Version

As part of the Configuration Management capability, iUML allows you to have one or more Project Version for each Project. Project Versions are considered as assemblies of domain versions.
Use Cases for the Petrol Station Control system

A good place to start when analysing the Petrol Station Control system is to consider the things you would expect to do with the system. In xUML, *Use Cases* describe the ways the system can be used. It is a “black box” view of required functionality of the system.

Initial Use Cases for the Petrol Station Control system will include: ‘Customer Fuels Car’ and ‘Tanker Delivers Fuel’.

Outside the Petrol Station Control system are the customer, the attendant and the tanker driver, in Use Case notation they are the *Actors*. An actor is an external entity that interacts with the system by invoking the behavior described by the Use Cases. An actor may represent:

- A role performed by a person (note that a single person may perform many roles)
- External equipment or other computer systems that interact with the system under analysis.

1. Creating a Use Case Diagram

   a. Expand the tree to *Projects > Petrol Station Project > Initial Version*.
   b. RMB pop-up Initial Version and select *Show Use Case Diagram > <New Diagram>*.
   c. Type the name *Fuel Delivery and Purchases* and click *OK*.

   The diagram frame now shows an empty Use Case Diagram.

Add an Actor

   a. RMB pop-up menu from the Diagram window and select *Add > Actor*.
   b. Enter the name of the Actor *Attendant* and click *OK*.
   c. Click where you want the Actor to be placed.
   d. In the same way, add the Actors, *Customer* and *Tanker Driver*.

Add a Use Case

   a. RMB pop-up menu and select *Add > Use Case*.
   b. LMB click where you want the Use Case to be placed.
   c. Enter the name of the Use Case *Customer Fuels Car* and click *OK*.
   d. Now add the Use Case *Fuel Delivery* in the same way.

Add a Communication

A communication signifies a link between an Actor and a Use Case.

   a. LMB select the Actor *Attendant*.
   b. RMB pop-up menu and select *Add > Communication*.
   c. LMB click on the Use Case *Customer Fuels Car*. 
d. Add further communications from Customer to Customer Fuels Car and from Tanker Driver to Fuel Delivery

Note that Communications can be added in either direction, i.e. they can be added from the Use Case to the Actor, or vice versa.

2. Arrange the Use Case Diagram

To arrange the Use Case diagram, select Actors and Use cases and drag them into position on the screen. Hold the MMB and drag the target to view different areas of the model.

Use Cases should be moved to the middle of the diagram and Actors placed around them.
Creating a Domain Model

In xUML, the system being analysed is separated into subject matters called *domains*. The domains are captured as individual xUML models, each of which addresses one subject matter only, and is uncontaminated by knowledge of other system aspects. This results in components that are exceptionally reusable.

The set of domains that are to be used in a particular project are shown on a Domain Model as UML packages. The domain model also shows the client-server Dependencies which exist between the domains.

<table>
<thead>
<tr>
<th>Domain Name</th>
<th>Domain Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Petrol Station Control</td>
<td>Application</td>
</tr>
<tr>
<td>Attendant Interface</td>
<td>Service</td>
</tr>
<tr>
<td>Pump Interface</td>
<td>Service</td>
</tr>
<tr>
<td>SWA</td>
<td>Architecture</td>
</tr>
<tr>
<td>C</td>
<td>Implementation</td>
</tr>
<tr>
<td>OS9</td>
<td>Implementation</td>
</tr>
<tr>
<td>Hardware Interface</td>
<td>Implementation</td>
</tr>
<tr>
<td>GUI</td>
<td>Service</td>
</tr>
</tbody>
</table>

**Add a Domain to the Domain Model**

The aim of this exercise is to represent the identified Domains on a Domain Model. A domain is represented as a package on the Domain Model.

a. Expand the tree to Projects > Petrol Station Project.
b. Click Initial Version.
c. From an empty space in the Diagram Window and RMB pop-up Add > Domain Package.
d. Enter the name Petrol Station Control and description: The Petrol Station Control domain manages the dispensing of fuel, customer payments and tank levels. in the Domain Details dialog and click OK.
e. Point to where you wish to place a domain symbol and click.

**Domains Types in xUML**

There are four types of domain in xUML:

- **Application Domain**
  
  Represents the purpose of the system from the end users point of view.

- **Service Domains**
  
  Provide generic services to support the application domain

- **Architecture Domain**

  Represent the globally applied design and coding strategies.

- **Implementation Domains**

  Represent pre-existing software components, either bought-in or “legacy” code.
From the description of the Petrol Station Control system we find that it separates into the following domains.

1. Add these domains to the model:

<table>
<thead>
<tr>
<th>Domain Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attendant Interface</td>
<td>Provides interaction with the petrol station forecourt attendant.</td>
</tr>
<tr>
<td>Pump Interface</td>
<td>Provides interaction with the elements of the pump hardware, including the cost display, the motor and the clutch.</td>
</tr>
<tr>
<td>SWA</td>
<td>The Software Architecture deals with the architectural mechanisms and the translation rules for the target.</td>
</tr>
<tr>
<td>C</td>
<td>The c programming language</td>
</tr>
<tr>
<td>OS9</td>
<td>The OS/9 run time executive.</td>
</tr>
<tr>
<td>Hardware Interface</td>
<td>Implementation software that drives the hardware at the lowest level. (This may also be legacy code from the existing system.)</td>
</tr>
<tr>
<td>GUI</td>
<td>The Graphical User Interface</td>
</tr>
</tbody>
</table>

Adding a Domain Dependency to the Domain Model

The domains are organised on the Domain Chart into a hierarchy of Client Server Dependencies. In each Client/Server pair the Client is the Domain which requires the services of the Server in order to fulfil its mission.

Each dependency between a Client/Server Domain Pair has a description which explains the purpose of the dependency.

Now that the Domains have been added to the Domain Model you can add the Client/Server Dependencies.

In the Diagram window:

a. From the Client Domain Petrol Station Control RMB pop-up Add > Dependency.
b. Click on the server domain Attendant Interface.

The Client/Server dependency has now been added to the diagram.

Adding a Description to the Dependency

a. From the Petrol Station Control to Attendant Interface Dependency line on the Package Diagram, RMB pull-down Modify > Edit Description.
b. Enter the following description in the edit field and click OK.
The Petrol Station Control domain uses the capabilities of the Attendant Interface domain to:

- Present Customer transaction to the forecourt attendant
- Allow the forecourt attendant to enable a pump

c. Add the following Dependencies and Descriptions to the diagram.

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Petrol Station Control</td>
<td>Pump Interface</td>
<td>As a tutorial exercise, try writing your own dependency descriptions from the knowledge of the model you have acquired so far.</td>
</tr>
<tr>
<td>Petrol Station Control</td>
<td>SWA</td>
<td></td>
</tr>
<tr>
<td>Attendant Interface</td>
<td>GUI</td>
<td></td>
</tr>
<tr>
<td>Pump Interface</td>
<td>Hardware Interface Driver</td>
<td></td>
</tr>
<tr>
<td>Pump Interface</td>
<td>SWA</td>
<td></td>
</tr>
<tr>
<td>SWA</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>SWA</td>
<td>OS9</td>
<td></td>
</tr>
</tbody>
</table>

Summary

In this section you have:

- Created an iUML Database
- Represented subject areas as Domains on a Domain Chart
- Created Use Cases to represent uses of the system

In the next section you will further analyse the system requirements using Sequence Diagram techniques.
Sequence Diagrams

In the last section (Getting Started & Building Models) you learnt about partitioning the system into subject matters (Domains). You were also introduced to the technique of modelling Use Cases.

An important objective in the early stages of an xUML project is to gain a high level of confidence in the domain split you have decided upon, a mistake at this stage might result in extensive rework later.

One of the techniques the you can employ to reduce the risk of incorrect domain partitioning is the class blitz, that is a rapid xUML modelling exercise designed to dig deeper into the subject matter of the various domains. A weakness of this technique however is that is tends to focus on the static nature of each domain rather than their dynamic behaviour.

Domain Interaction Analysis is a complimentary technique which allows the project team to analyse the dynamics of the complete system of domains. iUML supports this approach through the provision of Sequence Diagrams.

Sequence Diagrams bring together the work already done on Use Cases and Domain Partitioning. Using the sequence diagram the analyst models the interactions between selected domains necessary to provide the behaviour specified in a particular Use Case.

So, returning to our Petrol Station Example, or the “Customer Fuels Car” Use Case we would expect interactions between domains such as:

- Pump Interface tells Petrol Station Control that the Gun has been removed
- Petrol Station Control tells Attendant User Interface to request the pump to be enabled
- Attendant User Interface accepts a Pump Enabled message from the Attendant
Sequence Diagrams provide a powerful modelling technique with the following benefits:

- Early exploration of the dynamic interactions between domains.
- Reduced risk of incorrect domain partitioning
- Provides traceability from use cases to domain models
- Leads to the definition of the provided and required interfaces for each domain
- Documents the system design by answering the question ‘how does the system do this?’
- Forms the basis for planning Multi-Domain Simulation and Integration testing
Creating Sequence Diagrams

Typically a Sequence Diagram captures the system threads which correspond to a single Use Case. In this tutorial we consider the Use Case of a customer fueling their car and trace the interactions between domains in the system and the outside world.

