

# UI Design, Plug-In Development, and Movie Making with Java Media Framework

Christopher Stelma  
Independent Study 20041, 20042  
Professor Hans-Peter Bischof  
Report Submitted: 18 May 2005

# UI and Plugin Software

## Introduction

I started working with the GRAPEcluster project in September of 2004. In an attempt to work on something that was slightly familiar, yet still embedded in the system, I chose to work with the team whose task was to redesign the UI for usability and extensibility. The first goal was to reduce the learning curve for the UI. The second goal, and one less visible, was to make it easier to add elements and features to the UI.

The original UI was usable, but was not designed to be easy to use. There was one window, in which there existed a mass of buttons, a few sliders, and not much else. We quickly went through a few concept designs, and finally decided on a rectangular layout, divided into four equal smaller rectangles in a grid pattern. The design layout hasn't changed much since then. The idea was to break up the large blob of UI into several smaller chunks.

The upper left section of the UI was to be a Java3D canvas in which the simulation would appear. Under the original system, when the simulation was loaded, it would appear in an entirely separate JFrame; this is still an artifact today.

With the idea that the simulation would be in the upper left, and that all ideal features would be implemented, there would be a way to click on a star or black hole in the simulation, and somewhere information about it would appear. We reserved the upper right corner for display of this information.

A member of the team suggested a history window, so that there would be feedback of what happened when the user did something with the controls. This was put in the lower left section.

The most important section of the four, and the only one that ever gets much use, is the "control" section. Since it would be cumbersome to cram all of the buttons into this small area, and would look rather horrible, the buttons were grouped and put into tabbed panes. Specifically, the controls would be grouped by similar function and/or by relationship. We couldn't decide on exactly what the logic should be behind the grouping, but settled on grouping by function. For example, buttons that deal with files are in one group, and the set of buttons and sliders that control the camera position are in another. Finally, most other buttons, such as those which run a certain pre-determined flight through the simulation, stopping the simulation, and moving forward in simulation time are located in what was the last section.

Since one of the goals was to make it easier to add things to the UI, instead of having to edit the main program code every time, I started to work on a system of plug-ins for the system. My goal was to be able to add UI elements without editing any code in the main program whatsoever. Also, since this feature was centered on the idea of pushing functionality to an already installed system, a way to add and remove plugins without recompiling was needed.

This functionality requires use of Java's Classloader. Polymorphism is also taken advantage of to treat all plugins the same way through an interface. However, in order for the Classloader to be able to load anything, one has to know the name of the class to be loaded. Since we want to be able to add any new functionality without the system caring what it is, the class name needs to be derived before it can be passed to the Classloader. While it is easy to define a certain directory to be the location of the plugins, and to find the plugin directories and classes within it, determining which class to load is not as easy. As I understood it, there were two easy ways to do this; one, require all plugins to include a class with a common name for loading and two, require meta information to be located in a commonly named file. Renaming the already existing classes

seemed an inferior idea, and so the use of a meta file was used. The use of a meta file allowed for a fairly simple process of converting any existing UI element or system into a plug-in.

## Extensibility

The original UI code followed the "blob" design. Anytime a new feature was added, it got stuck in. The same is true now, but now it is either added to an existing plugin or a new feature set is added as a new plugin. In either case, the feature is contained within a subset of the UI. The code for the feature is local to this subset, and guarantees that when it is edited, code outside of the plugin is not altered. If something breaks due to an edit, the problem can be isolated to a single plugin, and is most likely due to the latest change made. The ability to swap plugins in and out ensures that at least most of the UI and/or system is functioning at any one time. In the old system where one Class was responsible for the entire UI, if something broke, the whole system went down. Using plugins, development can happen in isolation, working out the bugs before integration, and even then, the system does not become wholly dependant on the new section to function.

The modularity of the system also means that the main code does not have to be changed all the time. Once MapViewUI became stable, there was no longer any need to edit it as far as UI components went.

## Meta File

XML was chosen as the format in which to represent the plugin manifest file. There were two main reasons for this decision; one, it was an established format for representing arbitrary amounts of data and two, Java comes with an XML parser. These combined, it was extremely easy to get data for unique identification into Java, while allowing some freedom with which the user organizes the manifest. Within the limits of XML, one can organize the visual appearance of the manifest however they choose. In retrospect, XML was probably overkill as a way to represent the data, since so little data is actually contained in the manifest. However, the format allows for further development without breaking backwards compatibility. Also, the hierarchal structure of XML may be useful in the future for representation of parts of the UI, since Swing components are also hierarchal.

