Wonderland Development
Part 2: Intro to Modules

To extend Wonderland's functionality, you need to understand modules. A module is simply a set of Java class files and other resources packaged up into a jar file. This tutorial covers the classes and resources needed in a module. Note that a more technical and extensive tutorial, by Jordan Slott, is available at:

http://wiki.java.net/bin/view/Javadesktop/ProjectWonderlandModuleTutorialPart1Dev3.

1. Introduction

A key problem with big software systems is how to extend them.

For instance, Wonderland can open 3D files in Collada format, but not in .obj format.

One way of adding extra functionality is simply to write more source code, recompile the project, and then tell everyone to scrap the old version and update to the new one.

But a better way is to create some mechanism for adding extra functionality without having to recompile and redistribute the entire system.

This is what modules are for.

A module is basically a plug-in, which contains all the resources needed for this extra functionality. These resources include the compiled program code, any artwork required, and other files such as data and configuration files.

Wonderland is able to access these resources to make use of the extra functionality. So, once a new module for opening .obj files is developed, all you have to do is upload it to the Wonderland server, and in a few seconds, you'll be able to open and display 3D .obj models.

In fact, Wonderland is so carefully designed that a great deal of core functionality, such as the display and control of avatars, is implemented as modules, making it much easier to modify and improve these core systems, or even replace them with completely new ones.

As we go through this tutorial, you'll see how powerful and useful Wonderland's modular architecture can be.

2. Module Interface

Wonderland must be able to access the resources in a module.

This is done via the module's *interface*.

An interface is basically a set of functions (or methods, in Java-speak) that the module implements, and which Wonderland can call. For example, if Wonderland needs to know the name of a module's cell class (see Tutorial Part 1), it simply calls the module's

`getClientCellClassName` method. This method is part of the module's *managed object* class, which extends Wonderland's `CellMO` class.

This raises two questions:

- How does Wonderland know that there is a class in the module that extends `CellMO`?
- How does Wonderland know to call the `getClientCellClassName` method?

The reason why Wonderland knows these things, is that a module won't work without them!

(It's not quite correct to say that Wonderland *knows* about these classes and methods – you, the developer, are supposed to know about them!)

These are all part of Wonderland's module specification. So, if the specification states that a module must contain a class that extends `CellMO`, and that this must have a `getClientCellClassName` method, then you have to follow these instructions.

But before going further into module structure, we need to understand Wonderland in more detail, identifying the necessary parts of a module as we go.

3. Module Contents

3.1 Cells

Wonderland is a client-server application. The virtual world that you see on your client computer is also seen by other users on other clients - this world must be the same for everyone.

Every object displayed in the virtual world is represented by a cell object. (Tutorial Part 1 discussed the idea of cells, which represent 3D volumes in Wonderland.)

For example, if the client displays a 3D model, then the data about this model is stored in the cell object on the client.

Each client has its own copy of the cell object. So, if Nicole and Jon are both looking at the same book object, they will each have a copy of the book's cell object on their computer.

When the client makes a change to an object, say by moving it, this changes the cell object's data (in this case its position data). This change must be sent to the server, which must then update all other clients.

RULE 1: If a module requires an object to be displayed in the world, it must contain a cell class, which must be be a subclass of Wonderland's `Cell` class. This is used on the client-side.

Note that a cell represents a visible object in the virtual world. You can write a module that doesn't display anything, but simply extends Wonderland's functionality in some other way, in which case the module doesn't need a cell class. This advanced topic will be covered in another tutorial.

3.2 Managed Objects

To ensure synchronisation between clients, Wonderland keeps a copy of all the world data on the server. If anything changes (such as the position of an object), this data is updated on the server and then sent to all the clients, which then update their own local views of the world.

All the objects in the world are *managed* by the server.

The objects stored on the server are therefore called *Managed Objects*.

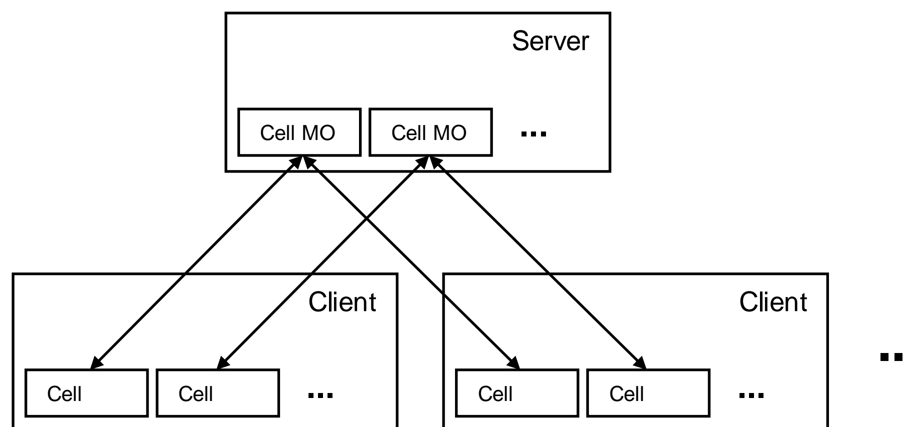
A managed object is the server-side representation of a client-side object. So, while Nicole and Jon each has a copy of the book cell object, the server has its own copy – a kind of master-copy used to keep all clients up to date.

