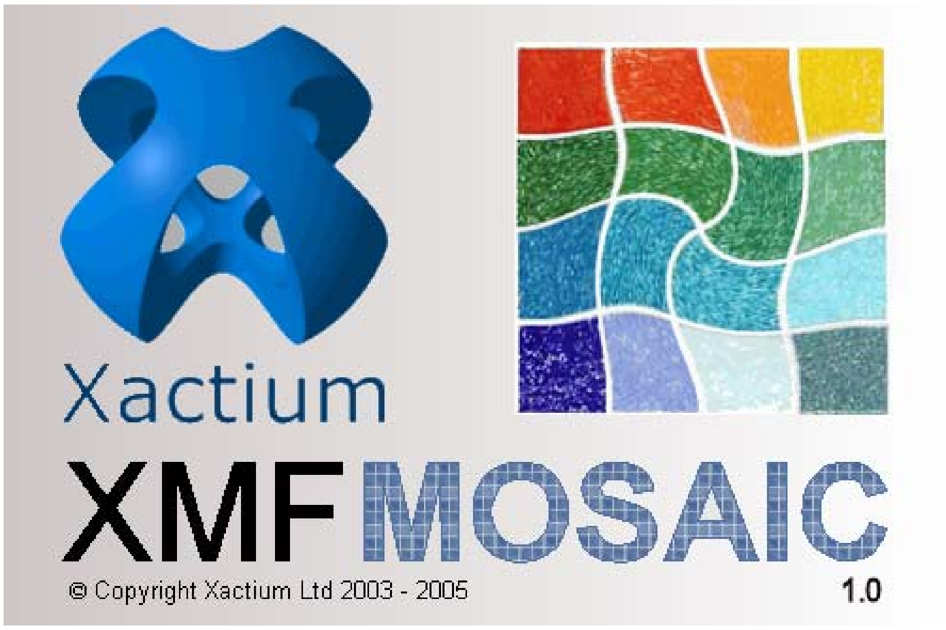


XMF-Mosaic
Getting Started Guide

Version 1.0

July 2005

1. Introduction

This is a short guide to getting started with XMF-Mosaic 1.0 using a simple Hello World example. Note, for a full in-depth introduction to the many different capabilities of XMF-Mosaic, you should read the BlueBook manual, which can be find via the Help menu in the tool.

2. What is XMF-Mosaic?

XMF-Mosaic is a platform for building tailored tools that provide high level automation, modelling and programming support for specific development processes, languages and application domains.

XMF-Mosaic aims to help significantly improve software productivity by providing a means of automating software development practices which might be traditionally performed using manual processes.

As an example, a tool could be designed in XMF-Mosaic that supports the rapid construction of databases and associated front end web interfaces. It would provide: a domain specific modelling or programming language for capturing the key features of the design (including the design of the interface), facilities for validating the correctness of the design, and support for generating application specific code for both the back and front end applications.

XMF-Mosaic implements many aspects of an MDA tool: providing a MOF editor and transformation engine, however, it goes beyond it, providing many useful and generic facilities for domain modelling.

XMF-Mosaic is completely described in itself – in other words, all aspects of the tool are described in terms of instances of models written in specific languages (even the user interface is described this way using a collection of user interface languages). Because the user has access to these models, it provides a highly open and transparent platform for domain specific tool design.

2.1 Benefits

Using XMF-Mosaic you can:

Have control over the tools you use

- Rapidly create best fit domain specific tools that specifically target your business domain.

Automate your development processes

- Precisely control development processes by capturing them as models, including code generation rules, business rules and validation processes.

Benefit from MDA

- XMF-Mosaic delivers the benefits of MDA, including platform independence, tool interoperability and agile development.

Add value to your assets

- XMF-Mosaic can be easily integrated with your existing tools, models and processes.

Avoid Vendor Lock-in

- Full access to the underlying representation of the tools means that they are fully accessible and can be rapidly ported to other tool platforms if necessary.

2.2 Supported Capabilities in 1.0

XMF-Mosaic 1.0 is instantiated with the following capabilities:

- Visual domain modelling tool.
- Model instantiation using Snapshots.

- User interface modelling capabilities.
- Full support for writing constraints
- Full support for meta-programming in XOCL (eXtensible Object Command Language).
- Built in engine for running XOCL.
- On the fly compilation of code to a platform independent MOF Virtual Machine.
- Model to model mappings using XMap.
- Support for XML grammar definition, parsing and model population.
- Support for grammar definition, parsing and model population.
- Advanced meta-profile capabilities.
- Full pattern matching support embedded within XOCL.
- Standards aware: XML, XMI, Java, Sockets.

2.3 What's Coming

There are many cool features coming in future versions of XMF-Mosaic. Here are just a few:

- More documentation, walkthroughs and wizards (1.1)
- Off the shelf transformation to other languages, e.g. C#, J2EE (1.1-1.2)
- More extensive debugging facilities
- IDE programming support: code completion, etc (2.0)
- More flavours of XMI import and export (1.1)
- More facilities for modelling tool user interfaces (1.1-1.2)
- Deployment of standalone Eclipse tools (1.2-2.0)
- Integration with EMF (2.0)
- Model merge (1.2)
- Copy and Paste (1.2)
- Collaborative working (2.0)
- Model to model synchronisation (1.2)

2.4 Getting Started – Hello World

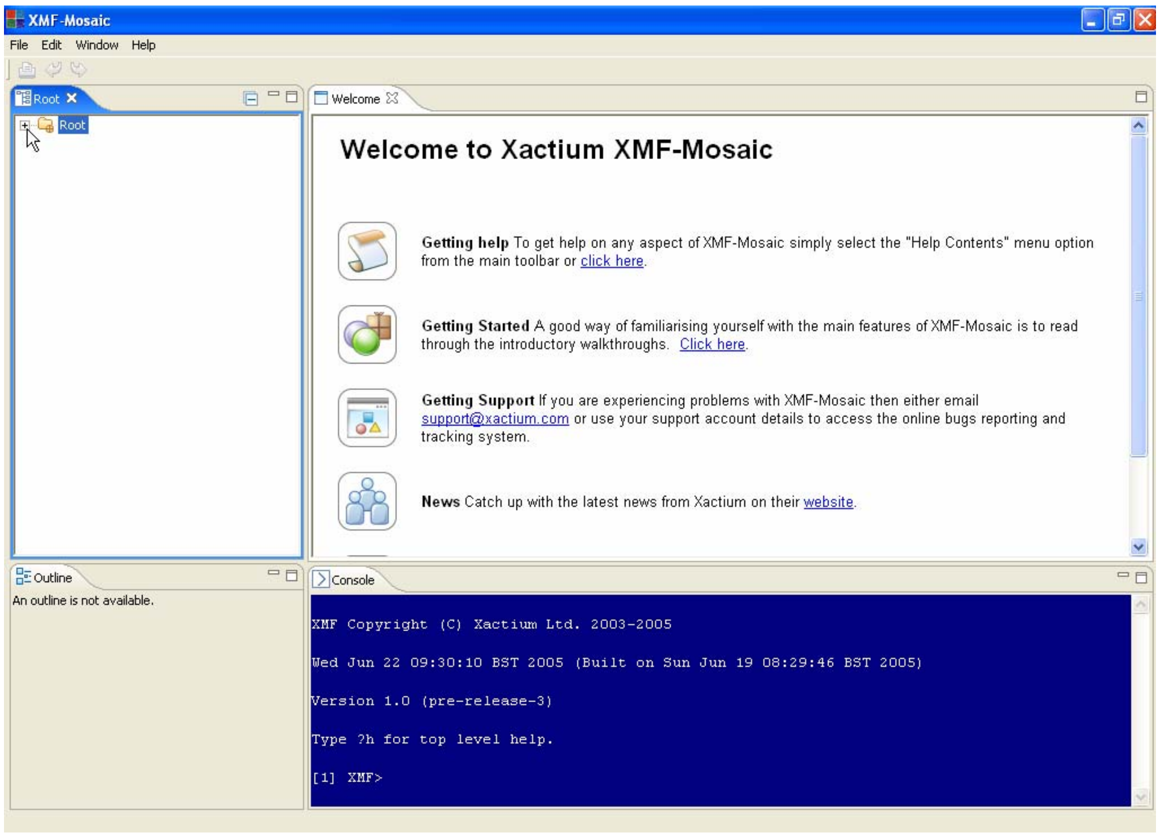
The remainder of this document presents a simple tutorial.

In this tutorial we are going to create a very simple domain specific tool - a "Hello World" tool, which can be used to create and test Hello World designs and then generate a wide variety of Hello World solutions, for example a solution that runs on XMF-Mosaic, or one which runs in Java.

The first step is to describe the domain concepts that the tool will manipulate, and then construct two types of editors: a text editor and associated parser, and a diagram editor. After that two simple generators will be defined for the language: one that runs on the meta-programming language provided with XMF-Mosaic, and the other a Java program.

Note, if you have problems or queries, please contact support@xactium.com. Free evaluation copies of XMF-Mosaic are available for download from Xactium's web-site: www.xactium.com.

Let’s start up XMF-Mosaic. Choose the XMF-Mosaic tool icon from wherever you chose to put it when the tool was installed.



Welcome to Xactium XMF-Mosaic

Getting help To get help on any aspect of XMF-Mosaic simply select the "Help Contents" menu option from the main toolbar or click here.

Getting Started A good way of familiarising yourself with the main features of XMF-Mosaic is to read through the introductory walkthroughs. Click here.

Getting Support If you are experiencing problems with XMF-Mosaic then either email support@xactium.com or use your support account details to access the online bugs reporting and tracking system.

News Catch up with the latest news from Xactium on their website.

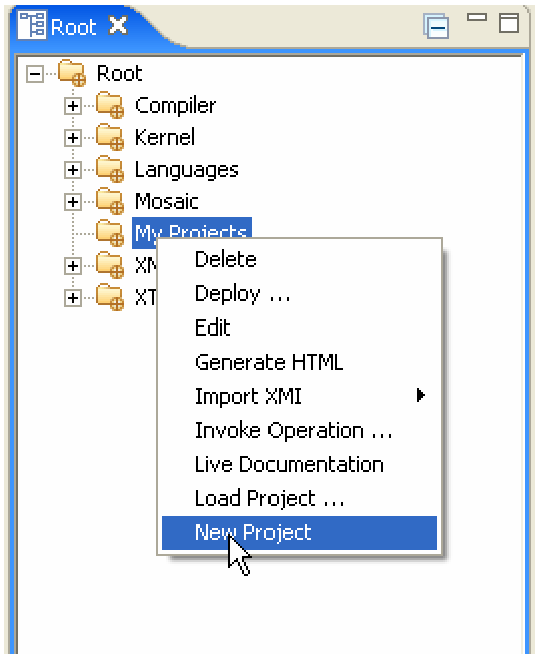
An outline is not available.

```
XMF Copyright (C) Xactium Ltd. 2003-2005

Wed Jun 22 09:30:10 BST 2005 (Built on Sun Jun 19 08:29:46 BST 2005)

Version 1.0 (pre-release-3)

Type ?h for top level help.

[1] XMF>
```

There are four main windows in XMF-Mosaic. The browser (top-left), the main editing pane (top-right), the property editor and console (bottom right) and the outline window (bottom left).