The original inspiration for the plugin system and also the xml file came from using Eclipse (eclipse.org). The idea that the entire system is built from plugins seemed like a nice model to follow. Using the same name for the file ("plugin.xml") as is used in Eclipse turned out to not work so well. This was because if one attempted to edit "plugin.xml" in Eclipse, it would be interpreted as a meta file for Eclipse. Adding "grape" to form "grapeplugin.xml" solved this problem.

One problem with the current "MapViewUI.java" is that the system should theoretically be able to load without any plugins present, but will fail to do so if the history window is not present. This was originally a quick fix to include the history window plugin, and was replaced with a more robust fix, which has not yet made its way into the code base. Since this plugin is a heavily used one (in terms of code), it is not likely to be removed, however, this is still a bug. The more robust

solution was to add another method to the Plugin interface (for a total of two), to allow communication of Objects to a plugin.

Other information can also be stored in the XML file. When integrating the movie making client (discussed below) into the UI, I included tags (and the supporting code to use them) for server and port. This allowed the user to set a static server and port, which would be loaded automatically into the client. This adds convenience, but does not prevent connecting to a different server and port during the session. A possible improvement for the future would be to allow the user to save this information to the XML file, but the current system is not set up to allow editing the XML from within the program. However, the current system does not prevent it, and allows fairly quick access to the needed File object.

XML tags used by the system are shown in the example below.

```
<plugin>
<class>InfoWindow</class>
<position>2</position>
<type>main</type>
<registeras>Star Info</registeras>
</plugin>
```

This is the grapeplugin.xml file from the "InfoWindow" plugin. The outer tag "plugin" is present only to separate the plugin data from anything else in the file. This leaves room for multiple plugins to be defined by the same XML file, if an implementation was created to use it. By the current code structure, the separation is ignored, so this would require a major rewrite of the parsing code.

The tag "class" defines the name of the .class file which implements the Plugin interface within the same directory. So, according to this grapeplugin.xml, "InfoWindow.class" contains a valid implementation of a Plugin.

The tag "type" determines what kind of plugin we have. Types of plugins currently defined and in use are "main", "control", and "Input3d". A type of "main" defines a plugin which will be one of the four major sections of the UI. "control" defines a plugin which should be loaded as a tab within the controls plugin. The controls plugin is of type "main". The type "Input3d" is a plugin which is loaded by the Inputs3d plugin, which itself is a "control" plugin. The structure behind loading is discussed more in depth below.

The tag "position" represents the order in which plugins will be added to the UI. So, in the example, position 2 will mean that this plugin will be added to the UI after any plugin of type "main" whose position is less than 2 and before those with a position greater than 2. Knowing that the other plugins of type main have positions 1, 3, and 4, this means that the InfoWindow will be loaded second into the main UI. The main UI uses a 2x2 GridLayout. Since the GridLayout loads from left to right and then top to bottom (at least in a 2x2), this means that InfoWindow will be in the top right. If, for example, one wanted to swap the position of the InfoWindow with the plugin in the lower right (position 3), one could simply edit this grapeplugin.xml and put "3.5" in the type field. One could also edit both files and swap the numbers, which would be necessary if they were not adjacent, but in this case, since position is a floating point, only one edit is required.

The tag "registeras" (Register As) is used to uniquely identify the plugin within the system. The system uses a Map from a String representation of this field to a PluginCapsule Object which holds the information about the plugin. Therefore, there cannot be two plugins present with the same registration. As a simple protection, if a second plugin tries to register using the same name, "duplicate plugin registration" is sent to the error stream defined by "InOutErr.err".

## Plugin Classes

There are a handful of classes used for the plug-in system. The main class is called “PluginLoader”. Other classes include Plugin, PluginCapsule, and PluginException. PluginException is used to identify separate errors when loading a Plugin from similar errors in other places.

### Plugin.java

The Plugin interface currently has a single method:

```
The single method of the Plugin interface
public Component build(MapViewUI loaderUI);
```

This method is designed to give the plugin access to the rest of the system, and to be part of the UI if it should be. A simple Component is returned, in which the Plugin will have built its entire UI, if it has one; a plugin is not actually required to return a UI. It may return null, and could also run without any interaction with the rest of the system. I don't believe any current plugins do this, however, it may be useful in the future.

### PluginCapsule.java

The PluginCapsule class is mostly a data object. Its only job is to hold relevant data about the plugin, and to be a single reference that can be passed around. Fields in the XML file common to all plugins are loaded and stored when the PluginCapsule is instantiated, so that they do not have to be parsed more than once. The Capsule can also instantiate the Plugin once.