On the Sequence Diagram, the outside world is represented by a thick vertical bar on the left hand side of the diagram which is called the System Boundary.

In this tutorial, the sequence diagram you create will be for the Customer Fuels Car thread.

To Show the Sequence Diagram

1. Expand the tree to Projects > Petrol Station Project > Initial Version > Use Case Model > Use Cases > Customer Fuels Car > Sequence Diagrams.
2. Single click Customer Fuels Car.
3. On the Diagram window, click the pencil icon to acquire the lock if required.

The Diagram Frame will display an empty sequence diagram for Customer Fuels Car, containing a vertical Boundary Line.

Add a Lifeline

A communication between a pair of domains or between a single domain and the system boundary is known as an interaction. An interaction may be synchronous or asynchronous. Typically each interaction will have an associated required and provided interface. The flow of control is from required to provided. Interactions may be named after either their associated required or provided service or alternatively they may be given their own label.

Each domain required to support the behaviour of the Use Case is depicted on the sequence diagram as a vertical line with the domain name at the top.

A Lifeline represents a Domain involved in the Use Case.

1. From an empty area of the Sequence Diagram, RMB pop-up menu and select Add > Lifeline.
2. Select the domain name Pump Interface from the pick list.
3. Place the lifeline a few cms to the right of the Boundary Line.

Repeat this for the Attendant Interface and Petrol Station Control domains.

Add an Interaction

Each interaction represents a service required by the source domain and a service provided by the destination domain or system boundary. In this exercise you will
add the sequence of domain interactions for part of the thread corresponding to the ‘Customer Fuels Car’ Use Case.

a. LMB Select the source Lifeline or Boundary Line - in this case, select the Boundary Line.
b. RMB pop-up menu Add > Interaction.
c. LMB Click on the destination lifeline - in this case the Pump Interface.
d. Select New Interaction from the pick list.
e. Enter the Interaction Name Gun Removed from Holster (you can leave the Provided and Required service fields blank) and click OK.
f. Enter the corresponding statement text Customer removes Gun from Pump and click OK.
g. Repeat this sequence for the interaction from Pump Interface to Petrol Station Control called Gun Removed, leaving the Statement text blank.

Add a Statement

So that you can understand complex sequence diagrams, you can add a structured commentary to the left hand side of the diagram. This commentary is made up of statements which amplify the diagram interactions using constructs such as if, then and else.

You can build a sequence diagram in a number of ways. In the previous steps, you added two interactions and documented each with a statement (the second statement was left blank). However, an alternative style of working is to provide sections of the Statement ‘commentary’ before specifying the interactions. iUML supports both ways of working and to illustrate this you will now add a statement without specifying any associated interaction.

a. LMB select the vertical dotted Statements ‘width bar’.
b. Ctrl. RMB pop-up Add > Statement.
c. Select <End> from the Position before Statement... pick list.
d. Enter the statement If fuel level low and click OK.

The other Interactions are as follows, add as many as you feel is necessary to familiarise yourself with the process.

<table>
<thead>
<tr>
<th>No.</th>
<th>From</th>
<th>To</th>
<th>Interaction Name</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Petrol Station</td>
<td>Pump Interface</td>
<td>Pump Unavailable</td>
<td>Inform Customer Fuel not available</td>
</tr>
<tr>
<td>5</td>
<td>Pump Interface</td>
<td>Boundary</td>
<td>Set Pump Unavailable Indicator</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>Else</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Petrol Station</td>
<td>Attendant User</td>
<td>Request Pump Enable</td>
<td>Request Attendant to enable Pump</td>
</tr>
<tr>
<td>8</td>
<td>Attendant User</td>
<td>Boundary</td>
<td>Set Pump Icon Waiting</td>
<td>Enable</td>
</tr>
<tr>
<td>9</td>
<td>Boundary</td>
<td>Attendant User</td>
<td>Pump Enables</td>
<td></td>
</tr>
<tr>
<td>No.</td>
<td>From</td>
<td>To</td>
<td>Interaction Name</td>
<td>Statement</td>
</tr>
<tr>
<td>-----</td>
<td>-----------------------------</td>
<td>---------------------------</td>
<td>----------------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>10</td>
<td>Attendant User Interface</td>
<td>Petrol Station Control</td>
<td>Create Delivery</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Petrol Station Control</td>
<td>Pump Interface</td>
<td>Start Motor</td>
<td>Start Motor on ready to dispense fuel</td>
</tr>
<tr>
<td>12</td>
<td>Boundary</td>
<td>Pump Interface</td>
<td>Trigger Pressed</td>
<td>Customer Presses Trigger on Gun</td>
</tr>
<tr>
<td>13</td>
<td>Pump Interface</td>
<td>Petrol Station Control</td>
<td>Trigger Depressed</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Petrol Station Control</td>
<td>Pump Interface</td>
<td>Engage Clutch</td>
<td>Start Pumping Fuel</td>
</tr>
</tbody>
</table>
Arranging the Sequence Diagram

As with all iUML diagrams, the middle mouse button dragging and scroll bars enable you to move about the diagram frame and the zoom facilities allow you to see the whole diagram.

On the Sequence Diagram, you can also:

- LMB select and drag the vertical Lifelines and Width Bar to make more space available for the interaction names.
- LMB select and drag the Interaction names to reposition them with respect to their lines.

Experiment with these features to arrange the diagram.

Summary

In this section you have:

- Created a Sequence diagram showing the interactions between the domains
- Documented each of the interactions on the sequence diagram.

In the next section, you will create a Static Model for the Petrol Station Control Domain.
5

Static Modelling

Domain Modelling with iUML

So far you have seen how a set of Use Cases:

• Expose the usage of the Petrol Station Control system from an external viewpoint.
• Supplement the system requirements so that the initial domain partitioning can be specified using a Domain model.
• Describe the interactions amongst the domains (domain level threads), using a number of sequence diagrams.

From this analysis, an analyst may:

• make a qualitative assessment of the viability of the domain model;
• establish the interfaces between the domains;
• identify the required dependencies between domains.

What you will do in this section

• Learn about the purpose of Static Modelling
• Add Classes, Associations, Attributes and Identifiers to the Static Model
• Explore the viewing and diagram editing facilities of iUML

Building a Static Model

For the purposes of this tutorial, we decide that the initial domain partitioning is viable and so the analysis of each of the domains can begin on the basis of this decision.

The first stage of the analysis of each domain is to build the Static Model.

In this section of the tutorial you will build the static model for the Petrol Station Control Domain.
We identified the purpose of the Petrol Station Control domain to be to track the delivery of fuel to the motorists. So, the system needs to record deliveries. Each delivery that is made has the same characteristics, but different values, like; the time it was made, how much fuel was dispensed and what type of fuel it was. The things that describe the characteristics of the classes in the system are called attributes.

For a particular delivery of fuel, the pump used to deliver it must be recorded, this is an association between the delivery and the pump.

Accordingly you will be adding classes to the Static Model, linking them with various types of associations and specifying attributes.

---

**Static Model Overview**

The purpose of the Static Model is to describe the state of the problem domain at any point in time. The data described by the model will change over time as the system executes. A static model is represented on a Class Diagram which is enhanced with supportive descriptions and data definitions. By contrast, the Dynamic model describes the events and operations which will cause the data described in the static model to change.

A Class Diagram depicts the classes in the domain. Classes represent the conceptual entities which make up the subject matter under consideration. They each possess pertinent characteristics, which are depicted as Attributes, and they participate in relevant associations, which are depicted as links between the classes.

**Identifying Classes**

When identifying candidate classes, it is useful to have a few “mental hooks”. These provide a framework for building a Static Model in a strategic way. The xUML method places classes into five categories:

- **Tangible**
  These are the “obvious” classes. They jump out of the problem. An analyst modelling an Air Traffic Control system would find it hard to miss the “Aircraft” class. In an engine management system, they would be classes like *Actual Cylinder* and *Actual Valve*.

- **Role**
  These classes represent roles performed by people or things. They capture the attributes and associations specific to that role. For example, a *Actual Valve* class might perform the role of an *Open Valve* and a *Closed Valve*. Role classes typically manifest themselves as subtypes, as illustrated later.

- **Incident**
  Incident classes are abstractions of signals that the system needs to capture. They may represent past incidents, or signals that are to occur in the future. For example, in an engine management system there may be incident classes such as *Previous Cylinder Cycle*, recording the efficiency of the previous combustion cycle for use in calculating values for the next one, and *Pending Fuel Injection*, representing an incident that must occur within the next few milliseconds.

- **Interaction**
  Interaction classes arise as a result of an association between other classes. For example, a *Fuel Supply Path* represents an interaction between a *Fuel Pump* and a *Fuel Injector*.

- **Specification**
  Specification classes capture common qualities possessed by a set of objects of another class. For example, the *Type of Cylinder* class captures common attributes of *Actual Cylinder*, such as *swept volume*. All objects of *Actual Cylinder* of a given *Type of Cylinder* have the same value for *Swept Volume*.
Domain Creation

Each of the Domains on the Domain model will require a corresponding Domain within the database in which the analysis can be conducted - as such you will be required to add one or more domains to the database.