We want to create a new project. To do this open the Root tab in the browser.

Right click on the MyProjects project and select New > Project.



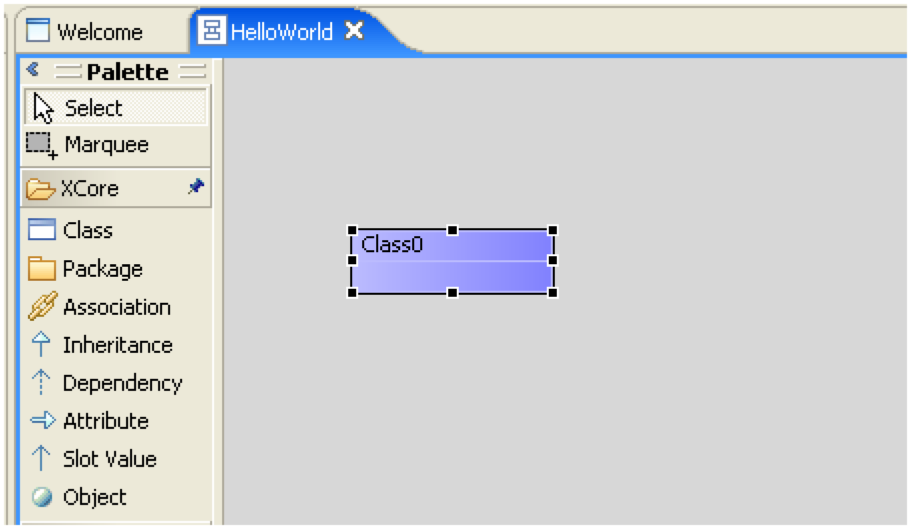
A new project will be created called Project0. Click on the name and change it to HelloWorld.

2.5 Building a Domain Model

The domain of our HelloWorld example is very simple, just one class! However, we will add to this later on.

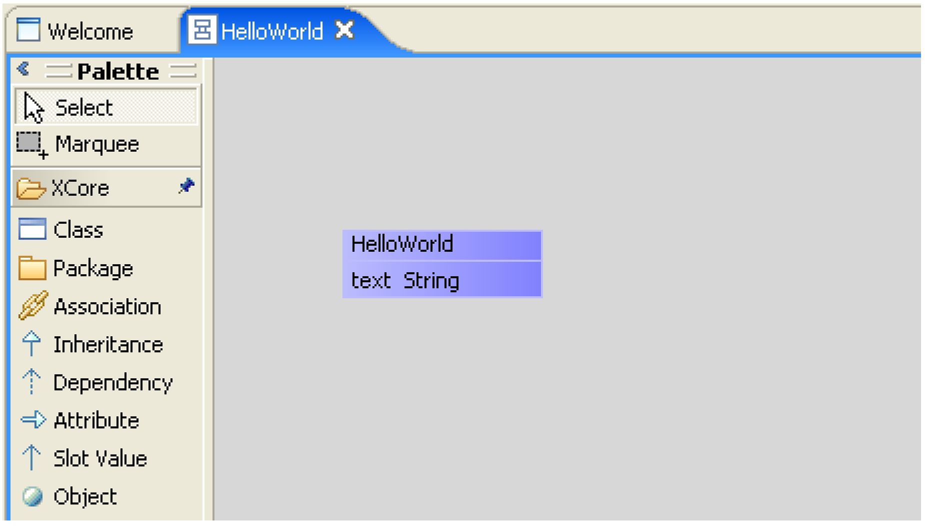
We need to open the diagram editor for the HelloWorld project. Double click on the project and a package will be displayed in the browser. Right click on this and select ShowDiagram.

Create a new concept in the Diagram editor by selecting the Class icon and then clicking on the diagram to create it. In XMF-Mosaic domain concepts are represented as classes – we will use the terms interchangeably from now on.



Click on the name of the new concept and change it to HelloWorld.

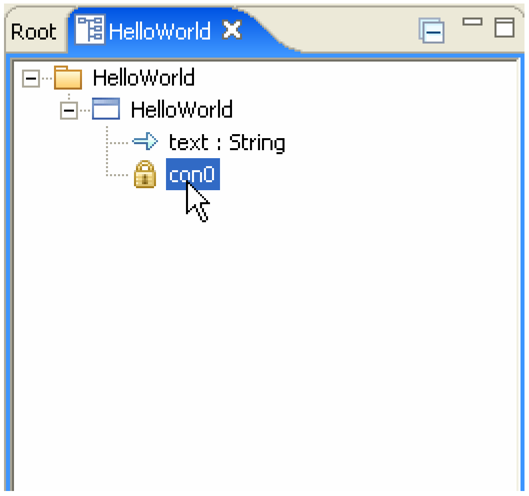
Next we will add an attribute (field) to the concept. Right click on the concept and select New > Attribute > String. Change its name to be "text" by clicking on it. The result should look like this:



2.5.1 *Adding Constraints to the Domain Model*

Now we've defined our concept, however, we might want to place some constraints on it that rule out certain invalid instances of the concept. For example, we might want to ensure say that the value of the text attribute cannot be "GoodBye".

To do this, right click on the concept in the browser and select New > Constraint (this can also be done in the diagram editor). A new constraint will have been added to the concept.

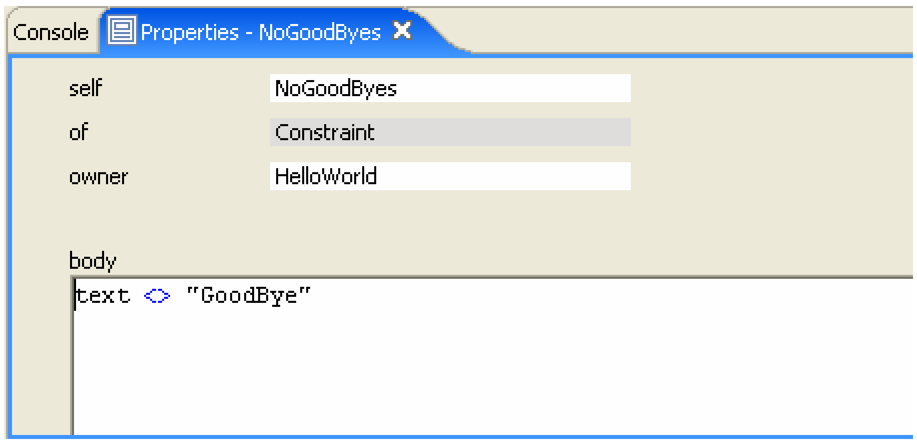


Edit the name of the constraint and call it NoGoodByes.

We are now ready to add a rule to the constraint. Double click on the constraint icon in the browser and a constraint editor will be displayed in the bottom right window of the tool.

Add the rule by editing the text in the body of the constraint. The editor will turn pink until the constraint has been added. To do this, right click in the background of the text editor and select Commit.

The rule to add is as follows:



self	NoGoodByes
of	Constraint
owner	HelloWorld

body

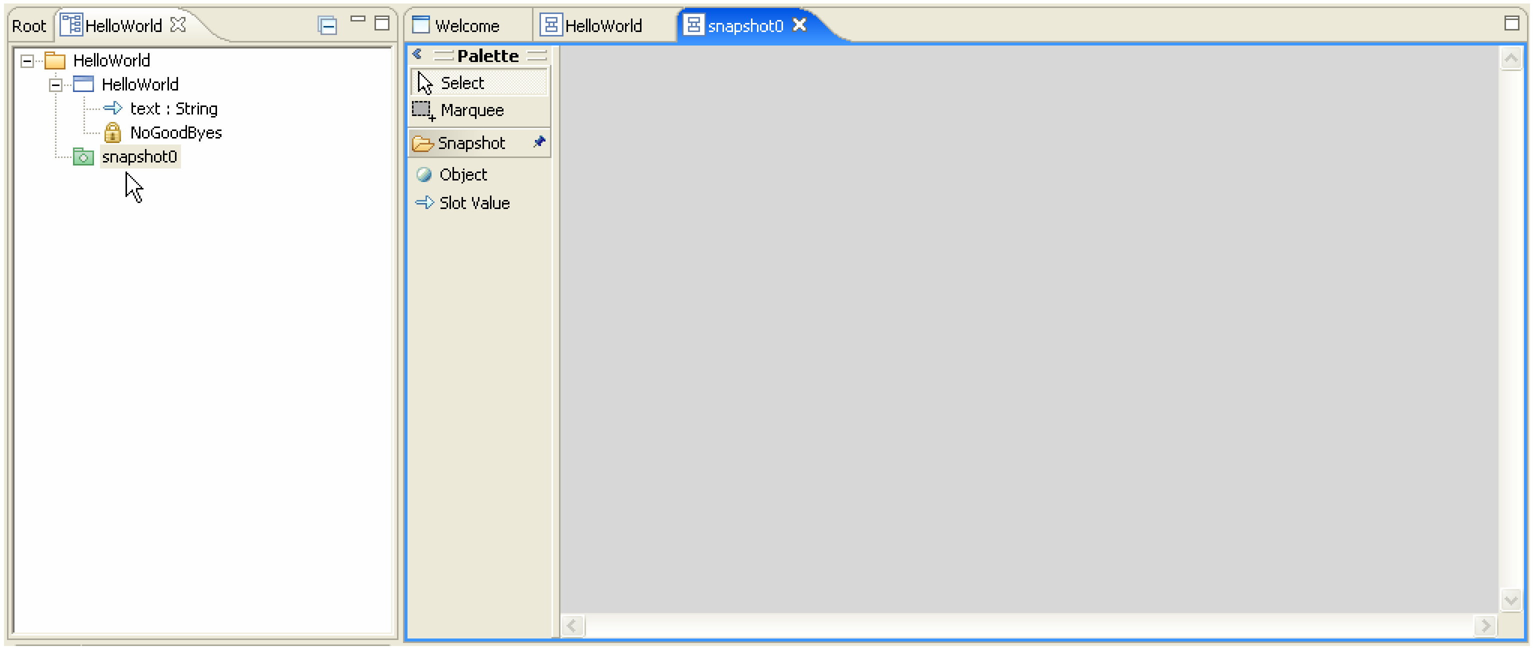
```
text <> "GoodBye"
```

Committing the code parsers and compiles the constraint in XMF-Mosaic VM instructions. The next section describes how we can run the constraint.