### PluginLoader.java

PluginLoader does most of the work determining what a plugin is, where it goes, and any other information associated with it. When PluginLoader is instantiated, it doesn't really do anything. This separates instantiation from the time when the work is done. No function of PluginLoader should be called before load(). This Class has only three variables which are not constant. There is a boolean value to make sure that the load() method only works once, otherwise it will just not do anything. Another variable is a Map with String keys "registeras" data parsed from the XML meta file to PluginCapsule values holding the remaining useful data about a plugin. Upon instantiation, this map is empty. Another Map exists with the same mapping, used to hold only references which have not yet been "claimed" to be loaded.

In order for a plugin to make it successfully into the system, once it has been written and implemented as a Plugin, several things need to happen. The XML file must be found and successfully parsed. Once this happens, the Class file for the Plugin must be successfully located. Then, the full class name must be generated and used to load the Class object for the Plugin from the file. The rest of the PluginCapsule must be populated as well. PluginCapsule just calls methods provided by PluginLoader, which does all the work.

## getStringFromPlugin() and getClassFromPlugin() code discussed

```
public static String getStringFromPlugin(PluginCapsule capsule,
    String tagname) throws PluginException {
    Document doc = capsule.getPluginDoc();
    NodeList tagList = doc.getElementsByTagName(tagname);
    String tagValue = null;
    NodeList childrenToTest = tagList.item(0).getChildNodes();
    searchForText:
    for(int i = 0; i < childrenToTest.getLength(); i++){
        if (childrenToTest.item(i).getNodeName() == Node.TEXT_NODE){
            tagValue = childrenToTest.item(i).getNodeValue();
            tagValue = stripWhitespace(tagValue);
            if ( !(tagValue == null || tagValue.equals("")) ){
                break searchForText;
            }
        }
    }
    if (tagValue == null || tagValue.equals(""))
        throw new PluginException("tag " + tagname + " is malformed");
    return tagValue;
}

public static Class getClassFromPlugin(PluginCapsule capsule)
throws PluginException {
    File enclosingFolder = capsule.getDefFile().getParentFile();
    String classFileName = getStringFromPlugin(capsule, "class");
    String classFileStr = enclosingFolder.getPath() + separator
    + classFileName + ".class";
    File classFile = new File(classFileStr);
    if (classFile.exists() && classFile.isFile() && classFile.canRead()){
        InOutErr.err.println("Class File \'' + classFile.getName()
            + '\'' was good");
    } else {
        InOutErr.err.println("Class File \'' + classFile.getName()
            + '\'' was not good");
    }
    boolean notdead = true;
    File currtrace = classFile.getParentFile();
    String classname = "." + classFileName;
    while (notdead) {
        classname = currtrace.getName() + classname;
        currtrace = currtrace.getParentFile();
        if (currtrace == null) {
            notdead = false;
        } else {
            classname = "." + classname;
        }
    }
    classname = "spiegel." + classname;
    Class myNewClass = null;
    try {
        myNewClass = Class.forName(classname);
    } catch (ClassNotFoundException e) {
        InOutErr.err.println("ok, somebody told me that " + classname
            + " existed, but I can't find it");
        e.printStackTrace();
    }
    return myNewClass;
}
```

The function that does much of the parsing work is `getStringFromPlugin(PluginCapsule capsule, String tagname)`. This method uses the Document representation of the XML file to find all Nodes that are the same as "tagname". Nodes are Java's representation of the hierarchal tree structure of XML; lower Nodes are considered child nodes of those higher up. The first such match is used, since currently there should be only one such tag in the Document. To account for any extraneous data within the tag being located, every child node is examined by type until one is found that is a text node. A text node is one such as "`<tag>text</tag>`", where "text" is a text node. It is possible to have more than one, since nodes are broken up by tags; "`text<newtag><newtag/>text`". Again, the first such match is used. This text is then returned, after any leading or following white space has been removed. The function `getDoubleFromPlugin` is included only for convenience; it calls `getStringFromPlugin` and attempts to convert the String into a Double. "`getClassFromPlugin`" is somewhat more complicated. A call is made to `getStringFromPlugin` with the tagname argument of "class". The abstract pathname is generated. This path is traversed up to the folder from which the program was launched and the top level package name is added since launching usually occurs from within the "spiegel" directory. Finally, `Class.forName(String)` is called to get the Class.