In this exercise you will add the 'Petrol Station Control' domain to the database which has been identified on the Domain model.

Add a Domain to the database

a. Select Domains > Acquire Lock > Top Level if necessary.
b. Select Domains > Add Domain > <New>.

Fill in the details of the Domain as follows:

| Number | 3 |
| Name   | Petrol Station Control |
| Key Letter | PSC |
| Short Description | Petrol Station Application Domain |
| Mission Statement | To control the dispensing of fuel, customer payments and tank levels. |
| Type | Application |
| Initial Version | Standard |
| Build Area | leave blank |

c. Click OK to add the Domain to the Database.
Classes

The first stage of editing the static model involves adding classes to the Class Diagram and the associations between them.

1. Showing the Class Diagram

The Class Diagram ‘belongs’ to the domain version.

   a. Expand the tree to Domains > Petrol Station Control.
   b. Click Initial Version.

The Class Diagram, which will be initially blank, is shown in the Diagram Window.

2. Adding a Class

This is the graphical representation of the TANK class:

   a. On an empty area of the Diagram Window RMB pop-up Add > Class.
   b. Enter the Class Details for the TANK class in the table and click OK.
   c. Click to place the Class
   d. Repeat for the other classes shown in the table.

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Key Letter</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>TANK</td>
<td>TNK</td>
</tr>
<tr>
<td>3</td>
<td>PUMP</td>
<td>PMP</td>
</tr>
<tr>
<td>2</td>
<td>DELIVERY</td>
<td>DLV</td>
</tr>
<tr>
<td>5</td>
<td>FUEL GRADE</td>
<td>FGR</td>
</tr>
<tr>
<td>6</td>
<td>TRANSACTION</td>
<td>TRN</td>
</tr>
</tbody>
</table>
Attributes

The analyst must specify all the pertinent characteristics of each class. Each characteristic is represented as an attribute.

Attributes describe the characteristics of a class in the ‘real world’, for example, the Petrol Station will have two tanks which have following characteristics when the system is first executed.

<table>
<thead>
<tr>
<th>TANK Number</th>
<th>TANK capacity</th>
<th>TANK level</th>
<th>TANK empty flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100000</td>
<td>0 (the tank is initially empty)</td>
<td>True</td>
</tr>
<tr>
<td>2</td>
<td>200000</td>
<td>0 (the tank is initially empty)</td>
<td>True</td>
</tr>
</tbody>
</table>

iUML has a method of capturing this information (Test Methods) which we will encounter later in this tutorial.

1. Adding Attributes to Classes

   a. From the Class RMB pop-up Add > Attribute.
   b. Enter the attributes and details from the table and click OK (leaving the Default Value blank).

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tank Number</td>
<td></td>
<td>Integer</td>
</tr>
<tr>
<td>Tank Capacity</td>
<td>Volume of tank in litres.</td>
<td>Real</td>
</tr>
<tr>
<td>Tank level</td>
<td>Litres of fuel in tank.</td>
<td>Real</td>
</tr>
<tr>
<td>Tank Empty Flag</td>
<td>Set to true when tank level falls below the Empty Threshold.</td>
<td>Boolean</td>
</tr>
<tr>
<td>Empty Threshold</td>
<td>If the tank level falls below this threshold the forecourt attendant is prevented from enabling pumps supplied by that tank.</td>
<td>Real</td>
</tr>
</tbody>
</table>

   c. Here are the attributes of the other classes in the diagram.

FUEL GRADE

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grade Name</td>
<td>Fuel type and identifier</td>
<td>Text</td>
</tr>
<tr>
<td>Unit Price</td>
<td>Price per litre in pounds sterling</td>
<td>Real</td>
</tr>
</tbody>
</table>

PUMP

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pump Number</td>
<td>Pump identifier</td>
<td>Integer</td>
</tr>
</tbody>
</table>
In modelling many types of domain it is useful to be able to model the nature of the identity of the objects of a given class. For example, if we were tracking the ownership of motor cars, it is imperative to know how an individual Car is identified. For example, in the UK, cars carry highly visible Registration Plates containing a unique “Registration Number”. This serves to identify a given vehicle for most purposes. However, under some circumstances, it is permitted to change the Registration Number of a given Car. Does this mean that this is now a different car? The answer (which in this case is “No”) depends on the nature of the problem domain and the purpose of the system being built.

iUML allows analysts to capture such domain knowledge through the concept of “Identifiers”. An identifier is a collection of one or more attributes that will uniquely identify an instance of the class. In the case of the Car class, there will be an identifier with one attribute (“Vehicle Identification Number”).

Often there is more than one way of identifying a instance of a class. iUML thus allows the definition of multiple alternative identifiers. Of these, one is always designated as the “Preferred” by the analyst. The preferred identifier is presented by default on some model views.

Such identifiers are also used by iUML to “formalise” associations by the automatic creation of Referential Attributes in associated classes. Such attributes serve to further emphasise the nature of an association either by renaming of them or by merging of them with other referential attributes.

Since there are also types of domains where the identity of objects is in some sense “arbitrary”, the use of explicitly modelled identifiers in iUML is optional and
may be omitted if desired. In this case, the xUML formalism supported by ASL will manage the identity of object in an automatic fashion.

a. From the attribute RMB pop-up **Modify > Add to Preferred Identifier**.

b. Here are the other attributes which are the preferred identifiers of their Classes:

<table>
<thead>
<tr>
<th>Class</th>
<th>Attribute to make preferred</th>
</tr>
</thead>
<tbody>
<tr>
<td>TANK</td>
<td>Tank Number</td>
</tr>
<tr>
<td>FUEL GRADE</td>
<td>Grade Name</td>
</tr>
<tr>
<td>TRANSACTION</td>
<td>Transaction number</td>
</tr>
<tr>
<td>DELIVERY</td>
<td>Delivery Time</td>
</tr>
<tr>
<td>PUMP</td>
<td>Pump number</td>
</tr>
</tbody>
</table>

The attributes which have now been added to the preferred identifier.

Normally the properties which show which Attributes are in a preferred identifier are hidden. They can be seen on the Class Diagram by:

a. Choose a class to change the display properties for.

b. RMB pop-up **Display > Attribute > Properties: Visible** from pick list.

e.g. If the TANK class was selected, in the attribute compartment the attribute Tank Number would be displayed as: `Tank Number (I= (1))`

Which indicates that the Tank Number is the first identifier and is the preferred identifier.
Associations

So far we have defined the 'information' relating to the domain in terms of classes and attributes. The relationships between the classes are equally important since these provide information about the way objects of the classes (objects) are related to one another. Such relationships are modelled as Associations between the respective classes. For example - there is an association between TANK and PUMP.

Every association has a unique association number which is automatically assigned by iUML. At each end, the association has:

- **a role phrase**
  Every association has a pair of role phrases, indicating the meaning of the association from each class’s point of view.

- **multiplicity**
  Each end of an association has a multiplicity string indicating how many objects of one class can participate with respect to a single object of the other class.

Additionally the association can be documented with an association description.

There are two useful features worth noting while adding associations to the diagram.

- **The shift key**
  Hold the shift key to enable you to draw lines perpendicular to the classes you have created.

- **Waypoints**
  After selecting the Association option from the pop-up menu, press RMB to insert a Waypoint on the line. Use RMB while holding the shift key to add 90° corners to the line.

**Add an Association between TANK and PUMP**

a. From the PUMP class RMB pop-up **Add > Association**
b. Click on the TANK class.
This results in the Association Details Dialogue appearing:

![Association Details Dialogue]

**c.** Enter the details from table into the Association Details dialogue and click **OK**:

<table>
<thead>
<tr>
<th>Class</th>
<th>Conditionality</th>
<th>Role</th>
<th>Multiplicity</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUMP (A)</td>
<td>Unconditionally</td>
<td>is pumping fuel from</td>
<td>one</td>
<td>TANK (B)</td>
</tr>
<tr>
<td>TANK (B)</td>
<td>Conditionally</td>
<td>is providing fuel for</td>
<td>many</td>
<td>PUMP (A)</td>
</tr>
</tbody>
</table>

**Type:** Referential  
**Formalised in:** A  
**B Identifier:** P

**d.** You can also add the other two associations between TANK & FUEL GRADE and DELIVERY & TRANSACTION using the details from the following tables.

<table>
<thead>
<tr>
<th>Class</th>
<th>Conditionality</th>
<th>Role</th>
<th>Multiplicity</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>TANK (A)</td>
<td>Unconditionally</td>
<td>stores</td>
<td>one</td>
<td>FUEL GRADE(B)</td>
</tr>
<tr>
<td>FUEL GRADE(B)</td>
<td>Conditionally</td>
<td>is stored in</td>
<td>one</td>
<td>TANK (A)</td>
</tr>
</tbody>
</table>

**Type:** Referential  
**Formalised in:** A  
**B Identifier:** P

<table>
<thead>
<tr>
<th>Class</th>
<th>Conditionality</th>
<th>Role</th>
<th>Multiplicity</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUMP (A)</td>
<td>Conditionally</td>
<td>is being used for</td>
<td>one</td>
<td>DELIVERY (B)</td>
</tr>
<tr>
<td>DELIVERY (B)</td>
<td>Unconditionally</td>
<td>is being made using</td>
<td>one</td>
<td>PUMP (A)</td>
</tr>
</tbody>
</table>

**Type:** Referential  
**Formalised in:** B  
**A Identifier:** P
Reviewing the Preferred Identifiers

All classes should now have attributes which identify them. The DELIVERY class, however cannot be uniquely identified by its present attribute Time as two deliveries may occur simultaneously at different pumps. Another attribute must be added to DELIVERY’s preferred identifier. In this case it is the Pump Number attribute.