2.6 Testing the Domain Model

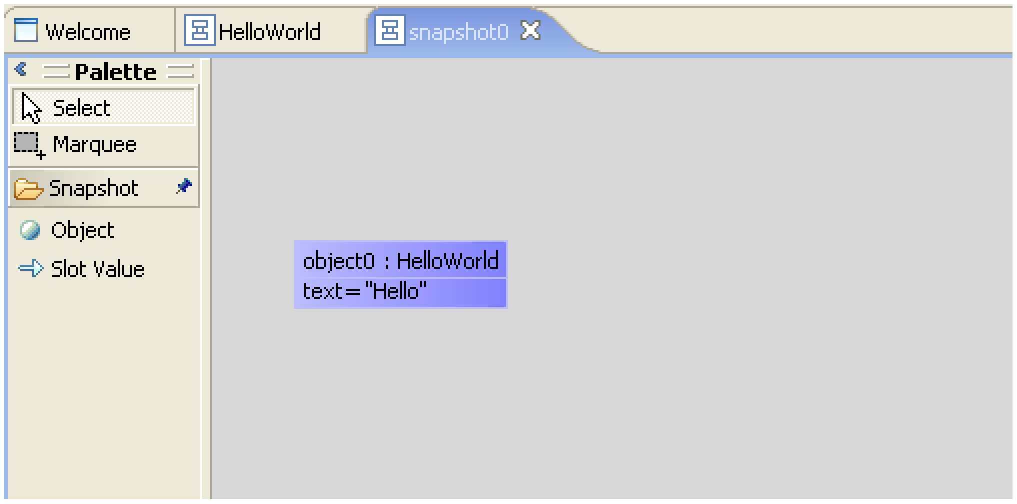
XMF-Mosaic provides a rich array of facilities for testing domain models. The first approach, creating a snapshot, allows you to diagrammatically construct instances of domain concepts. To do this, right click on the HelloWorld package in the browser and select Create Snapshot.

A new snapshot will be added to the browser and a snapshot diagram editor will be displayed.



To create an instance of the HelloWorld concept select the Object icon in the editor and click on the diagram. A menu of classes to instantiation will be displayed. Select the class HelloWorld.

The object will be added to the diagram. The value of the text attribute can be edited by clicking on it. The result should look something like this.

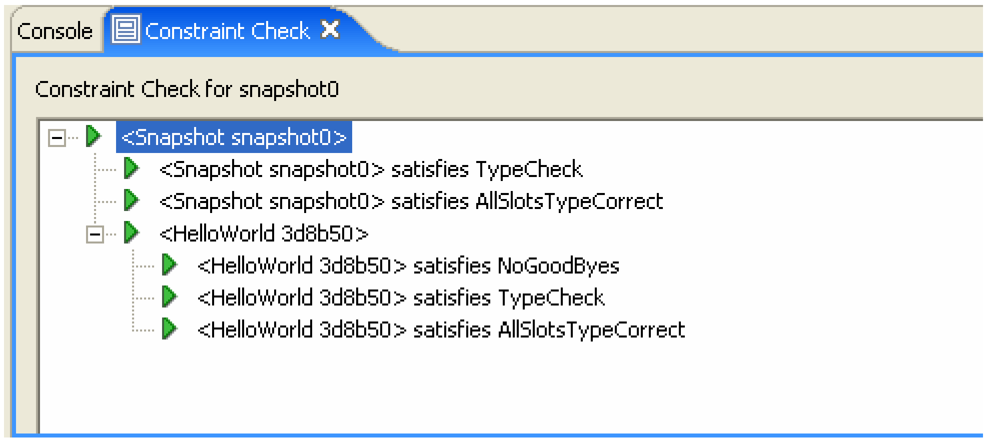


Its properties can be edited by right clicking on the object and selecting Edit.

2.6.1 *Running the Constraint Checker*

Finally, we can check the constraints of all the objects in the snapshot. To do that, right click on the background of the snapshot and select Check Constraints.

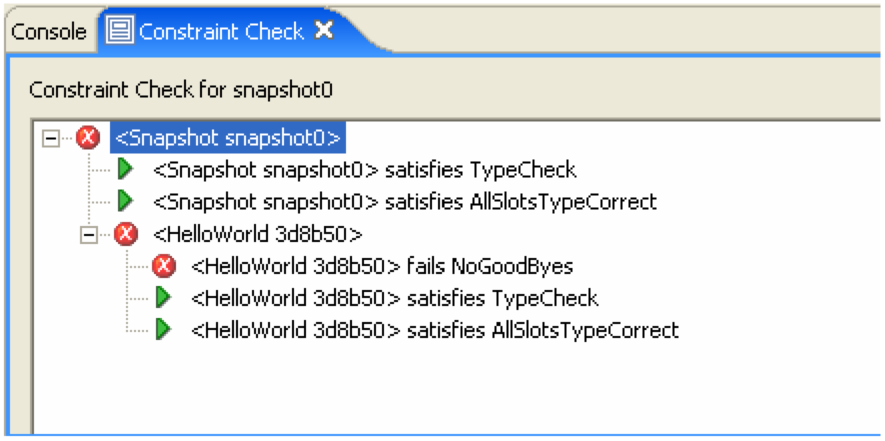
A Constraint Report will be displayed:



```
Constraint Check for snapshot0
<Snapshot snapshot0>
    <Snapshot snapshot0> satisfies TypeCheck
    <Snapshot snapshot0> satisfies AllSlotsTypeCorrect
    <HelloWorld 3d8b50>
        <HelloWorld 3d8b50> satisfies NoGoodByes
        <HelloWorld 3d8b50> satisfies TypeCheck
        <HelloWorld 3d8b50> satisfies AllSlotsTypeCorrect
```

This reports that the HelloWorld object satisfies the constraint NoGoodByes. It also satisfies a couple of type check rules as well.

Try changing the text attribute value to be "GoodBye" and re-running the constraint check. This time the constraint fails. Clicking on the failed constraint takes you to the constraint itself.

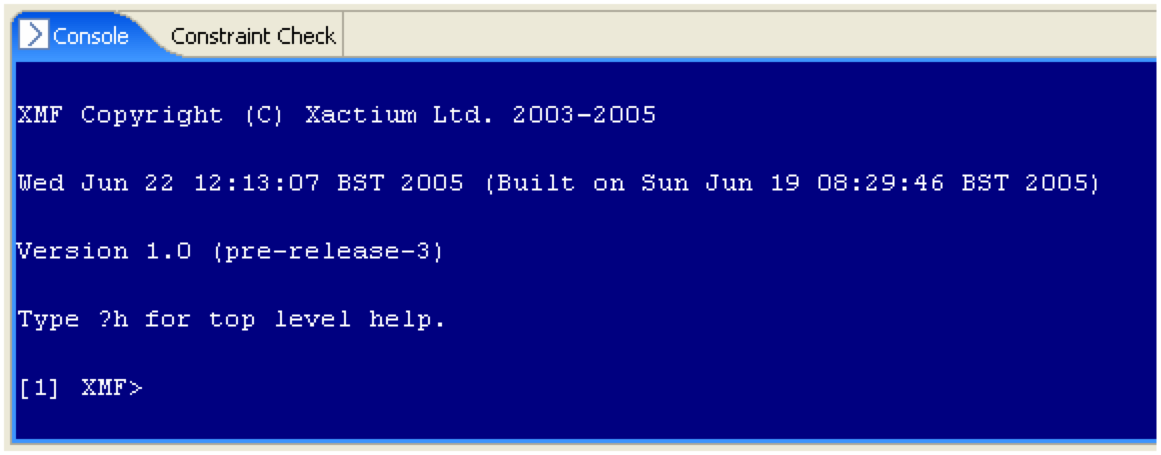


```
Constraint Check for snapshot0
<Snapshot snapshot0>
    <Snapshot snapshot0> satisfies TypeCheck
    <Snapshot snapshot0> satisfies AllSlotsTypeCorrect
    <HelloWorld 3d8b50>
        <HelloWorld 3d8b50> fails NoGoodByes
        <HelloWorld 3d8b50> satisfies TypeCheck
        <HelloWorld 3d8b50> satisfies AllSlotsTypeCorrect
```

2.6.2 *Running the Console Interpreter*

An alternative way of testing the domain model is to interact with it via the Console. The console provides a fully interactive interpreter for accessing the tool (at any level you want). You can run XOCL expressions in the console to navigate models, create instances of models and generally test and debug the tools you build. XOCL stands for eXtensible Object Command Language, and provide a fully feature language for programming in the tool.

To view the console, click on the Console button in the property editor. The console will be displayed:

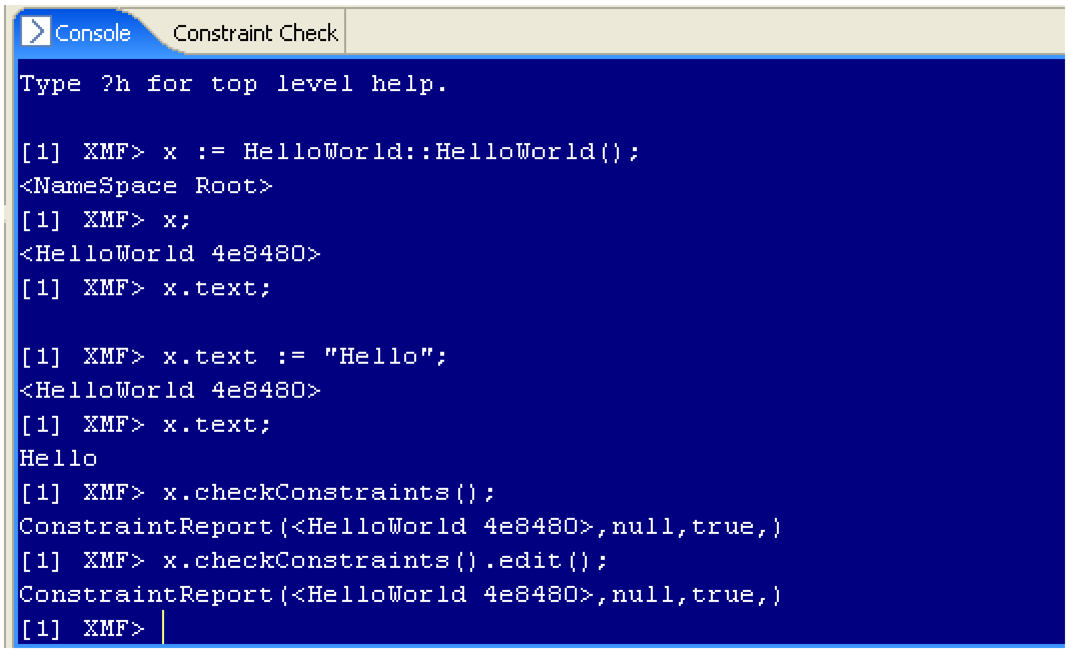


```
XMF Copyright (C) Xactium Ltd. 2003-2005

Wed Jun 22 12:13:07 BST 2005 (Built on Sun Jun 19 08:29:46 BST 2005)

Version 1.0 (pre-release-3)

Type ?h for top level help.

[1] XMF>
```

Using the console requires knowledge of XOCL, but here are some simple things to try:

- Create an instance of the HelloWorld class and assign it to a variable

- Examine the text attribute value of the object
- Assign it a new text value
- Check that it has been updated
- Run its checkConstraints() operation



```
Type ?h for top level help.

[1] XMF> x := HelloWorld::HelloWorld();
<NameSpace Root>
[1] XMF> x;
<HelloWorld 4e8480>
[1] XMF> x.text;

[1] XMF> x.text := "Hello";
<HelloWorld 4e8480>
[1] XMF> x.text;
Hello
[1] XMF> x.checkConstraints();
ConstraintReport(<HelloWorld 4e8480>,null,true,)
[1] XMF> x.checkConstraints().edit();
ConstraintReport(<HelloWorld 4e8480>,null,true,)
[1] XMF>
```

Note that we needed to navigate to the class via its package using a path. We were also able to view the ConstraintReport in the tool by running edit() on it – in general any element can be edited by calling this operation.