`findplugins()` code discussed

```
private static List findplugins() {
    File pluginfolder = new File("viewcontrol"
        + separator + "mapview" + separator + "plugins");
    List pluginfiles = new ArrayList(Arrays.asList(pluginfolder.listFiles()));
    ArrayList pluginFilesToRemove = new ArrayList();
    for (Iterator i = pluginfiles.iterator(); i.hasNext();) {
        File current = (File) i.next();
        boolean keep = false;
        if (current.isDirectory()) {
            List singlePluginFiles = new ArrayList(Arrays
                .asList(current.listFiles()));
            for (int j = 0; j < singlePluginFiles.size(); j++) {
                File insideCurrent = (File) singlePluginFiles.get(j);
                if (insideCurrent.getName().equals(pluginDefFilename)) {
                    keep = true;
                }
            }
        }
        if (!keep) {
            pluginFilesToRemove.add(current);
        }
    }
    pluginfiles.removeAll(pluginFilesToRemove);
    return pluginfiles;
}
```

The purpose of the `load()` method in `PluginLoader` is to populate the maps with `PluginCapsules`. The `findplugins()` function returns a list of directories within the "plugins" directory that contain a "grapeplugin.xml". "`getDefinitionFiles`" is then called to return a list of those XML Files which actually exist. From this list of XML Files, `PluginCapsules` are instantiated. They are then put into the registration Map, and an error is generated if a capsule with the same registration has already been added. After all valid plugins are registered, the registration Map is copied into the "unclaimed" Map. These `PluginCapsules` will become "claimed" (removed

from this Map), when the `getPluginCapsulesForLoading(String type)` function is called. This function returns a Map with mappings the same as those in the other maps. Every unclaimed PluginCapsule is checked for the same type as requested. If it matches, it is inserted into the return Map. Once all capsules have been checked, any in the return Map are removed from the unclaimed map.

## Plugin Loading Algorithm

When `MapViewUI`, the main UI class, is instantiated it is given a reference to the `ViewController`. As far as the UI is concerned, the `ViewController` is the rest of the system that is relevant. Also, a `PluginLoader` is instantiated and the `load()` method is called on it. Eventually, the `createUI()` method is called, at which point a `JFrame` is created and populated. The population happens in the `buildUI()` method, which is separate so that if a window existed already, the UI could be built inside that, instead of a new `JFrame`. The process of creating and positioning all the buttons, sliders, and other components used to happen directly within `buildUI()`; resulting in a large mass of somewhat unorganized code. This was broken up into separate classes. The intermediate code loaded all of these manually. Upon integration of `PluginLoader`, all of this now happens in a small section, approximately 20 lines long. 10 or so lines of this code revolves around setting up the `JDockingPanel`'s which allow breaking the UI up into separate frames.

The code that happens in `MapViewUI` works as follows. The `PluginLoader`, which already did most of the necessary work in the `load()` method, is asked for a Map populated with `PluginCapsules` of a type "main". Using the correct method guarantees that these plugins will not be returned to any other code for loading, so they are safe to use. This guarantee is necessary, as each `PluginCapsule` hold only one instance of a plugin; Attempting to put a UI Component in two different places simultaneously is not a good idea. The values of the Map are pulled out and inserted into a List. The List is passed to `PluginLoader` to be sorted by the position. After sorting, the `PluginCapsules` return instances of Plugins. These Plugins are passed with their registration as arguments to a `JDockingPanel`, which itself is then loaded as a panel in the four-section UI. Type, position, and registration as mentioned above are fields in the plugin XML file. One can see that the work of loading plugins is mostly automated as far as `MapViewUI` is concerned.

## Movie Making

### What is the reason for making movies?

To display the simulation, Spiegel generates a series of frames. During generation, the rate at which the frames are displayed is limited by the speed of the computer and components to read the raw data and create the image. For every particle in the simulation, there are many pieces of data that need to be read in to display it. This must be done for every particle in the simulation, and often there is the step of interpolating this position from two or more others. The current maximum rate for showing a simulation directly from the raw data is around 5 frames per second (fps). This is rather slow, and makes it difficult to watch anything move. For movement to seem smooth, we need a frame rate at around 25 to 30 fps. One can see that to look smooth, our simulation needs to be much faster. Since Spiegel is targeted to run on the average available computer, we cannot

simply use a faster computer. We need a way to speed up the display of the frames, even though we cannot speed up the generation of frames.

The data needed to display each frame is much less than what is required to generate the frame. Once generated, only the image data is needed. Spiegel is currently capable of saving frames to JPEG and PNG file formats. The goal of this part of the project is to take generated frames and convert them to common movie formats. The current software is capable of converting JPEG images to the QuickTime movie format.