Every managed object class is a subclass of Wonderland's `CellMO` class.

RULE 2: A module must contain a managed object class that extends `CellMO`. This is used on the server-side.

Again, note that this applies only to modules that have some visual object to display, ie. that create a cell on the client.

As mentioned above, the server contains a managed object for each cell, which is used to maintain synchronisation of the cell between different clients.



The relationship between the server and client objects is shown in the diagram above.

Note that the client cell object also communicates with the server - how this is done will be discussed later.

3.3 Client and Server States

When a client connects to Wonderland, it needs to know what objects (cells) are visible, and how they are displayed (position, orientation, etc). This information is obtained from the Wonderland server.

The `CellServerState` class stores data about the state of a cell, and so every cell has an associated `CellServerState` object.

When creating a new cell in a module, you must subclass `CellServerState`. It is this subclass that actually stores the data about your new cell. `CellServerState` is extended to add any new properties needed to display your cell. For example, a cell for displaying an image would need a property to specify the URL of the current image; a new class, say `ImageCellServerState`, would be created as a subclass of `CellServerState`, and an `imageURL` property added to it.

Every cell in Wonderland has its own specific subclass of `CellServerState`.

`CellServerState` objects are used in two ways:

1. To store data about a cell when the Wonderland server first starts

   Data about all the cells in a Wonderland world are stored in xml files on the server. This is part of the Wonderland File Service (wfs). When the server starts, it reads the xml files, and stores the data for each cell in the cell's `CellServerState` object. When a client logs on to the world, these `CellServerState` objects are sent to the client, which then uses this data to display the virtual world locally.

2. To update clients when a cell is modified in some way

   If a client notifies the server that it has changed a cell in some way (eg. selected a different image to display), the new state of the client cell is sent to the server and stored in the cell's `CellServerState` object. The server then sends this object to all other clients, which then update their copies of the cell with the new data.

Data about the client's copy of a cell must be sent to the server.

This is done using a `CellClientState` object.

Every cell has its own specific subclass of `CellClientState`.

So, when a client modifies its local copy of a cell in some way, it stores the updated data in its `CellClientState` object. This object is then sent to the server, which stores the updated data in its `CellServerState` object, which is then sent to all the other clients.

You can think of it as water (data) being passed from one cup to the next. The client has its own cup, which it fills with water. It then pours this water into the server's cup, which then pours it into all the other clients' cups.

If you're curious about the details, look in the Wonderland source's `CellMO` class. You'll find two methods, `handleGetStateMessage` and `handleSetStateMessage`, which use `CellServerStateSetMessage` and `CellServerStateRequestMessage` objects, containing a `CellServerState` object.

RULE 3: A module must contain a subclass of `CellServerState`, and a subclass of `CellClientState`. These are used on the client-side AND server-side.

Note that this mechanism *might* change in future.

Also note that a server state class from one module can be used in another. (See the Portal module, which creates a `JmeColladaCellServerState` object to store the server state data for its 3D model – though the actual portal properties, such as server URL, are defined by `PortalComponentServerState` and `PortalComponentClientState` classes,.)

3.4 Renderers

An object must be displayed in Wonderland for people to see it. The technical term for displaying an object is *rendering*. Different objects need to be rendered in different ways; for example, a simple 3D model is rendered differently from a dynamic piece of cloth that flaps in the wind.

Code for rendering a simple object might look something like this:

```
private Node scene;
private Geometry geom;
scene = new Node();
scene.setName(cell.getCellID().toString());
scene.attachChild(geom = new Sphere("Sphere", 10, 10, 1.0));
geom.setModelBound(new BoundingSphere());
geom.updateModelBound();
MaterialState matState = (MaterialState)
ClientContextJME.getWorldManager().getRenderManager().
      createRendererState(RenderState.RS_MATERIAL);
MaterialJME matJME = ((SimpleShapeCell)cell).getMaterialJME();
if (matJME!=null)
     matJME.apply(matState);
geom.setRenderState(matState);
```

(Taken from testcells-module/ShapeRenderer.java)

The rendering code must be placed inside a renderer class. This renderer class should subclass `BasicRenderer`.

RULE 4: A module must contain a renderer class that is a subclass of `BasicRenderer`.

Again, this obviously applies only to modules that have some visual object to display.

3.5 Cell Factories

When a module is added to the server, an instance of the module's cell is created via the client side world-building tools. The main tool is the Cell Palette, which lists all available cells.