2.7 Building a Textual Syntax and Parser

Once we have defined the domain model and tested it to our satisfaction, we can beginning constructing a tool for entering instances of the domain model. In this section, we'll build a textual grammar for the domain model that enables instances to be represented textually and parsed into the tool. As an example, let's build a grammar that can parse HelloWorld programs that look this:

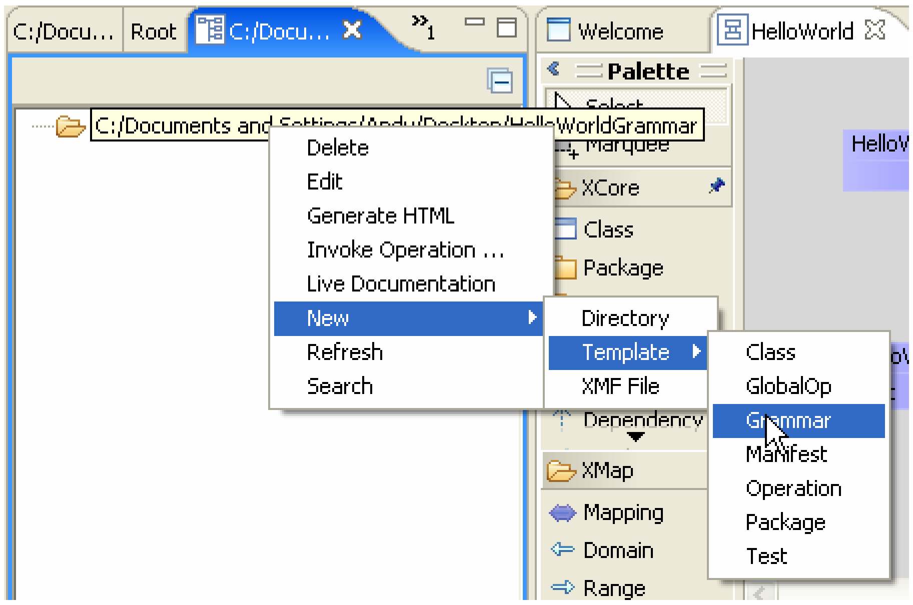
```
@HelloWorldModel
  hello Andy
  hello James
  hello Tony
  hello Sheila
  hello Lili
end;
```

To do this, we will use XMF-Mosaic's powerful grammar definition language, XBNF. Using XBNF you can construct a grammar for a specific language or collection of domain concepts and attach actions to the expressions for synthesising instances of models.

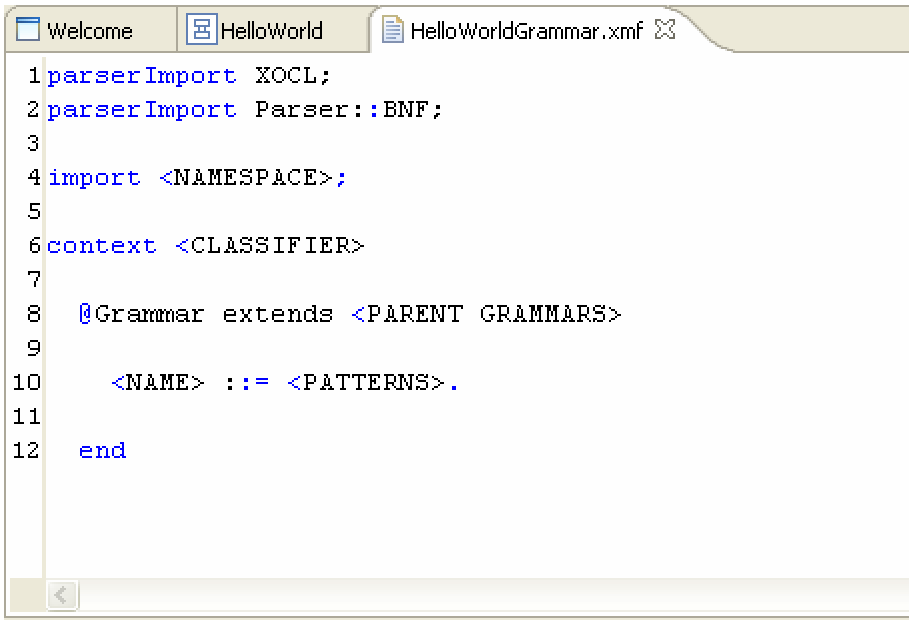
XBNF definitions are entered textually, so a new text file needs to be created.

Go to File > File Browser … and open a file browser in the directory where you wish to save your grammar file (for example Desktop/HelloWorldGrammar/).

Now create a new file. A number of pre-defined templates for creating files are available. Select the Grammar Template:



A new file will be created in the browser. Give it a name, e.g. HelloWorldGrammar and double click on the file to edit it. This provides the following basic structure:



```
parserImport XOCL;
parserImport Parser::BNF;

import <NAMESPACE>;

context <CLASSIFIER>

  @Grammar extends <PARENT GRAMMARS>

    <NAME> ::= <PATTERNS>.

  end
```

The first two lines import the parser for XOCL and the parser for XBNF.

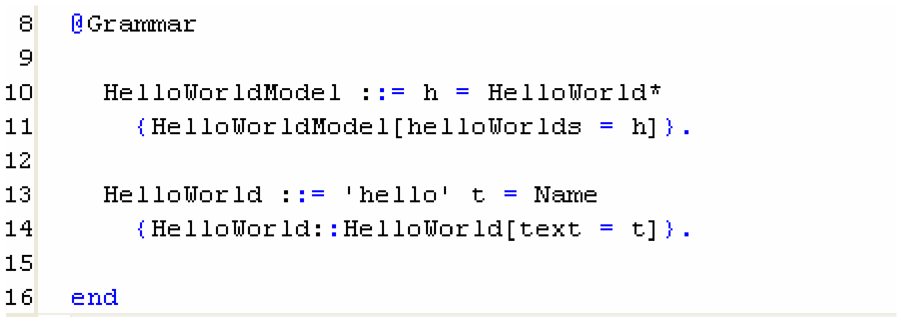
The remainder is changed as follows:

- The HelloWorld package is added to the import list.
- The context of the grammar is changed to the HelloWorldModel class.
- The parent grammars are removed (there are none).

The file now looks like this:

```
parserImport XOCL;
parserImport Parser::BNF;

import HelloWorld;

context HelloWorldModel

  @Grammar

```

Now for the actual detail of the grammar. This is defined as follows:



```
@Grammar

  HelloWorldModel ::= h = HelloWorld*
    {HelloWorldModel[helloWorlds = h]}.

  HelloWorld ::= 'hello' t = Name
    {HelloWorld::HelloWorld[text = t]}.

end
```

The grammar definition consists of two grammar rules.

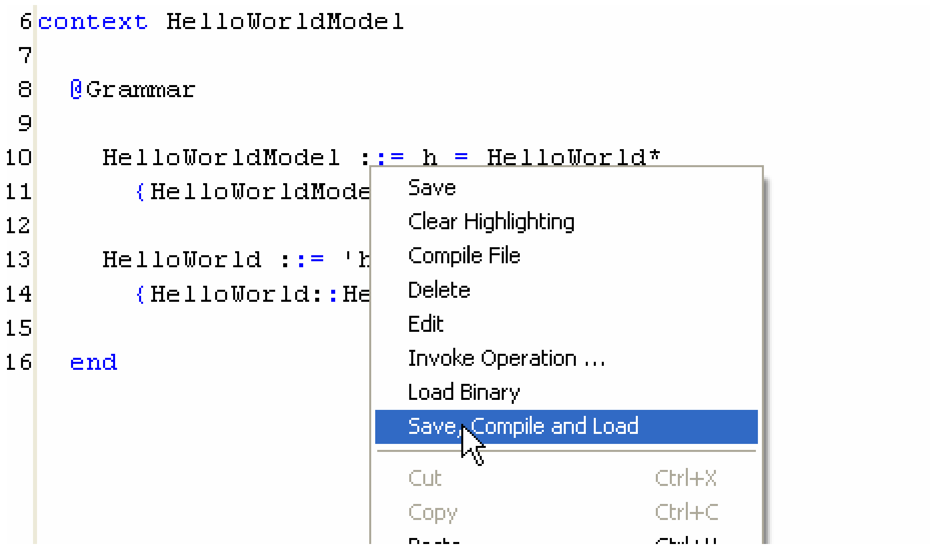
In XBNF, grammar rules consist of a grammar pattern followed by an action (inside the curly brackets).

In this example, the first rule deals with the parsing of a HelloWorldModel. It expects to parse a sequence of HelloWorld rules, the result of which it will assign to the variable h. The * denotes sequence here.

Its action constructs an instance of the HelloWorldModel class and sets the hello Worlds attribute to h. Note the [] brackets are just a simple constructor notation for assigning variable values.

The second rule deals with the parsing of a HelloWorld. It expects the name 'hello' followed by another name, which will be assigned to the variable t. Its action creates an instance of the HelloWorld class and assigns its text attribute to be t.

To save, parse, compile and load the grammar definition, right click on the background of the editor and select Save, Compile and Load as follows:



```
context HelloWorldModel

  @Grammar

    HelloWorldModel ::= h = HelloWorld*
      {HelloWorldMode

    HelloWorld ::= 'h
      {HelloWorld::He

  end
```

2.7.1 *Running the Parser*

Once a grammar has been defined, it can be used to parse code, or used within another grammar definition. The latter use facilities the rapid construction of rich, complex language definitions from simpler ones.

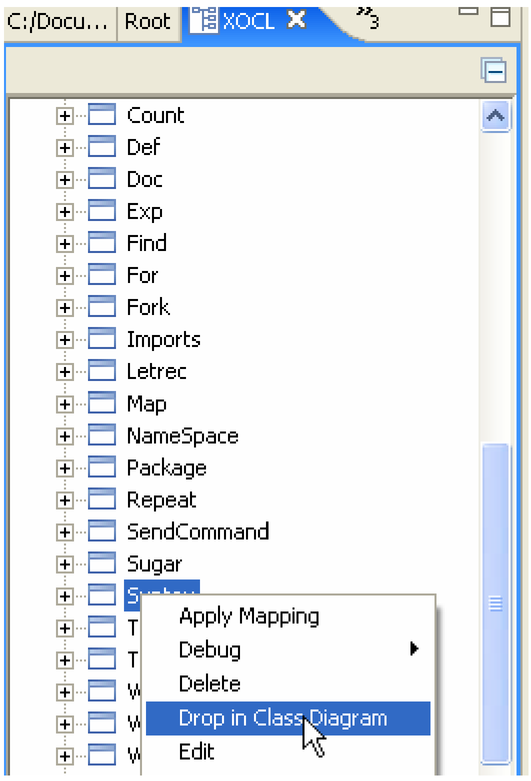
We'll use the grammar as follows:

First create a blank file in the browser and call it say test. Now add the following code:

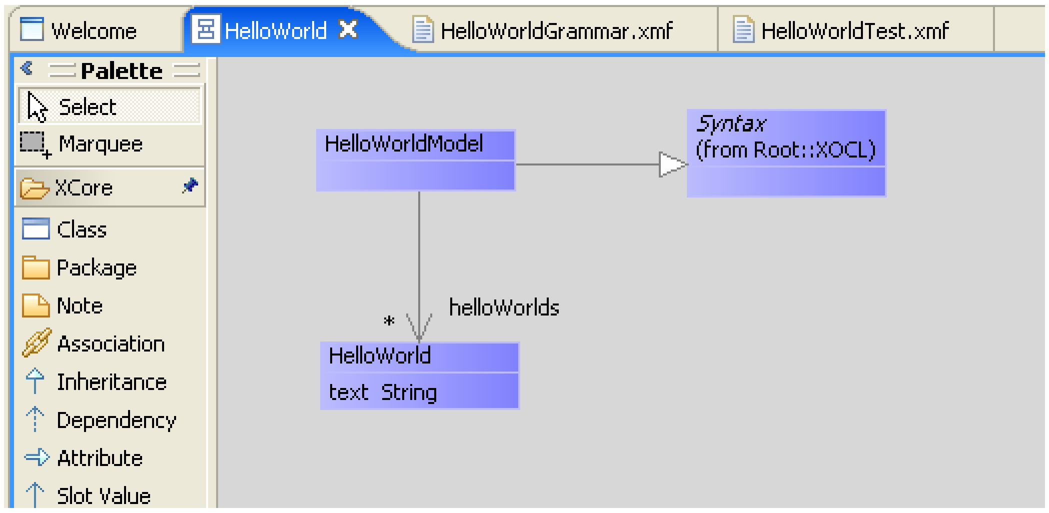
```
parserImport HelloWorld;

import HelloWorld;

Root::p :=
  @HelloWorldModel
    hello Andy
    hello James
    hello Tony
    hello Sheila
    hello Lili
  end;
```

This imports the new parser. In XMF-Mosaic, it is sufficient to parse the container of the parser (in this case the package HelloWorld). It also imports the HelloWorld package (because we will be instantiating its contents).

At this point we could try parsing the code in by right clicking > Save, Compile and Load. However, this will not work. The reason is that when the parser evaluates a grammar it by default calls compile() on the instance of the class returned by the evaluation (in this case HelloWorldModel). However, our model does not define compile() on HelloWorldModel. To get around this we have to make the class HelloWorldModel inherit from a class that supplies a compile operation that just returns the result. This class, called Syntax, can be found in the XOCL package. Drop it into the HelloWorld model by navigating to it from the browser under XMF > XOCL like so and dropping it into the HelloWorld class diagram.

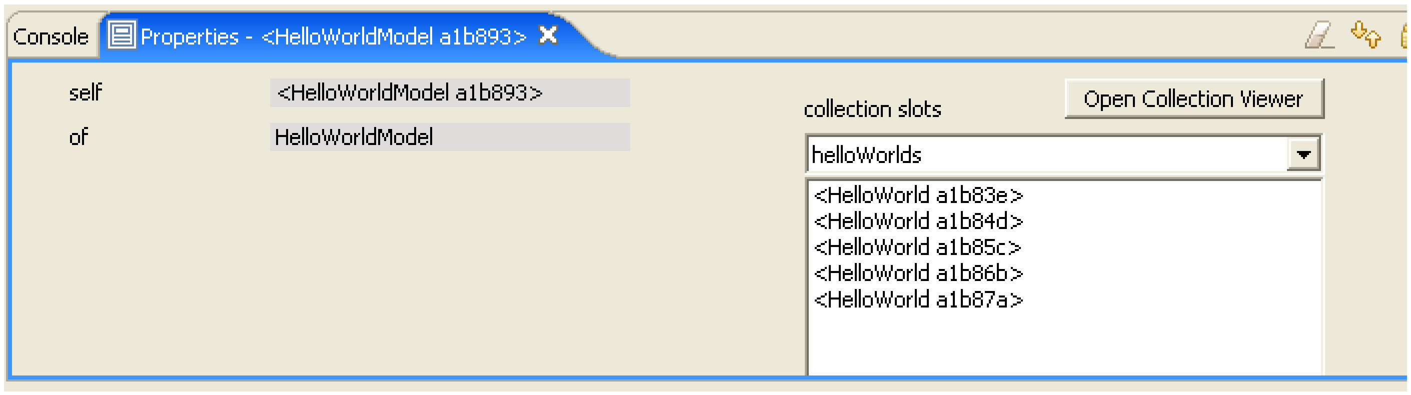


Inherit from it like so:



Now the parser can be run. Go back to the test file and compile and load it.

The result is stored in the variable p. Typing p.edit() in the console will produce the following result:



```
self    <HelloWorldModel a1b893>
of      HelloWorldModel
collection slots
helloWorlds
<HelloWorld a1b83e>
<HelloWorld a1b84d>
<HelloWorld a1b85c>
<HelloWorld a1b86b>
<HelloWorld a1b87a>
```

I.e. the program has been parsed and the model has been instantiated as we would expect.

That's the essentials of defining and parsing grammars. There are many more features of XBNF that are not covered here, including the ability to defer the evaluation of grammar rules. These will be described later.

2.8 Building a Vanilla Diagram Editor

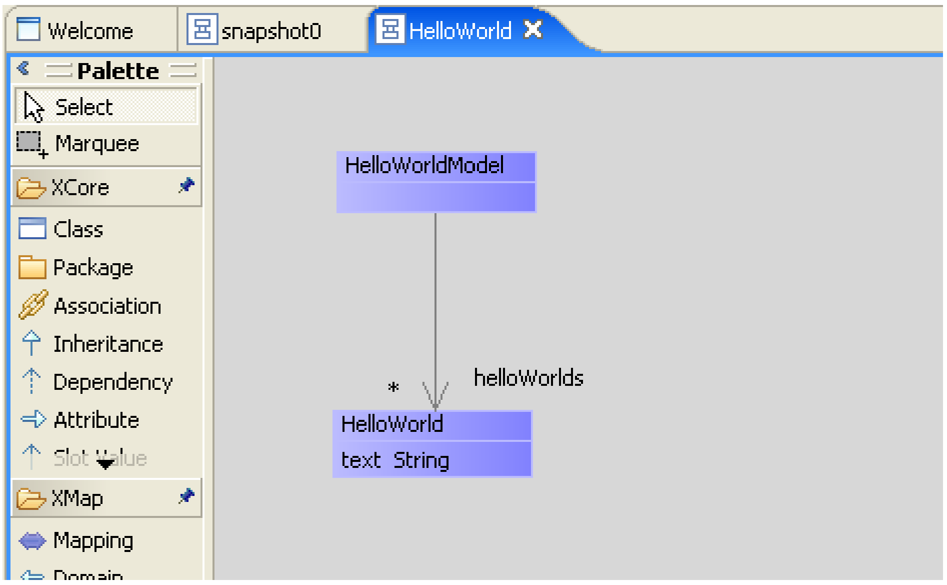
XMF-Mosaic provides a family of user-interface description languages known as XTools. In this example, we are going to show how to generate a vanilla diagram editor from a domain model and take a brief look at the DSL code that is generated and loaded.

Before we do so, we need to add a Root class that contains all HelloWorld concepts.

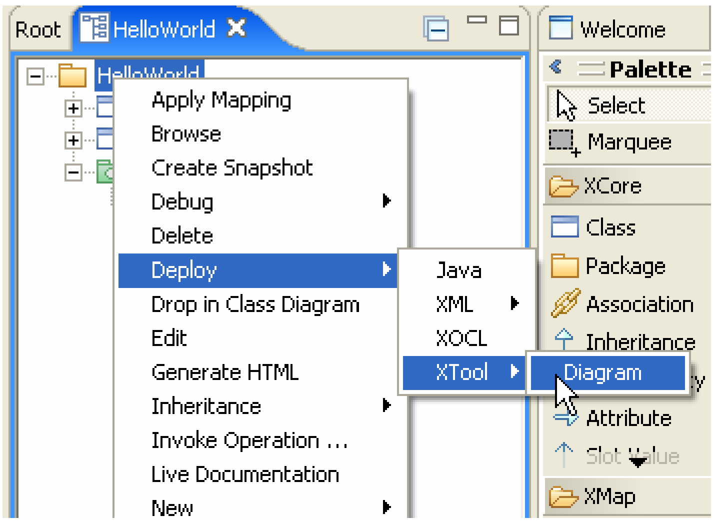
First add a new class called HelloWorldModel to the domain model.

Next add an attribute of type HelloWorld to HelloWorldModel. To add an attribute, click on the Attribute button in the editor, then drag the attribute line from source to the target concept. The name of the attribute can be edited by clicking on it – in this case we make it a plural by appending an "s".

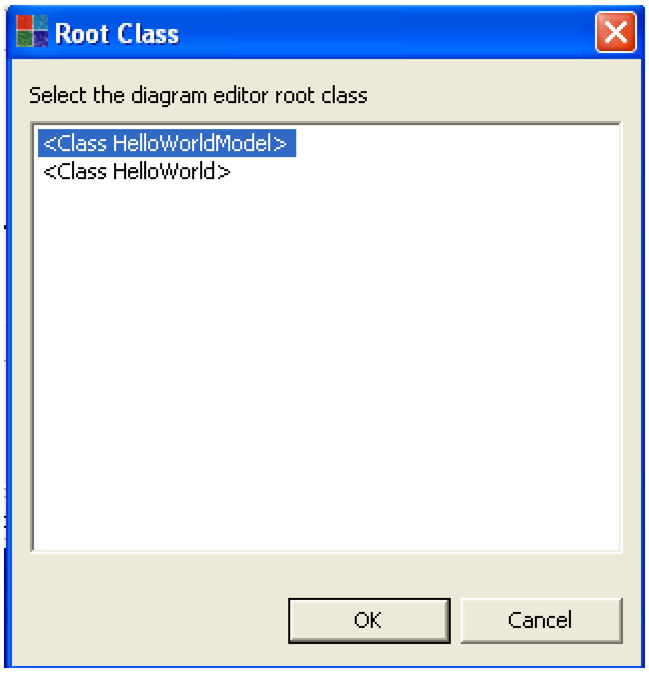
Finally, we need to change the multiplicity of the attribute to be *. Do this by right clicking on the attribute and select Multiplicity then *. The result should look like this:



We can now start the process of generating the vanilla XTool definition and diagram editor. First, right click on the HelloWorld package and select Deploy > XTool > Diagram.



A dialog to select the root class of the editor will appear. Select HelloWorldModel and press OK.


Root Class

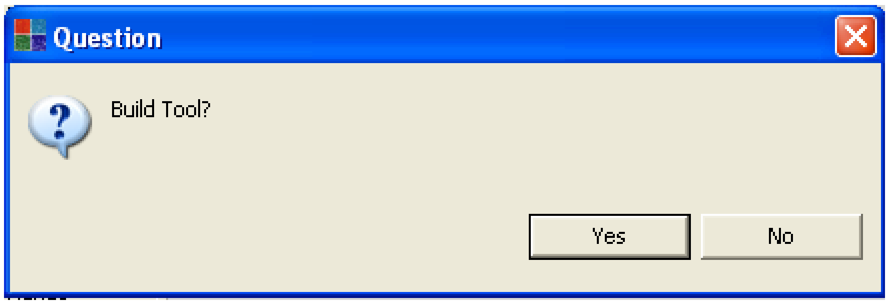
Select the diagram editor root class

<Class HelloWorldModel>
<Class HelloWorld>

Next a directory browser will appear. Choose a directory in which the XTool DSL code will be saved.