Pump Number is a referential attribute in DELIVERY via the association R3.

Add Pump Number to the Preferred Identifier

From Pump Number in DELIVERY, RMB pop-up menu Modify > Add to Preferred Identifier.

Pump Number will now be added to the preferred identifier of DELIVERY.
Generalisations

So far, we have modelled a transaction as a single class (TRANSACTION). There are, however, three distinct types of transaction that the model needs to deal with:

- Pending Transactions
- Paid Transactions
- Evaded Transactions

There are a number of characteristics that are common to each type of transaction and, in the case of an evaded transaction, the attendant is required to note all relevant observations (Registration, Description of driver etc.). The different types of transaction are modelled in a Super/Sub Class Generalisation.

1. Add the Super/Sub Classes and their Attributes

Super/sub classes are added in the same way as other classes. The superclass will contain the attributes common to all the subclasses.

To create the Transaction super/sub class group:

a. Add the three classes with Names, Key Letters and Numbers to the Class Diagram.

<table>
<thead>
<tr>
<th>Name</th>
<th>Key Letter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PENDING TRANSACTION</td>
<td>PND</td>
<td></td>
</tr>
<tr>
<td>PAID TRANSACTION</td>
<td>PDT</td>
<td></td>
</tr>
<tr>
<td>EVADED TRANSACTION</td>
<td>EVT</td>
<td></td>
</tr>
</tbody>
</table>

b. Add the attribute Observations to the Evaded Transaction class (needed to describe the evading customer - Registration number, make of car, driver description etc.).

2. Add the Generalisation

The first generalisation drawn from subclass to the superclass defines the new subclass family. Further relationships from sub to super-class can be added to the existing link.

To create a Generalisation:

a. From PENDING TRANSACTION class and RMB pull-down Add > Generalisation.

b. Click on the TRANSACTION class to create the association between them.
Generalisations

To add the other Subclasses to the super/sub class hierarchy:

a. From the classes RMB pull-down Add > Generalisation.

b. Click on TRANSACTION and select the existing Generalisation from the pick list.

Notice that the preferred identifier of the superclass is propagated to each of the subclass (as referential attributes) and becomes the default preferred identifier.
Adding New Types

iUML provides a number of pre-defined data types which include Integer, Real and Text. During the course of analysis you may find that you need to create your own Types in the model.

At present, the TRANSACTION class is not ‘aware’ of which type of transaction is taking place. By adding an attribute with your own Type, this information can be specified within TRANSACTION.

‘Transaction Subtype’ will be of the base type ‘Enumeration’ and will have three values.

- Paid
- Pending
- Evaded

Create an Attribute with a New Type

a. From the TRANSACTION class, RMB pull-down Add > Attribute.
b. Enter the name Transaction Subtype.
c. Click Select...
d. Select <New Type>.
e. Complete the dialog as follows and click OK.
Add the Enumeration Values for the Type

a. Expand the tree to Domains > Petrol Station Control > Initial Version > User Defined Types.
b. Single click Transaction Subtype to show its details window.
a. From Transaction Subtype RMB pop-up Add > Enumeration Value.
b. Enter the Name as Paid and click OK.
c. You can repeat this to add the Pending and Evaded values.
Summary

In this section on Static Modelling you have:

• Seen how classes are identified
• Created a Class Diagram with Classes, Associations, Attributes and Identifiers.
Introduction to the Class Collaboration Diagram

In this section we move from satic modelling to dynamic modelling, but before specifying class behavior in detail, it is important to establish:

- The primary responsibilities of each class.
- The interface provided by each class.
- The way in which the classes will interact.

This is achieved by building the Class Collaboration Diagram. This provides a graphical summary of all the interactions between the classes. The interactions will be in one of two forms:

- **Asynchronous Interactions**
  These are modelled using Signal Transmissions.

- **Synchronous Interactions**
  These are modelled with Operation Invocations.

The behaviour of a class may be defined as:

- **State Dependent**
  Where the class behaviour exhibits a ‘life cycle’ form which is defined in a State Machine. It is this state machine that defines the responses to the respective Asynchronous Interactions. In the case study, Tank, Pump, Delivery and Transaction, have state dependant behaviour.

- **State Independent**
  The classes behaviour is expressed as a set of provided operations (analogous to a function/procedure/subroutine in most programming languages). It is these services that define the responses to the respective synchronous interactions.
This section deals with the interactions between classes, allowing them to be visualised as ‘black boxes. The next section will show how state machines can be used to present a ‘white box’ view of how the interfaces are realised.

This is an example of how the customer interacts with the Petrol Station:

1. The customer drives into the station and parks at a free pump.
2. The customer removes the gun from the pump holster.
3. The attendant then enables the pump which creates a delivery and starts the pump motor.
4. The customer presses the trigger on the gun...
5. ... which engages the clutch
6. ... and delivers fuel via a meter.
7. When the customer has delivered enough fuel they release the trigger...
8. ... which disengages the clutch
9. They return the gun to the pump holster.
10. This signals the end of the delivery, the motor stops and a transaction is created.
11. The system then waits for the customer to pay for the fuel.
12. The customer pays for their fuel, or absconds without paying

**Classes on the Class Collaboration Diagram**

The Class Collaboration Diagram summarises the interactions in terms of:

- A class.
- A Terminator - which is an abstraction of something outside this domain.
- A half arrow head represents an asynchronous Signal Transmission from one state machine to another.
- A full arrow head represents a synchronous Operation Invocation. Note that the direction of the arrowhead reflects the invocation direction, not the data flow.
Editing the Class Collaboration Diagram

1. Adding Object State Machines

Each class in the domain can have one object state machine. Some Association Classes may also have an Assigner State machine (these are not dealt with in this tutorial). In this exercise you will add object state machines to some of the classes in the domain. Any changes made here are automatically updated throughout the database.

Add a State Machine to the Pump class

a. Expand the tree to Domains > Petrol Station Control > Initial Version.
b. RMB pull-down Initial Version > Show Class Collaboration Diagram.
c. In the Diagram Window, RMB pop-up PUMP class > Modify > Create State Machine.
d. Confirm by clicking Yes.

The PUMP class now has <<state machine>> annotated at the top of the class. This indicates that the PUMP class has a State Machine.

e. You can also add state machines to the classes TRANSACTION, TANK and DELIVERY.

Dynamic Model

The Dynamic Model outlines the behaviour of the system in response to signals and invocations.

The Dynamic Model models real world policies and behaviour, moving closer to the implementation of domain requirements.

The modelling components for a dynamic model are State Machines, Signals and Operations.

These components are represented on Class Diagrams (operations), the Class Collaboration Diagram, State Charts, State Tables and Sequence Diagrams.
2. Arranging the Class Collaboration Diagram

- **Lots of Transmissions - large class**
  The more transmissions a class has, the larger it should be on the Class Collaboration Diagram, you can see this with the PUMP Class. This has two benefits, it’s easier to add transmissions to it and it stands out as an important class.

- **Right to Left arrangement**
  The Class Collaboration Diagram is arranged from left to right in order of ‘intelligence’. This means that Classes and Terminators with most responsibility are positioned on the left of the diagram and those with least responsibility are moved to the right giving rise to a layered set of classes.

- **Horizontal Transmissions**
  Where possible, transmission lines should be horizontal. Classes should be moved or resized to accommodate the transmissions.

A useful quality check that can be applied to the Class Collaboration Diagram is to verify that there is an organised ‘layering’ of Class responsibilities within the Domain and that Interactions generally only span adjacent layers.

3. Adding Terminators to the Class Collaboration Diagram

A terminator is the abstraction of an external entity being controlled or monitored by the domain of concern. The terminator represents the ‘ultimate’ destination or source of interactions that take place with that domain.
In the Petrol Station Control system, the 'Petrol Station Control' domain depends upon the 'Attendant Interface' domain to support interactions with the forecourt attendant. The ultimate source and destination of such interactions in this case is the attendant, and as such the terminator used to represent this external entity from the perspective of 'Petrol Station Control' should be named using the language of the subject matter of the domain in which it resides - we have proposed 'Attendant' in the following part of the tutorial.

Whilst naming the terminator something like 'Attendant Interface' would not strictly be incorrect, it would be considered poor modelling practice - after-all, in another version of the Petrol Station Control system the 'Attendant Interface' might be replaced by a different domain with a similar mission.

### Add a Non-Counterpart Terminator

a. RMB click on an empty space on the left of the Class Collaboration Diagram. Pop-up **Add > Terminator**.
b. Enter the details in the Terminator Details dialogue.
c. Click to place the Terminator on the Class Collaboration Diagram.
d. A rectangle will be annotated with `<Terminator>` at the top of the rectangle on the Class Collaboration Diagram.