For a cell to appear in the Cell Palette, a `CellFactory` class must be implemented. This class must implement the `CellFactorySPI` interface. The Javadoc for the `CellFactorySPI` interface states that:

A CellFactorySPI class is responsible for generating the necessary information to generate a new cell. This includes: a default cell setup class; a display name and image to be used in a palette of cell types; a list of file extensions which can be rendered by this cell type.

And in case you're wondering what SPI stands for, here's an explanation from Jordan Slott:

SPI stands for "Service Provider Interface". An API is an interface that is provided by the platform and used by applications. An SPI is an interface that is provided by the platform and used by lower-level software, for example, "device drivers".

RULE 5: A module must contain a factory class that implements `CellFactorySPI`.

This also applies only to modules which implement a cell.

3.6 Components

Components are classes that implement some functionality used by a module. For example, a component may be written to automatically rotate an object, such as a pick-up item in a game.

We won't go into components in this tutorial – but note that components are very useful, since a component developed for one module can be used in another. As you become more familiar with Wonderland module development, it pays to take the time to abstract your module's functionality into reusable components.

3.7 Other Resources

Modules may require other resources, such as artwork. These can be added to the module, and accessed by the module code.

For example, the Portal module includes a Portal2.png image file, which is used as the preview image for the Cell Palette. This image file is accessed in `PortalCellFactory`.

Note that art-related resources, such as images, should be placed inside a folder called **art**. The files are then loaded by using a special URL. We'll look at how to do this in the next tutorial.

4. Module Structure

Now that we've taken a good look at the main elements of a module, we can examine how a module should be structured.

A module is simply a jar file. A jar file is basically a zip file that contains Java-related resources. (Java is a type of coffee, so what holds coffee? – A jar! JAR also means **J**ava **AR**chive.) The files are located in various packages. There are four main packages, each of which contains either:

- Classes that are used on the client side
- Classes that are used on the server side
- Classes that are are used on both the client and server sides
- Other resources, such as image files

If you look at the samplemodule, in Wonderland's modules folder, you can see that there are four packages, which include the files listed:

- org.jdesktop.wonderland.modules.sample.client ← *Classes used on the client*
  - SampleCell.java
  - SampleCellFactory.java
  - SampleRenderer.java
- org.jdesktop.wonderland.modules. sample.server ← *Classes used on the server*
  - SampleCellMO.java
- org.jdesktop.wonderland.modules. sample.common ← *Classes used on both*
  - SampleCellClientState.java
  - SampleCellServerState.java
- org.jdesktop.wonderland.modules. sample.client.resources ← *Resource files* *
  - sample_preview.jpg

(* A newer, and better, way to include art related files is to use the art folder, as mentioned above. Using a resources package is still an alternative, though.)

The module also includes other files, which we don't need to look at now.

When you build your module, the package structure tells Wonderland which classes should go where: on the server, the client, or both.

And just like any other Java project, you can create more packages than those needed by Wonderland.

For example, an image processing module would have all the necessary classes and packages for interfacing with Wonderland, but may also include other classes for image processing, not directly needed by Wonderland. These classes would be used internally by the module. The package structure might look something like this:

- org.jdesktop.wonderland.modules.imageprocessor.client
  - ImageProcessorCell.java
  - ImageProcessorCellFactory.java
  - ImageProcessorRenderer.java
- org.jdesktop.wonderland.modules.imageprocessor.server
  - ImageProcessorCellMO.java
- org.jdesktop.wonderland.modules.imageprocessor.common
  - ImageProcessorCellClientState.java
  - ImageProcessorCellServerState.java
- org.jdesktop.wonderland.modules.imageprocessor.client.resources
  imageprocessor_preview.jpg
- **org.jdesktop.wonderland.modules.imageprocessor.client.filters**
  - **Blur.java**
  - **Sharpen.java**
  - **Grayscale.java**

If you wanted to use, for example, the `Blur` class in `ImageProcessorRenderer`, all you'd have to do is add the line

```
import org.jdesktop.wonderland.modules.imageprocessor.client.filters.Blur;
```

to the top of ImageProcessorRenderer.java.

You can see that there's nothing funny or difficult about a Wonderland module file. It's simply a regular Java jar file that conforms to a specific structure, and which follows a few simple rules about how to name your packages.

5. Review

So far, we've covered the basics of Wonderland modules.

We've looked at:

- Why modules are so useful and powerful
- The basics of how Wonderland interfaces with modules
- The contents of a module
- How these contents are structured inside a module

Hopefully, you can see that modules are nothing magical – just a bunch of Java classes and other resources packaged up into a jar file for Wonderland to access according to a few simple rules.

Also, the modular system is very flexible, so it's a good idea to look through different modules to understand all the different ways of doing things.

Modules can be very simple, or very complex.

An example of a simple module is one for displaying a primitive object.

A much more complex module is Wonderland's avatar system.

In the next tutorial, we'll go through actually writing a new module. The module won't be so simple, since we can touch on other interesting topics by tackling a more complex problem. And remember, a great way to learn about modules is to go through Jordan Slott's tutorials on the Wonderland wiki.