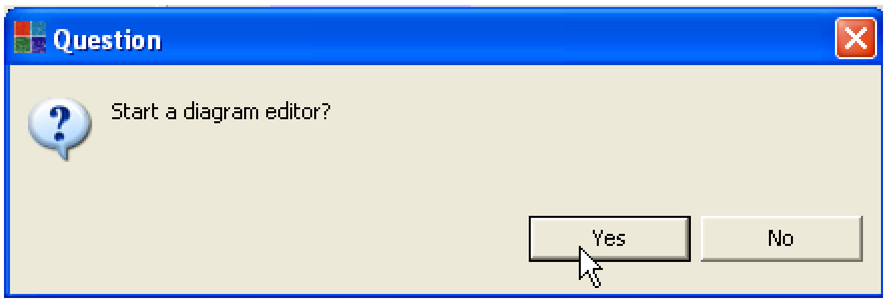
A dialogue will appear where you can choose the name of the tool to be generated. We'll leave it as the default.

The DSL code will be generated, and you will be prompted whether you wish to build it. Answer yes.


Question

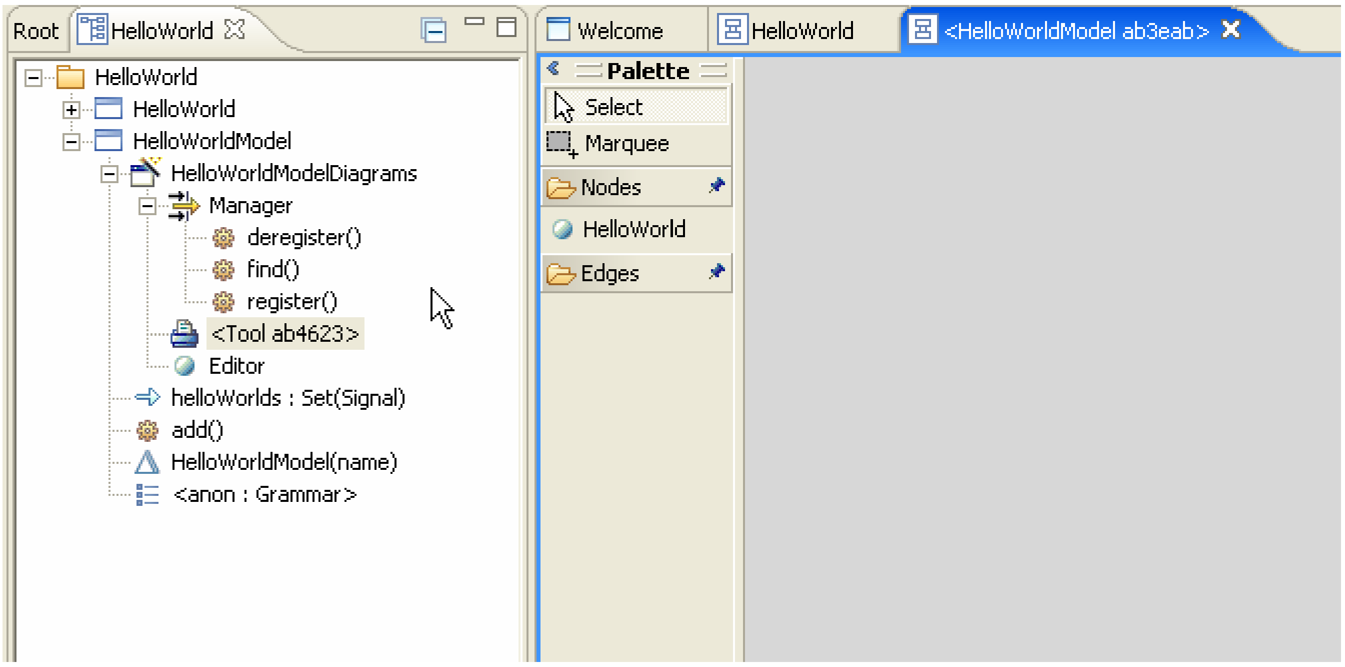
Build Tool?

Finally, a diagram editor for the built tool can be run.


Question

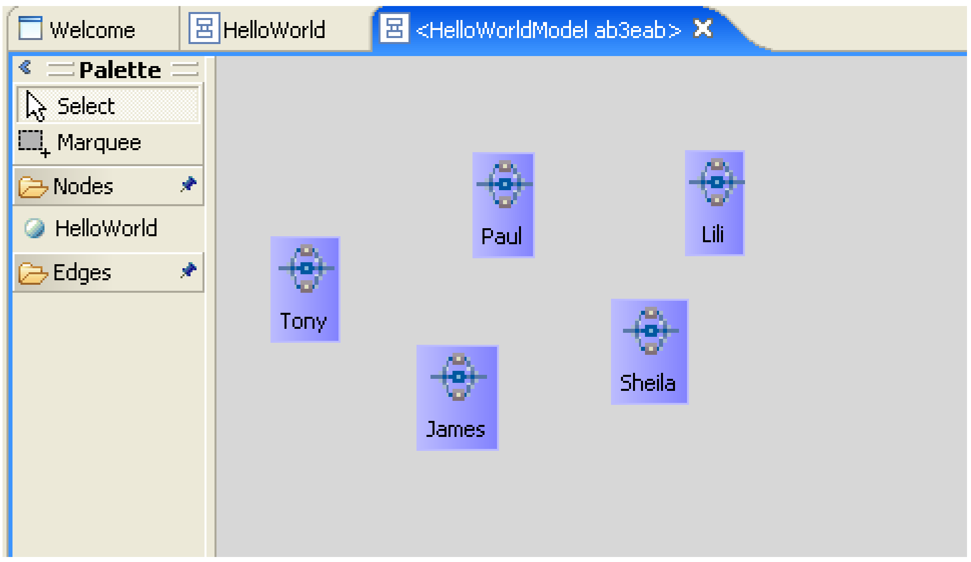
Start a diagram editor?

The result is a new diagram editor with buttons for creating new HelloWorld instances!

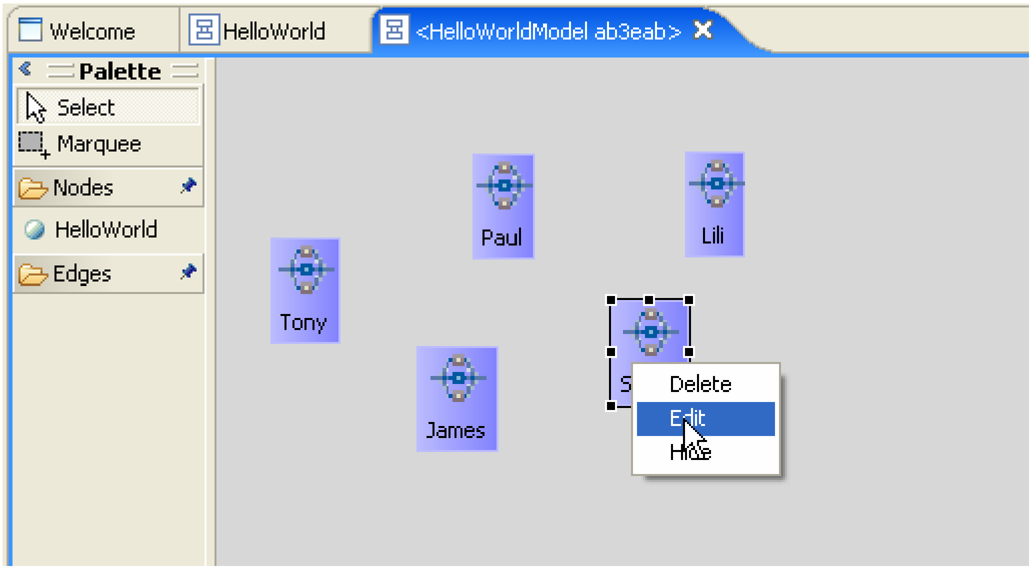


Note that a new diagrams tool type has been added to the class HelloWorldModel. This contains all the tool information necessary to manage the diagram editor and its synchronisation with instances of the domain model.

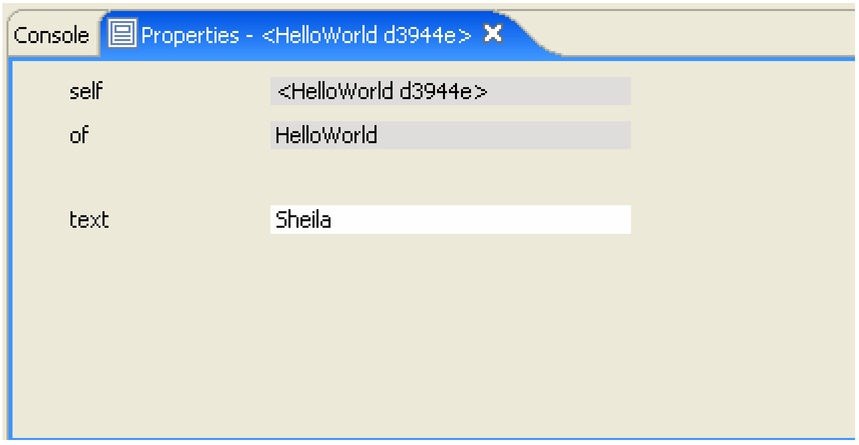
We can now use the new tool to create new HelloWorld instances. Simply click on the HelloWorld button and then click on the diagram. The text value of the instance can be edited by clicking on it. Here's an example:



A number of default menu options are provided.

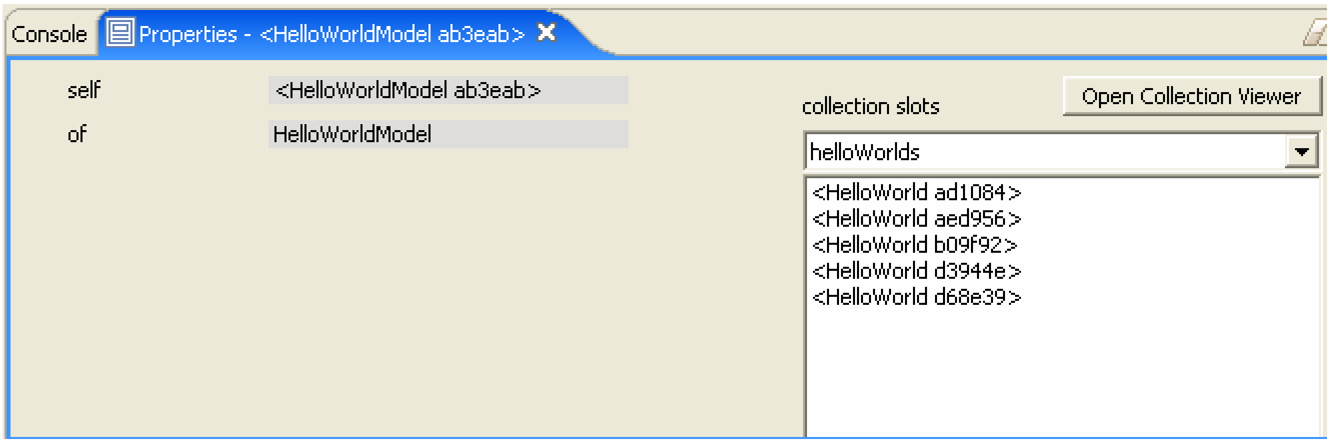


Selecting Edit shows a property editor for the instance of the domain model that the diagram element represents:


Console | Properties - <HelloWorld d3944e>

self	<HelloWorld d3944e>
of	HelloWorld
text	Sheila

Editing the background of the diagram editor will show a property editor for the instance of HelloWorldModel that is being managed by the editor.