<table>
<thead>
<tr>
<th>Name</th>
<th>Key Letter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATTENDANT</td>
<td>AT</td>
<td>The forecourt attendant who enables the pumps, deals with payments and makes observations of evading customers.</td>
</tr>
<tr>
<td>MOTOR</td>
<td>MO</td>
<td>The motor which is engaged when the attendant enabled the Pump</td>
</tr>
<tr>
<td>CUSTOMER</td>
<td>CU</td>
<td>Person at petrol station attempting to fuel car.</td>
</tr>
<tr>
<td>METER</td>
<td>ME</td>
<td>Represents the device measuring amount of fuel delivered.</td>
</tr>
<tr>
<td>CLUTCH</td>
<td>CL</td>
<td>The clutch engaged when customer depresses trigger on Gun.</td>
</tr>
<tr>
<td>TANKER OPERATOR</td>
<td>TO</td>
<td>Represents person and associated equipment used to re-fuel tanks.</td>
</tr>
<tr>
<td>GUN</td>
<td>GU</td>
<td>Represents Gun at the end of the fuel hose with trigger to engage Clutch</td>
</tr>
<tr>
<td>HOLSTER</td>
<td>HO</td>
<td>Represents the place where Gun is removed from and replaced to.</td>
</tr>
</tbody>
</table>

### 4. Adding Interactions

You will now add Signal Transmissions and Operation Invocations to the Class Collaboration Diagram.
**Add a Signal Transmission**

a. From Class Terminator from which the signal is to be transmitted, RMB pop-up `Add > Send Signal`.
b. Click on Class Terminator to transmit to.
c. Click `<New Signal>`.
d. Enter the name of the Signal Transmission and click `OK`.

The Signal Transmission is added to the Class Collaboration Diagram.

You can also add the Transmissions to the Class Collaboration Diagram as shown in the table following:

<table>
<thead>
<tr>
<th>No.</th>
<th>From</th>
<th>To</th>
<th>Signal Transmission Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>HOLSTER</td>
<td>PUMP</td>
<td>Gun Removed</td>
</tr>
<tr>
<td>2</td>
<td>ATTENDANT</td>
<td>DELIVERY</td>
<td>Create Delivery</td>
</tr>
<tr>
<td>3</td>
<td>DELIVERY</td>
<td>PUMP</td>
<td>Pump Enabled</td>
</tr>
<tr>
<td>4</td>
<td>GUN</td>
<td>PUMP</td>
<td>Trigger Depressed</td>
</tr>
<tr>
<td>5</td>
<td>METER</td>
<td>DELIVERY</td>
<td>Fuel Unit Delivered</td>
</tr>
<tr>
<td>6</td>
<td>GUN</td>
<td>PUMP</td>
<td>Trigger Released</td>
</tr>
<tr>
<td>7</td>
<td>HOLSTER</td>
<td>PUMP</td>
<td>Gun Replaced</td>
</tr>
<tr>
<td>8</td>
<td>PUMP</td>
<td>DELIVERY</td>
<td>Delivery Complete</td>
</tr>
<tr>
<td>9</td>
<td>ATTENDANT</td>
<td>TRANSACTION</td>
<td>Payment Received</td>
</tr>
</tbody>
</table>
Add an Operation Invocation

a. From the Class Terminator from which the Operation Invocation is to be transmitted, RMB pop-up Add > Call Operation.
b. LMB click on Class Terminator to transmit to.
c. Click New Operation.
d. Enter the name of the Operation Invocation and click OK.

The Operation Invocation is added to the Class Collaboration Diagram.

Add the Invocations to the Class Collaboration Diagram as shown in the table following:

<table>
<thead>
<tr>
<th>No.</th>
<th>From</th>
<th>To</th>
<th>Service Invocation Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PUMP</td>
<td>ATTENDANT</td>
<td>Request Pump Enable</td>
</tr>
<tr>
<td>2</td>
<td>PUMP</td>
<td>MOTOR</td>
<td>Start Motor</td>
</tr>
<tr>
<td>3</td>
<td>PUMP</td>
<td>CLUTCH</td>
<td>Engage Clutch</td>
</tr>
<tr>
<td>4</td>
<td>PUMP</td>
<td>CLUTCH</td>
<td>Disengage Clutch</td>
</tr>
<tr>
<td>5</td>
<td>PUMP</td>
<td>MOTOR</td>
<td>Stop Motor</td>
</tr>
<tr>
<td>6</td>
<td>DELIVERY</td>
<td>TRANSACTION</td>
<td>Create Transaction</td>
</tr>
<tr>
<td>7</td>
<td>TRANSACTION</td>
<td>ATTENDANT</td>
<td>Transaction Pending</td>
</tr>
</tbody>
</table>

Operations

Within xUML, an Operation is a section of processing that will execute on request, optionally return data, and then terminate. The caller of the operation waits while the operation is being executed. In an xUML domain model there will be one definition and many invocations per operation.

Within ASL, operations are implemented by functions.

A function is a section of processing that can be called via a parameterised interface. The caller waits until the function is executed and resumes execution after the point at which the call is made. If desired values may be returned through the interface. Functions may invoke other functions, synchronously or provide asynchronous behaviour by generating Signals. In the latter case, the caller will not wait for the results of those Signals being processed.

There are four types of function:

- Domain-based
- Class-based
- Object-based
- Bridge

An ASL function provides a convenient way to encapsulate a set of xUML processes, which can then be invoked from within the processing of any xUML thread of control.

An ASL function is similar in many ways to the concept of a procedure, function, or subroutine found in many programming languages.

At run-time the ASL function is executed synchronously.

- ASL functions may have zero or more input parameters.
- ASL functions may have zero or more output parameters.
- Input and output parameters may be of any valid data type.
5. Adding Operations

You can write Operation Actions for the Operation Invocations you have added to the Class Collaboration Diagram. The Operation you will load for Create Transaction creates a pending transaction which a customer needs to pay for.

a. From the Operation Invocation name Create Transaction on the Class Collaboration Diagram RMB pop-up Show Details.

b. RMB pop-up <Method> text beneath Method and select Load.

c. Navigate to the <iUML release directory>/Manual/2.01/Tutorial_Files/ASL_code directory and double click Trans_operation.txt.

 d. Click OK on the Action Description dialog. The new Action is shown in the Text Window.
Summary

In this section you have:

• Created Object State Machines for the active Classes in the domain.
• Added Operations to Classes.
• Added Terminators to represent external entities.
• Added Interactions between the Classes and Terminators.

In the next section you will edit the Object State Machine for the **PUMP** class.
State Machines in iUML

How to build State Machines with iUML

The Class Collaboration Diagram provides a graphical representation of the interactions between the classes in a domain, in terms of the operation invocations and signal transmissions. The Class Collaboration Diagram only provides a black box view of the classes. The State Machine provides a representation of the way in which each class responds to signals. The State Machine is therefore a white box view of the state dependant interface of a class.

In the Petrol Station Control domain, classes such as TRANSACTION, PUMP, DELIVERY and TANK have interesting life cycles consisting of states and transitions between these states.

These life cycles are modelled using State Machines. In iUML they can be represented in two forms:

- **The State Chart**
  The State Chart provides a clear graphical representation of the state machine. It enables you to see clearly the sequences and cycles of a class.

- **The State Table**
  The State Chart, while clear, does not require you to address every possible combination of state and signal (which you must do to build the complete model). The tabular representation of the state machine used for the State Table supports the complete specification of the state machine.

In the State Table:

- Each row corresponds to a state
- Each column corresponds to a signal
- Each cell represents a potential transition. These cells are referred to as effects.

For the purposes of this Tutorial, we will concentrate purely on defining the State Machine using the State Chart - you should note that iUML ensures consistency between the State Chart and State Table. This means that anything you specify in the State Chart will also be represented in the State Table and vice versa.
State Chart Notation

State Machines contain four main elements, States, Signals, Transitions and Actions.

The State Chart

To build a state machine you will be required to identify the states, signals and transitions.

A good starting point for this is to consider the natural sequence of stages (states) in the behaviour you are trying to model and the incidents (signals) that cause that behaviour to progress through the stages.

Note that some of these Signals may already have been specified on the Class Collaboration Diagram.

Once the ‘normal’ behaviour has been modelled, you can add the states, transitions and signals that deal with errors, failures and unusual or undesired behaviour.

Consider the normal sequence of asynchronous interactions the PUMP class has to respond to:

A customer arrives at the Pump, they remove the gun from the holster. The Pump ‘signals’ to the attendant that the customer wishes to use it. The attendant enables the pump to deliver fuel. The customer then depresses the gun’s trigger, delivers an amount of fuel, releases the trigger and returns the gun to the holster. At this point, the delivery is considered to be over and the pump is automatically disabled. The pump then waits for the attendant to re-enable it for the next customer.
This sequence forms the starting point for the creation of the Pump’s state machine. From this you can investigate and add other sequences.

In this section you will develop the State Chart for the PUMP class.