Console | Properties - <HelloWorldModel ab3eab>

self	<HelloWorldModel ab3eab>
of	HelloWorldModel

collection slots Open Collection Viewer

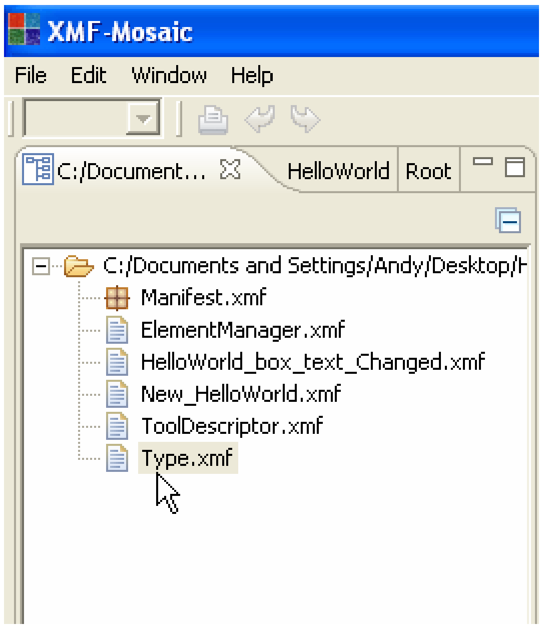
helloWorlds

```
<HelloWorld ad1084>
<HelloWorld aed956>
<HelloWorld b09f92>
<HelloWorld d3944e>
<HelloWorld d68e39>
```

Event handlers for synchronising changes to the diagram with the model are also automatically generated. For example, adding a new HelloWorld instance automatically updates the model (which can be seen in the property editor). Note in the Vanilla XTool the reverse event handlers are not generated.

Finally, let’s have a look at the XTool code itself. This is written using a number of DSL languages that have been specifically designed to support the rapid development of diagram editors (these are just some of the many DSL languages supported by XTools, and indeed the whole of XMF-Mosaic itself).

To view it, go to File > File Browser .. and then select the directory that the XTool code was generated to.

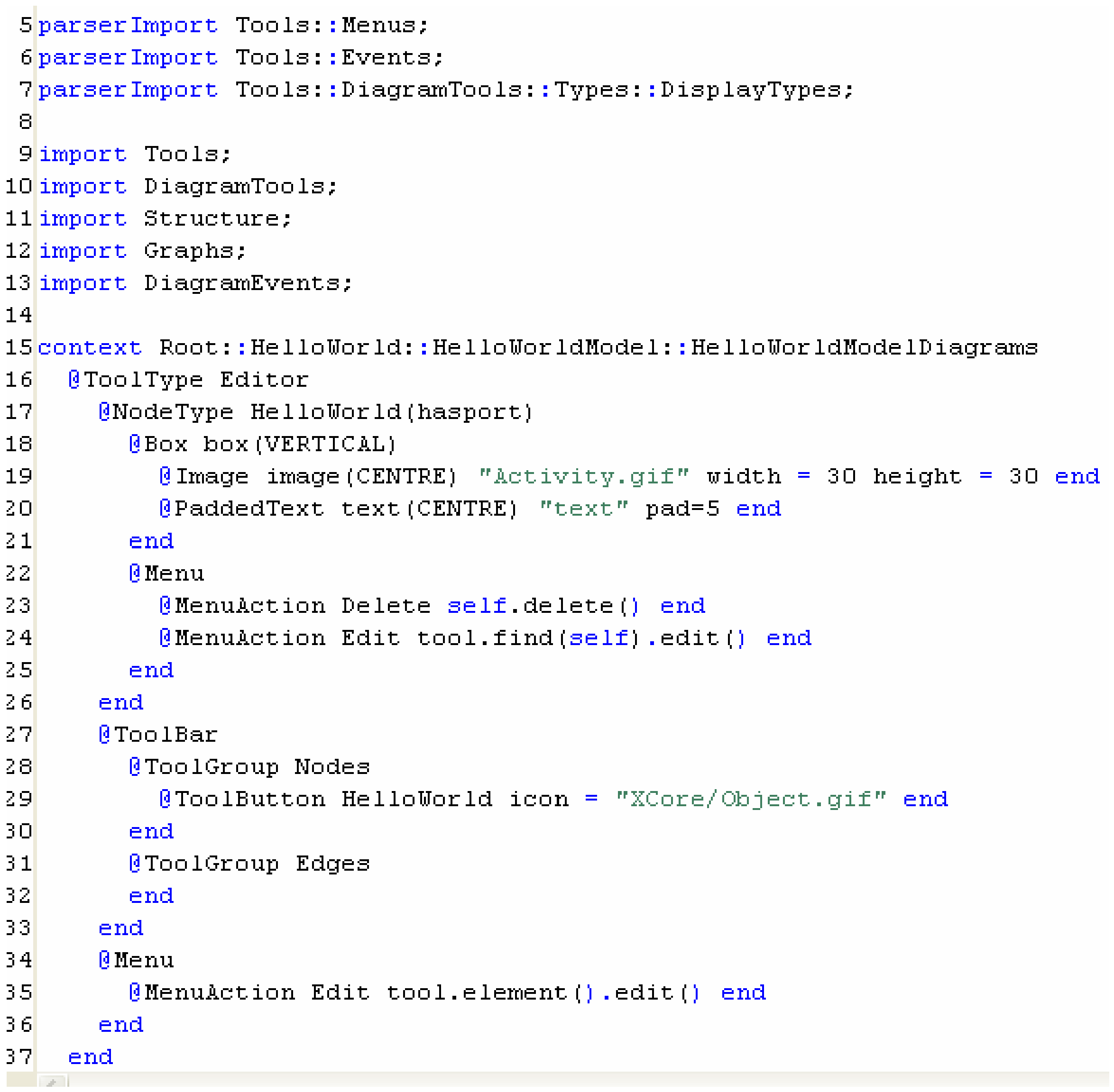


C:/Documents and Settings/Andy/Desktop/H
- Manifest.xmf
- ElementManager.xmf
- HelloWorld_box_text_Changed.xmf
- New_HelloWorld.xmf
- ToolDescriptor.xmf
- Type.xmf

The XTool directory contains the following files:

- A manifest file which manages the building and loading of the tool definition.
- An element manager file, which contains the element manager for the tool. This manages instances of the domain model.
- An event handler which handles changes to the text of a HelloWorld diagram element.
- An event handler which handles the creation of new HelloWorld diagram elements.
- A tool descriptor, which contains the entire tool definition.
- A tool type, which describe the diagram elements that are made available by the tool.

Let’s look at the tool type definition:



```
 5 parserImport Tools::Menus;
 6 parserImport Tools::Events;
 7 parserImport Tools::DiagramTools::Types::DisplayTypes;
 8
 9 import Tools;
10 import DiagramTools;
11 import Structure;
12 import Graphs;
13 import DiagramEvents;
14
15 context Root::HelloWorld::HelloWorldModel::HelloWorldModelDiagrams
16   @ToolType Editor
17     @NodeType HelloWorld(hasport)
18       @Box box(VERTICAL)
19         @Image image(CENTRE) "Activity.gif" width = 30 height = 30 end
20         @PaddedText text(CENTRE) "text" pad=5 end
21       end
22       @Menu
23         @MenuAction Delete self.delete() end
24         @MenuAction Edit tool.find(self).edit() end
25       end
26     end
27     @ToolBar
28       @ToolGroup Nodes
29         @ToolButton HelloWorld icon = "XCore/Object.gif" end
30       end
31       @ToolGroup Edges
32       end
33     end
34     @Menu
35       @MenuAction Edit tool.element().edit() end
36     end
37   end
```

Lines 1-14: This part deals with the import of specific parser definitions that are required by XTools and the import of relevant diagramming packages.

Line 16: The tool type definition contains a definition of the different node (and edge types if we had them) that are available in this tool.

Lines 17-21: This defines a node type, which corresponds to the HelloWorld diagram element. It is a box containing a default image and a text box. A wide variety of formatting parameters can be supplied with different elements. This includes centring, text padding and minimum size.

Lines 22-24: Here we have a menu definition, which is owned by the box. This gives the menu that we saw in the diagram editor and the code that is to be called when a specific menu item is selected.

Lines 27-33: This sets up the tool groups and tool buttons. There is two types of tool groups and one type of node button for the HelloWorld element.

The XMF-Mosaic Bluebook provides in-depth descriptions of how to use this language,

2.9 Semantics

The next step in defining our software development solution is to describe the behaviour of the concepts. XMF-Mosaic supports many different ways of achieving this, including:

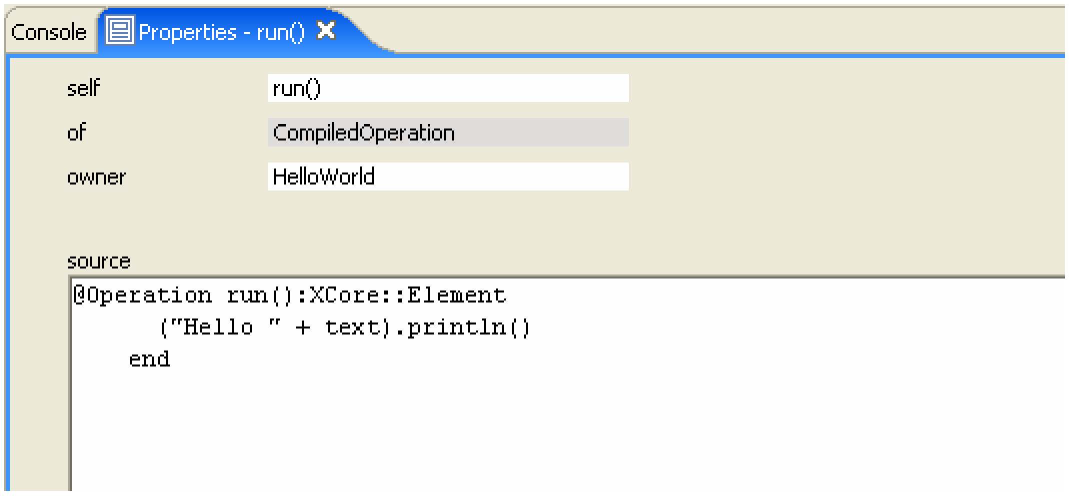
- Writing an interpreter using XMF-Mosaic's internal meta-programming language.
- Writing a compiler to byte code that runs on XMF-Mosaic's Virtual Machine.

Let's build a simple interpreter, which describes how the HelloWorld concept behaves. A HelloWorld concept has a very simple behaviour – it just prints out "Hello" followed by the value of its text variable.

We'll do this by adding an operation to the HelloWorld class called run().

Right click on the HelloWorld class in the browser or diagram editor and select New > Operation. A new operation will appear in the browser. Click on it to edit it.