**Add a State**

A state represents a condition of the class, subject to a defined set of rules, policies or physical laws.

a. Expand the tree to Domains > Petrol Station Control > Initial Version > Classes.
b. From PUMP, RMB pop-up Show State Chart.
c. From the empty chart RMB pop-up Add > State.
d. Enter the state name Waiting For Customer then click OK. The state will be shown on the diagram.
e. LMB click on an area of the frame to place the State.
f. You can add some or all of the states to the diagram in this way. As the states form a cycle of behaviour, add their boxes clockwise round the screen.

```
Name
Waiting For Pump Enable
Ready To Pump
Pumping
Pumping Paused
Fuel Delivery Complete
```

**Edit Action**

An action is the sequence of processing which takes place on entry to a given state.

The state actions entered in this exercise are comments taken from the ASL used by the example model. They represent the processing that would be addressed by ASL statements.

a. Double click the entry/ text in the Waiting For Customer state. This will open a text edit widow for you to enter the Action text.
b. Enter the action comments in the states from the tables following and click OK:

<table>
<thead>
<tr>
<th>State Name</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Waiting For Customer</td>
<td># The Pump is idle. Wait in this state until a customer removes the gun from the holster.</td>
</tr>
<tr>
<td>Waiting For Pump Enable</td>
<td># Determine whether the connected tank contains more than 4% of its capacity</td>
</tr>
<tr>
<td>Ready To Pump</td>
<td># Start pump motor and wait for the gun trigger to be depressed. Or the gun to be replaced</td>
</tr>
<tr>
<td>Pumping</td>
<td># Engage clutch which starts pumping. # Continue until the gun trigger is released.</td>
</tr>
</tbody>
</table>
Summary

Add a Transition

A transition specifies the state which an object will enter from a given state upon receipt of a specific signal.

a. LMB select the Waiting for Customer state and use the RMB to pop-up Add > Transition.

b. Click on the Waiting For Pump Enable state.

A line will appear, showing the transition between Waiting For Customer and Waiting For Pump Enable.

If you select the wrong state to begin the transition from, use Escape to cancel the transition. Also remember that you can use the RMB while drawing transitions to add waypoints to the line. Used in conjunction with the shift key, lines will be either horizontal or vertical.

c. You can also the following transitions to the State Chart for the normal thread of behaviour:

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>Waiting For Customer</td>
<td>Waiting For Pump Enable</td>
</tr>
<tr>
<td>Waiting For Pump Enable</td>
<td>Ready To Pump</td>
</tr>
<tr>
<td>Ready To Pump</td>
<td>Pumping</td>
</tr>
<tr>
<td>Pumping</td>
<td>Pumping Paused</td>
</tr>
<tr>
<td>Pumping Paused</td>
<td>Fuel Delivery Complete</td>
</tr>
<tr>
<td>Fuel Delivery Complete</td>
<td>Waiting For Customer</td>
</tr>
</tbody>
</table>

These transitions between states specify the normal behaviour of the pump, however, the customer may:

- Replace the gun to the holster before the pump is enabled
- Replace the gun after the pump is enabled and before any fuel has been delivered.
- Deliver fuel, stop and then deliver more fuel.

These extra cases can be added as transitions to the state diagram.
d. Add these extra transitions to the diagram as follows

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>Waiting For Pump Enable</td>
<td>Waiting For Customer</td>
</tr>
<tr>
<td>Ready To Pump</td>
<td>Fuel Delivery Complete</td>
</tr>
<tr>
<td>Pumping Paused</td>
<td>Pumping</td>
</tr>
</tbody>
</table>

Also, while the pump is in the Waiting Pump Enable state, the Tank level must be checked.

e. Add a new state called Fuel Unavailable in the centre of the diagram with the commented Action in the following table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fuel Unavailable</td>
<td># Inform customer that pump is unavailable.</td>
</tr>
<tr>
<td></td>
<td># Wait for fuel to become available for this pump.</td>
</tr>
</tbody>
</table>

f. Add transitions from the Waiting For Pump Enable state and to the Waiting For Customer state.

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>Waiting for Pump Enable</td>
<td>Fuel Unavailable</td>
</tr>
<tr>
<td>Fuel Unavailable</td>
<td>Waiting for Customer</td>
</tr>
</tbody>
</table>

Attach a Signal

A signal represents an incident that causes a change between states.

The Transmissions you have already defined on the Class Collaboration Diagram are available for you to use in the State model but some will need to be added as ‘New Signals’. If the signal appears on the pick-list, select it otherwise add it as a ‘New Signal’.

a. From the Transmission Waiting For Customer -> Waiting Pump Enable, RMB pop-up Modify > Attach Signal.

b. Select the Signal name Gun Removed from the pick-list and click Enter.

c. Add some or all of the other Signals from the following table, selecting New Signal from the picklist if the Signal does not already exist.

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Signal Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Waiting For Pump Enable</td>
<td>Waiting For Customer</td>
<td>Gun Replaced</td>
</tr>
<tr>
<td>Waiting For Pump Enable</td>
<td>Ready To Pump</td>
<td>Pump Enabled</td>
</tr>
<tr>
<td>Ready To Pump</td>
<td>Pumping</td>
<td>Trigger Depressed</td>
</tr>
<tr>
<td>Pumping</td>
<td>Pumping Paused</td>
<td>Trigger Released</td>
</tr>
<tr>
<td>From</td>
<td>To</td>
<td>Signal Name</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>Pumping Paused</td>
<td>Pumping</td>
<td>Trigger Depressed</td>
</tr>
<tr>
<td>Pumping Paused</td>
<td>Fuel Deliver Complete</td>
<td>Gun Replaced</td>
</tr>
<tr>
<td>Fuel Deliver Complete</td>
<td>Waiting For Customer</td>
<td>Customer Finished</td>
</tr>
<tr>
<td>Ready To Pump</td>
<td>Fuel Deliver Complete</td>
<td>Gun Replaced</td>
</tr>
<tr>
<td>Waiting For Pump</td>
<td>Fuel Unavailable</td>
<td>Fuel Level Low</td>
</tr>
<tr>
<td>Enable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fuel Unavailable</td>
<td>Waiting For Customer</td>
<td>Fuel Available</td>
</tr>
</tbody>
</table>
Building a System

*How to include all the necessary components to build an executable model for simulation.*

**Overview of Domain and Project Management**

Real-world domains are typically built as a series of incremental releases, where each subsequent release of a domain provides extended functionality over the previous one.

Additionally, software systems need to be constructed such that they can be supplied as a number of variants of a base product - the components of each variant thus need to be defined and managed.

iUML recognises this by providing support for version management of both domains and projects (for the purposes of this tutorial you can think of a project version as being an integrated set of domains, each at a defined version).

In order to understand the principles of the version management facilities provided by iUML, consider a typical situation where during the domain partitioning phase a requirement for a new domain was identified. Accordingly, you would create a new domain within the iUML database which will automatically be assigned a version number of 1 - you have already done this when you created the Petrol Station Control domain. The domain is then analysed and represented as an xUML model using all the techniques you have learnt during previous sections of this tutorial.

At some point the domain will reach a level of maturity at which you decide that it needs to be baselined so that it may be returned to any time in the future.

Alternatively, you might have reached a point where you would like to explore two or more concurrent development threads based upon the work you have done so far.

In either case, what you want to do is 'freeze' the current state of version 1, and to derive one or more new 'child' versions from it (each child version will initially be a simple copy of its parent version). Once a version has been frozen you will no longer be able to modify it - all future development work on the domain will be done on one or more of its child versions.
You can create a complex ancestry of Domain versions in this way as illustrated in the following diagram.

Version management of Projects may be achieved in a similar way.

Later on in this section of the tutorial you will baseline (freeze) your Initial Version of the Petrol Station Control domain, and then instead of completing the work on the domain by modifying the child version you will import the completed version (which we have provided), as a child of your Initial Version.

In this tutorial, you are not expected to learn ASL but you will add comments for the petrol station Initialisation Segment and for some of the Test Method Segments.

An Initialisation Segment is used to define the initial conditions of an xUML model e.g. the initial object population, the initial status of the signal queues etc.

An External is used to define how the xUML model may be stimulated at simulation runtime.

Prior to explaining how to build an executable model it is worth reviewing how iUML uses its Domain and Project management facilities to support the development of real systems.
Initialisation Segments

Initialisation Segments contain ASL statements which set up the initial conditions of the model to be simulated. Initialisation segments need to contain the necessary initialisations of a domain to build the executable model with the appropriate starting conditions.

A domain version may contain many initialisation sequences. Each initialisation sequence is essentially a list of initialisation segments defined in the order in which they are to be executed during the initialisation phase of the simulation run. One of the initialisation sequences must be made the current selection, iUML will use this current selection when building the model.

For example, in the petrol station, one of the initialisation segments will create a fuel tank (1002) with 200000 litres of fuel and another will link it to pumps 2&3.

1. Creating an Initialisation Segment

In this exercise you will load a previously written ASL Initialisation Segment. As part of the “Simulating the Model with iUML Simulator” section, the Initialisation Segment will be explained as you step through it.

   a. Expand the tree to Domains > Initial Version > Model Execution Environment.
   b. From Initialisation Segments RMB pull-down Add > Initialisation Segment.
   c. Enter the initialisation segment name Prototype Petrol Station and click OK.
   d. Click Prototype Petrol Station to display its details.
   e. From <Initialisation Segment Method> in the details frame, RMB pull-down Load.
   f. Navigate to the <iUML release directory>/Manual/2.01/Tutorial_Files/ASL_code and select proto_init.txt and click OK. The text will appear in the details frame.

2. Creating an Initialisation Sequence

In order to be part of the built model you must create an Initialisation Sequence to contain the Initialisation Segment:

   a. Expand the tree to Domains > Initial Version > Model Execution Environment.
   b. From Initialisation Segments RMB pull-down Add > Initialisation Sequence.
   c. Enter the name Petrol Station Release 1 in the dialog and click OK.

3. Making the Initialisation Sequence current selection

   From Petrol Station Release 1 pop-up Modify > Set as Current Selection.
   Petrol Station Release 1 will be shown as the current selection.

4. Adding an Initialisation Segment to Initialisation Sequence

   a. From Petrol Station Release 1, RMB pull-down Modify > Link a Segment to this Sequence...
b. Select **Segment number 1 - Prototype Petrol Station** from the menu.

The Petrol Station Release 1 will now contain the Initialisation Segment Prototype Petrol Station.

On a real project you would typically develop a number of Initialisation Segments and use subsets of them in a number of Initialisation Sequences.
Test Methods

Test Methods are also written in ASL. They contain statements which initiate threads in the simulated model or simulate replies when a thread leaves the model.

1. Creating a Test Method
   a. Expand the tree to Domains > Initial Version > Model Execution Environment.
   b. From Test Methods pop-up Add Test Method.
   c. Enter the name Customer removes gun from pump 2 and click OK.
   d. Click Customer removes gun from pump 2 to show the details frame
   e. From <Test Method> RMB pop-up Load.
   f. Navigate to the <iUML release directory>/Manual/2.01/Tutorial_Files/ASL_code directory and select rem_pump.txt and click OK.
   g. The ASL Test Method will be loaded with the Test Method, take a brief look at it, then click OK.

2. Creating a Test Method Set
   a. From Test Method Sets pop-up Add Test Method Set.
   b. Enter the name Test Methods For Testing Petrol Station Control and click OK.

3. Selecting the Test Method Set
   a. Expand the tree to Domains > Initial Version > Model Execution Environment.
   b. From Test Methods For Testing Petrol Station Control RMB pop-up Modify > Set as Current Selection.

4. Adding a Test Method to a Test Method Set
   a. RMB pop-up Test Methods For Testing Petrol Station > Link a Test Method to this Set...
   b. Select Test Method number - Customer removes gun from pump 2.

On a real project you would typically develop a number of Test Methods and use subsets of them in a number of Test Method Sets.
In this tutorial, you have built various parts of the Petrol Station Control system model but in its current form it is incomplete. Completion of the model would largely involve repeating what you have learnt so far for the other Classes, Associations, State Machines, etc.

To save you this effort, and to demonstrate the Import/Export facilities of iUML we have provided a completed model as an iUML exported domain version. The following paragraphs provide a step by step guide to re-importing this model which we will subsequently simulate using iUML Simulator.

Note: If you are using iUMLite, you will find that the import and export facilities of iUML are disabled. To continue with the tutorial, simply open the following database:

<iUML release directory>/Manual/2.01/Tutorial_Files/PSC.uml

and skip this section.

### 1. Importing the Domain

The completed model will be imported as a ‘child’ of the version that you have created during the tutorial. Such Parent/Child associations between different versions of the same domain can only exist if the parent has been frozen. Frozen versions can no longer be edited and therefore provide a stable baseline from which child versions can be associated. The step by step guidelines include freezing the Initial Version followed by the import of the provided version as a child.

#### Temporarily Freeze Initial Version of Petrol Station Control Domain

To temporarily freeze Initial Version:

a. Expand the Explorer to Domains > Petrol Station Control.
b. From Initial Version, RMB pop-up Acquire Lock > Top Level Lock.
c. From Initial Version, RMB pop-up Modify > Temporarily Freeze.
d. Confirm by clicking OK. You should note that Initial Version is now annotated with (TF) alongside its Status field.

#### Import Domain as a Child

a. From Initial Version, RMB pop-up Modify > Import Child Version from File....
b. Navigate to <iUML release directory>/Manual/2.01/Tutorial_Files, then select PSC_4_export and click OK.
c. Click Renumber in the Confirm Dialog.
d. Replace the name Initial Version with Second Version in the xUML-Version Details dialogue and click OK.

You should note that Initial Version is now annotated with (TFC)-Temporarily Frozen with Child.
2. Exploring the Imported Version

Before starting to prepare the Domain for simulation you might want to explore the imported version (or, in the case of iUMLite, explore the PSC.uml database).
Domain Simulation Preparation

Simulation of an xUML domain is achieved by translating the model into code, compiling it and then executing it within a simulation environment. This enables its behaviour to be verified. From the diagram you can see that this is a three stage process:

- **writing** the domain to the build area (an analyst-specified directory where the code generation stage takes place);
- **building** the simulation executable (this stage actually involves translating the model into code and then compiling and linking it to generate the executable);
- **executing** the model using iUML Simulator which provides a controlled environment in which to verify xUML models. The Simulator features include the ability to set breakpoints, asynchronously stimulate the model, and browse the object population at run-time.

Each of these stages is invoked separately from a menu option from within the iUML tool - these are described in the following sections.

1. **Defining the Build Area**

   As described in the previous section the Build Area is simply a directory which has been designated as the location where the code generation stage takes place.

   During the ‘Write To Build Area’ stage a domain version is extracted from the iUML database and written to a sub-directory of the Build Area - the name of this sub-directory is automatically assigned by iUML as a concatenation of the
domain key letter and its version number (for example the key letter of the Petrol Station Control domain is ‘PS’, so version 1 of this domain will be written to a sub-directory named ‘PS_1’.

If you were to inspect the contents of this sub-directory you would find a number of different files which are used by the code generator to build the simulation executable - it is not within the scope of this tutorial to define the detailed contents of each of the files.

a. Expand the tree to Domains > Petrol Station Control
b. Click Petrol Station Control to display the Domain details.
c. Double click the <Domain Build Area> text.
d. Select a location for the Build Area and Click OK.

2. Building the Simulation Executable

The following two paragraphs step you through the first two stages of the build and simulation outlined in the diagram above, namely:

• writing the domain version to the build area, and
• building the domain version (i.e. generating the code, and then compiling and linking it to create the simulation executable).

The final stage in this process is explained in the next section of the tutorial.

Write Domain to Build Area

b. Check the path is correct and click Yes.
c. When this stage is completed, you will be prompted with an Acknowledge dialog. Click OK.

Build the Domain Version

b. Check the path and Click Yes.

The Code Generator now starts up and presents its progress an a new window labelled ‘Domain Build: PSC’.

This stage will take a few minutes while the code generator translates the xUML models into code, compiles it and creates links to construct the simulation executable.
When the code generator has finished, the dialog shows the line Build Complete.

To clear this window from the desktop: Click inside it and press Return.
Simulating the Model with iUML Simulator

Using iUML Simulator to test the analysis model built with iUML

In this section you see how the model created in iUML can be verified through model execution. This allows detection of any errors very early in the development of your system.

The iUML Simulator provides an interface for you to:

• control the simulation
• view the ASL which is being executed
• view and navigate the runtime object data
• simulate single and multi domain applications

Aims of this section:

• To introduce you to iUML Simulator
• To show you how the iUML Simulator executes iUML models.
The iUML Simulator tool

iUML Simulator provides two main windows to enable you to interact with your executable model. Object data are shown using display tables.

Start iUML Simulator from iUML

b. Check that the path is correct and click OK.
c. Wait while iUML invokes iUML Simulator

The iUML Simulator Window

The buttons on the Control window control the execution of the application. At The Lower frame of the window is a status and error display; any errors or information generated by the application are shown here.
1. Populating the model with initial objects

The model is populated by executing the Initialisation Segments defined within the selected Initialisation Sequence.

In the case study, the petrol station supplies two grades of fuel “Four Star” & “Unleaded” which are stored separately in two storage tanks. “Tank 1001” contains “Four Star” and “Tank 1002” contains “Unleaded”. The station has three fuel pumps, pump 1 supplies “Four Star”, pumps 2 & 3 supply “Unleaded”.

In the Petrol Station Model, the Initialisation Segment will therefore do the following:

- create two objects of FUEL GRADE class
- creates two objects of TANK class
- ‘links’ the TANK classes to the FUEL GRADE classes
- creates three objects of PUMP class
- links the PUMP classes to the TANK classes.

**Show the Object Tables**

Prior to executing the Initialisation Segment there will be no objects within the executing model. You can prove this by viewing the Object Tables

a. Pull-down Data > Instances... and select FUEL_GRADE. This will open an object

---

**iUML Simulator Functions**

The Simulator commands are in four groups, Execution, Show, Breakpoints and Stimulus.

**Execution:**

- **Continue** executes the xUML model until there are no queued signals to process.
- **Step Signal** executes any remaining synchronous behaviour, de-queues the next signal (if there is one) and stops at the beginning of the respective state action.
- **Step Over** executes the next line of ASL. If the ASL statement involves calling an Operation, iUML Simulator will execute the operation but not show it being executed line by line. i.e. you ‘Step Over’ the operation.
- **Step In** executes the next line of ASL and then stops. If the next line is an Operation call, you will ‘Step In’ to the operation and you can then step through the operation line by line.