The code we need to add is as follows:



self	run()
of	CompiledOperation
owner	HelloWorld

source

```
@Operation run():XCore::Element
      ("Hello " + text).println()
    end
```

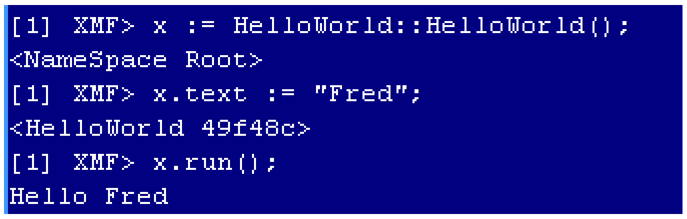
The + operator just appends the variable text to "Hello", and the result is printed using println().

The language we are using is called XOCL – the eXtensible Object Command Language. XOCL incorporates standard OCL (Object Constraint Language) facilities for navigating around models + a Java like action language. One of the key features of XOCL is that it too can be viewed as a DSL, and it can be easily extended with new language features.

We can test the run() operation in two ways:

- At the console, by creating a new instance of HelloWorld and invoking run()
- From a snapshot, by creating an object and Right Clicking > Invoke Operation ..

Here's what happens when we use the console:



```
[1] XMF> x := HelloWorld::HelloWorld();
<NameSpace Root>
[1] XMF> x.text := "Fred";
<HelloWorld 49f48c>
[1] XMF> x.run();
Hello Fred
```

Here, we created a new instance of the class and assigned it to the variable x. We assigned the name "Fred" to its text value and then invoked the run() operation.

2.10 Generation

Often we want to take our domain model and generate another model or piece of code written in another language.

In XMF-Mosaic, you can transform instances of one domain model into instances of another domain model using model to model transformations (we'll talk about these in more detail in another article.

Alternatively, if you have a grammar definition of the external language, you can use it to write code generation templates.

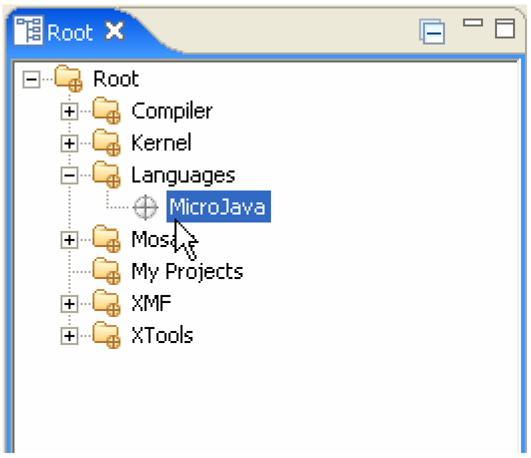
The way that XMF-Mosaic does this is a step on from traditional approaches. Unlike most template based languages which are effectively just textual insertion engines, XMF-Mosaic's templates generate *instances of models by interpreting the target language's grammar*. What does this mean? Well, for a language like Java, we can write Java templates that construct an instance of a model of Java. Pretty printing rules can be applied to the Java model to generate the physical code in any format we want.

This is a very useful and powerful facility for the following reasons:

- It enables a separation of concerns between the abstract syntax of a language like Java and its concrete representation – we could apply many different formatting standards without having to change he Java generation templates in any way.
- Generated models can be easily inserted within other models. This makes doing things like refactoring and reengineering must simpler than if we are working with text alone.

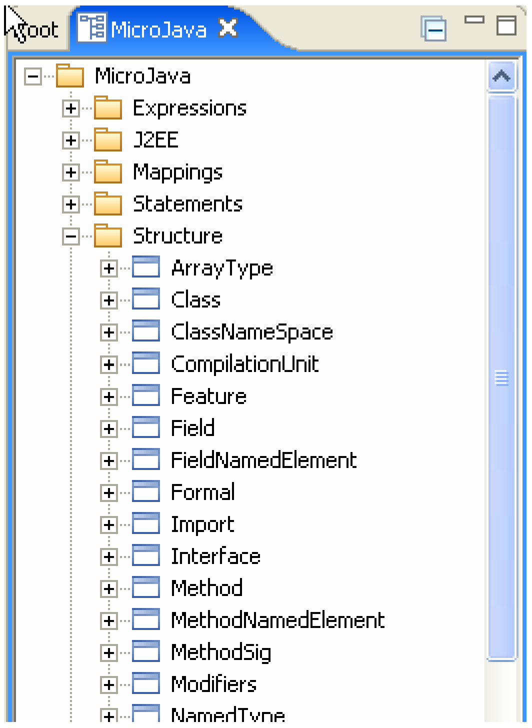
Let's have a go at this now.

First of all, we need to load the model of Java into the tool. To do this, open the Languages project under the Root browser. Double click on the MicroJava project icon.



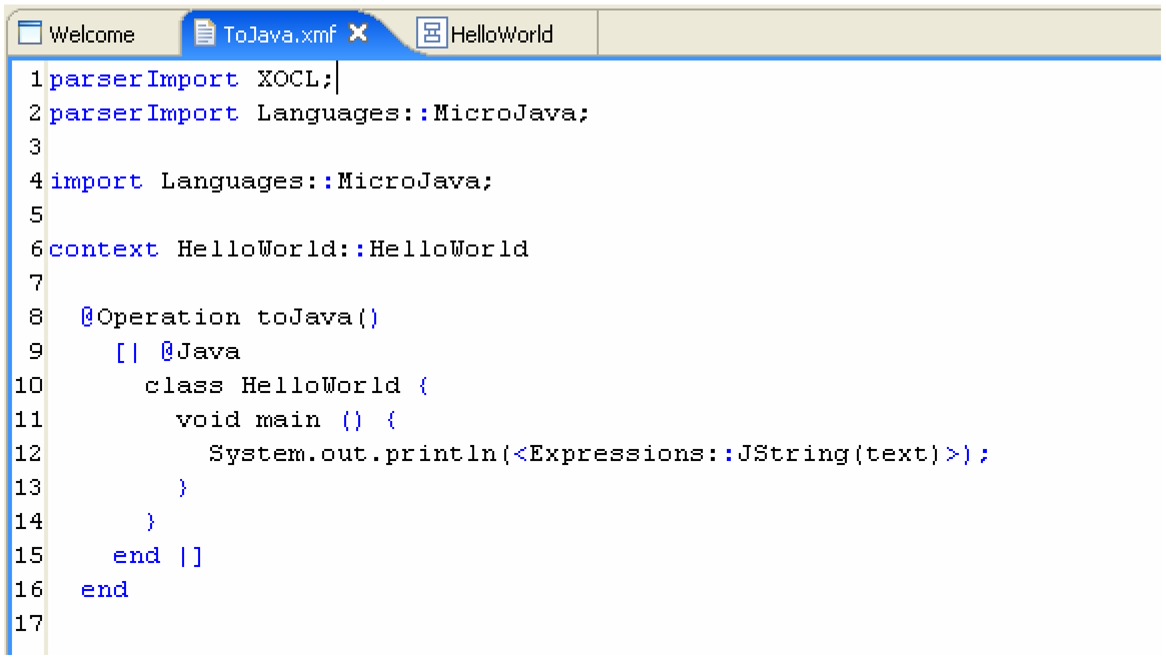
This will load the definition of MicroJava (a subset of Java) into the tool, including its grammar.

A quick look inside the project shows the various parts of the definition. The definition includes models of all aspects of Java, including its Expressions, Statements and Structure + a grammar definition of its syntax.



We can now make use of this to write a Java generator.

Let's create a new file in the file browser (as before). This time we're going to write an operation that generates some Java. Here it is:


Welcome | ToJava.xmf | HelloWorld

```
parserImport XOCL;
parserImport Languages::MicroJava;

import Languages::MicroJava;

context HelloWorld::HelloWorld

  @Operation toJava()
    [| @Java
      class HelloWorld {
        void main () {
          System.out.println(<Expressions::JString(text)>);
        }
      }
    end |]
  end

```

Note that we are importing the parser for MicroJava and the MicroJava model.

We then define an operation in the context of the HelloWorld class that generates some Java.

The body of the operation consists of a meta-template (or quasi-quote). A meta-template begins and ends with the [| and |] symbols. It contains an expression written in a specific language, which when evaluated returns the instance of the language domain model that the concrete syntax corresponds to.

Here the @Java symbol denotes the fact that everything after this symbol is written in Java.

Meta-templates support the dropping of expressions into the concrete syntax using the <> syntax. Thus we can drop the value of the variable text into the println method call as shown.

2.10.1 *Invoking the Generator*

We can invoke the generator simply by calling the toJava() operation on a HelloWorld object. For example, let's create an instance of the HelloWorld class in the console, assign it a text name, and call the toJava() operation on it:

```
[1] XMF> x := HelloWorld::HelloWorld();
<NameSpace Root>
[1] XMF> x.text := "Fred";
<HelloWorld 9c53a2>
[1] XMF> x.toJava();
CompilationUnit(Seq{},Class(HelloWorld))
```

The result of running the toJava() operation is the instance of the Java model class that corresponds to the Java code template.

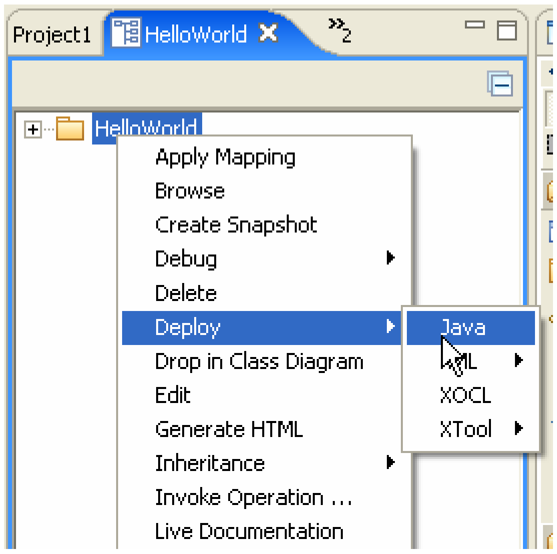
We can convert this to text by calling its pprint operation:

```
[1] XMF> x.toJava().pprint(stdout,0);

public class HelloWorld extends Value {

  public void main(){
    System.out.println("Fred");
  }
}<Format 13e87>
```

As expected, the name "Fred" has been substituted in the expression.

2.11 Using XMF-Mosaic's Java Generator

XMF-Mosaic provides an in-built Java generator for generating Java from models. To invoke this, select the package that you wish to generate in the browser, e.g. HelloWorld and right-click Deploy > Java.



You will be asked to choose a specific directory to generate the files to, and the code will be generated.

2.12 Conclusions

In this guide we've introduced some of the key features of XMF-Mosaic. However, to get a deeper understanding of its capabilities, the XMF-Mosaic Bluebook provides a rich array of walkthroughs and reference guides. To access this go to the Help option.

XMF-Mosaic provides developers with a rich, but consistently architected approach to defining tools that are tailored to their specific business domain and development processes. Our experience of applying XMF-Mosaic to real projects has been an important driver behind its development. Each of its capabilities has an important role to play in many real situations.

In the future we intend to extend these capabilities further with many application specific ones, for example web-services development or testing.