**Show:**

- **Instances** shows the Object Tables for a Class.
- **Assigners** shows the current states of all Assigner State models
- **Breakpoints** lists all currently set Breakpoints. (Except ASL and Signal breakpoints which are shown in the ASL Display window.
- **Signals** shows the current state of the Signal Queue.
- **Trace** displays the History of all Signals consumed so far by the xUML model.
- **Local Variables** shows the variables in the current ASL.

**Stimulus**

**Timer** is used in multi-domain simulation to list all the currently defined xUML Timers.

**Test Method** allows you to asynchronously stimulate the model by executing a specified Test Method.
table for the **FuelGrade** class.

An Object table is a tabular representation of the objects of the respective Class - each row in the table represents an object (object) of the class.

b. Repeat to open tables for;
   PUMP and TANK. Arrange them so they all are visible on your screen.

c. Click **Cancel** on the Instances window.

**Step lines of ASL**

In this exercise you will step through each line of ASL in the Initialisation Segment. You will notice that class and association objects are created and that class attributes are set as you step through and execute the statements.

a. Click the **Step Over** button in the top of the window. See that the Current Line marker is on line # 14.

This ASL statement creates an object of the class FUEL_GRADE and returns an object handle which is a reference to the newly created FUEL_GRADE object. This reference is the local variable `four_star_fuel_grade`.

The FUEL_GRADE class has a number of attributes which need to be initialised. The statement defines the identifying attribute, **Grade_Name** as Four star and **Unit_Price** as 62.9 (pence per litre).

b. Click **Step Over** to execute the this ASL Statement and notice that you are now at line #17.

c. Look at the FUEL_GRADE class object table and notice that a row has appeared. This represents a new object of the class FUEL GRADE which has been created as a result of executing the ASL statement at line #14.

Each column of an object table represents either:

- An attribute of a class.
- An association that the class is involved in.

Attributes can be checked in iUML Simulator by looking at objects and checking that the attributes are what is expected.

Look at the association cells and check that the related number of objects are correct.

The details of the FUEL GRADE class are displayed in the FUEL GRADE Object Table.

d. Click **Step Over** to execute the ASL statement at line #18.

e. Look at the FUEL GRADE object table and notice that another row has appeared.

Another object of the class FUEL GRADE has now been created.
f. Click Step Over twice to execute the ASL Statements which start on Lines #23 and #31. The Line Marker will then point to line #38 (the next line to be executed).

g. Look at the TANK object table and notice that two rows have appeared.

h. Click Step Over.

i. Look at the TANK object table again and notice that another object of the association R2 has been created between the Four Star FUEL GRADE object and Tank 1001. (You will need to increase the width of the dialogue or scroll the table horizontally to see the R2 Fuel Grade column).

j. You can verify this by clicking on the R2 cell for the object of FUEL GRADE. The tool will display a new window containing a table of the objects of TANK that are related to the Four Star Fuel Grade object via R2.

k. Dismiss the new window by clicking Cancel.

l. Click Step Over.

m. Look at the TANK object table and notice that another object of the association R2 has been created between the Unleaded Fuel grade object and Tank 1002.

n. Click Step Over 6 times

Execution of this Scenario has now created and linked three objects of the Pump class. Pump 1 is linked to Tank 1001 containing Four Star fuel and Pumps 2 and 3 are linked to Tank 1002 containing Unleaded fuel. The Line Indicator will now be pointing at line #61.

o. Look at the PUMP object table and notice that three rows have been created.

2. Verifying the model

The primary purpose of the iUML Simulator is allow you to debug the analysis models. By viewing and navigating the object data shown in the tables, you can compare the actual behaviour of your model with the expected behaviour.

Show Local Variables

For the ASL segment that is displayed in the ASL Display Window, there can be a number of local variables. You can view these variables during execution.

a. Pull-down Data > Local Variables.

b. Move the Show Local Variables away from the Control Window and ASL Display Window.
c. See that the dialogue displays all the local variables for the scenario along with their types and current value

```
<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>four_inter_fuel_grade</td>
<td>Instance</td>
<td>DEFINED</td>
</tr>
<tr>
<td>unleaded_fuel_grade</td>
<td>Instance</td>
<td>DEFINED</td>
</tr>
<tr>
<td>tank_1001</td>
<td>Instance</td>
<td>DEFINED</td>
</tr>
<tr>
<td>tank_1002</td>
<td>Instance</td>
<td>DEFINED</td>
</tr>
<tr>
<td>pump_1</td>
<td>Instance</td>
<td>DEFINED</td>
</tr>
<tr>
<td>pump_2</td>
<td>Instance</td>
<td>DEFINED</td>
</tr>
<tr>
<td>pump_3</td>
<td>Instance</td>
<td>DEFINED</td>
</tr>
</tbody>
</table>
```

d. Click Step Over once more - the Scenario is now completed, the ASL Display window will be blank.

Navigate Object Populations

You have now populated the application with the initial configuration of Class Objects. iUML Simulator can now be used to check that the objects are correct. This involves viewing the Class objects, comparing the attribute values and ensuring that the correct association links have been performed.

3. Simulating behaviour using Test Methods

In this exercise you will use Test Methods to simulate the behaviour external to the Petrol Station Control domain. The set of Test Methods in the list simulate the steps a customer goes through when ‘filling up’ with unleaded fuel from Pump2 and paying for the delivery.

a. Pull-down Data > Classes and open the DELIVERY and TRANSACTION classes (PUMP, TANK and FUEL GRADE should already be open).

b. Arrange them on your screen so all dialogues are visible. Drag the Class object dialogues as wide as you can so you can see most of the columns.

Show Signal Queue

The Signal Queue lists the signals that remain to be processed.

You can monitor the status of Signals in the Event (Signal) Queue by showing the Event Queue dialogue.

Pull-down Data > Signals... to display the dialogue.

Invoke a Test Method


b. Select CUSTOMER Removes Gun From Pump 2 and click OK. The Test Method's ASL is now shown in the Display Window.
Step Over the ASL in the Externals

The **Step Over** button steps each line of ASL in the ASL window.

- a. Click **Step Over** four times.
- b. See that **Pump 2** (PUMP Object Dialogue) has changed from state **Waiting For Customer** to **Waiting_Pump_Enable**.
- c. Repeat clicking **Step Over** until the Current Line Marker is pointing to line #8.

![ASL Code Window](image)

This ASL Statement is an operation invocation (i.e. Bridge Invocation) which in the final system would be mapped to an operation provided by the Attendant Interface Domain.

- d. Step into this Bridge Invocation by clicking **Step In**.

As only a single Domain is being simulated, the Attendant Interface is not available for execution. You should note that the ASL Display window contains ASL to stub the bridge, which in this case is two comment lines.

- e. Click **Step In** twice more to return to the ASL Statement immediately following the Bridge Invocation Statement.
- f. Click **Step Over** twice and notice that the ASL Code Window goes blank.
- g. Click **External**, select **ATTENDANT Enables Pump 2** and click **OK**.
- h. **Step Over** the ASL until the Signal Queue dialogue shows the **Create_Delivery** signal.

![Signal Queue Dialogue](image)

- i. Click **Step Over**. The **Create_Delivery** signal has now been consumed and the Signal Queue dialogue shows **No Data Held**.
- j. **Step Over** again until an object of **DELIVERY** is created (shown on the **DELIVERY** Object table). You will see that the time you invoked the delivery using the Test Method is shown in the first column of the data table is the current time.
- k. **Step Over** until the ASL window is clear.

Run the ASL using Continue

You can use **Continue** to execute an entire section of ASL in the Simulator.

- a. Invoke the **CUSTOMER Presses Trigger At Pump 2** Test Method.
- b. Click **Continue** to run the **CUSTOMER Presses Trigger At Pump 2** Test Method ASL
The iUML Simulator tool

until it is complete.

c. Use Continue to run the METER Delivers Fuel Unit For Pump 2 a number of times and to execute the CUSTOMER Releases Trigger At Pump 2 Test Method.

Deletion and Creation of Class Objects

During the execution of the model, classes and associations are created and deleted. You can use iUML Simulator to check that deletion and creation occurs in the correct order.

a. Invoke the CUSTOMER Replaces Gun At Pump 2 Test Method.

b. Click Step Over until the state of PUMP 2 changes to Fuel_Delivery_Complete.

c. Click Step Over until you reach Class: DELIVERY State: 6 Line 10.

d. Look at the DELIVERY Class Object table (R3 PUMP row) and click Step Over you will notice that the DELIVERY object is unlinked.

e. Click Step Over again and see that the DELIVERY object is deleted.

f. Click Continue to complete the actions of the Test Method.

Completing the Scenario

Finish this scenario by invoking and Executing the CUSTOMER Pays For Fuel For Pump 2.

To show the Signal Trace history pull-down Data > Signals Trace ... .
Ending Your iUML Simulator session

To exit iUML Simulator, pull-down File > Exit and confirm. Exit the Command tool window used for displaying by pressing Enter in the window.