## Server Software

The server uses the Java Media Framework (JMF) to convert from JPEG images to QuickTime movies. This process involves setting up input and output streams, using `DataSources` and `DataSinks`, respectively. Multiple streams can be located in either, though the movie making discussed here uses only a single video stream. One might also find other types of streams in these Classes in another system, such as audio. These classes are part of JMF, and are used to interact with various sources and destinations for data using a uniform interface.

After several tries, I managed to create a client-server architecture that would start with jpegs on the client side, do the processing on the server, and return a movie. The first few tries, however, were unsuccessful. To begin with, my team in winter quarter started with a few ideas. One was to use remote method invocation to make it seem like the server was local. It eventually became clear that this was overkill, since there were only a few different tasks that had to be done over the network. A few pieces of data were needed to start the conversion, and then the rest was just file-transferring.

## Client/Server Model

The most important goal for the software dealing with the making of movies is to reduce the amount of time needed for the user to interact with the system. Previous methods of making a movie from generated images always involved the use of an outside program to do the conversion. Such programs needed to be installed, and the time taken to select the files was large, even though they were numbered. An average time of five to ten minutes just to tell the computer which files to use is unacceptable. The movie making process in this system is done on a remote machine to ensure the user is not required to install yet another piece of software to get the system to run; more importantly, that the client computer does not need the software if it may not be used. Therefore, all of the software used to do the actual conversion is located on a remote server; the software included in the system is only enough to interact with the server. Since it is included, the time to install another program is avoided.

One of the goals was to make the server only work in RAM, so that all data-access was fast and nothing was saved to the hard drive. Looking at the way the example file `JpegImagesToMovie.java` from `java.sun.com` behaved, by using this code, I created a stand-alone program that converted images to movies on the same computer. Abstracting this to two different machines was fairly easy, except for the part about the input and output files. Using a TCP network stream, I determined a protocol for the way necessary start-up information would be sent and received. Most of these were just numbers, so `ObjectOutputStream.writeInt(int)` and the

corresponding read were sufficient. The next problem to tackle was how to get the images across, and not have to save them to the server file system. Once I understood that the example file was using a class called `ImageDataStream` to convert the files into a byte stream, I mimicked it to pass a request for the file to the client, and then to read from the network stream. From there, I simply substituted my stream for the existing one where necessary. The rest of the existing code would continue to function in the same way, except for minor fixes to get all of the arguments to work correctly.

## Input (JPEGs)

The server requests a certain image by telling the client the sequence number needed. The client determines which file this is, sends the length of the file as a count of the bytes, and then sends the bytes representing the file. This is all done through a custom `DataSource` to route the data over the network. On the client side, an intermediate Class is used to handle the reading of the data from the file so that this code is not intermixed with the rest of the program. This Class converts the JPEG files into a stream that the `DataSource` can understand.

Without question, the hardest part of this system was to get the finished movie back to the client. When the conversion code determines the `MediaLocator` for the output file, it takes a URL as an argument. I had hoped to define my own URL protocol to communicate with a destination via another TCP stream, and then simply substitute it, but I discovered `MediaLocators` do not behave this way. A `MediaLocator` is something like a URL without the need for an `URLHandler` or the ability to do input or output. The code was taking the `MediaLocator` and using it to create a `DataSink`. Since the method being used (`Manager.createDataSink`) was using default methods, an object of type `FileDataSink` was being returned. Upon further investigation, I found that `FileDataSink` was a class included with JMF seemingly just for this purpose. The problem with using and adapting this file is that it has no Javadoc, very few comments, and is "Copyright (c) 1996-2002 Sun Microsystems, Inc. All rights reserved.". In the interests of getting a prototype working, I made the program save the file to the server file system, send it back to the client, and then delete it. This is regrettably slow, but the system was designed to be a batch job, not the fastest possible implementation. The system is already fairly slow for performance, only because transferring the files over the network is so much slower than from a hard drive to RAM on the same system.

## Conversion

Once the `DataSource` and `DataSink` are established, a "Processor" is instantiated, set up, and started. The Processor also handles the conversion between the type of input and the type of output. The Processor also uses variables acquired from the client program, specifically, the frame rate, which is requested from the user, and the size of the frames, which is determined automatically from the height and width of the first image. This Processor tells the Sources and Sinks what needs to be done, passively or actively depending upon their type; a `PushBufferData(Source/Sink)` is used actively and a `PullBufferData(Source/Sink)` is passive. Internally, there are multiple Sources and Sinks, representing which way the data is flowing, and in what form.

## Output

The final destination for the movie is the client's computer. This requires either sending output directly to the client's computer, or first to the server's local file system and then transferring it over the network to the client.

JMF comes with implementation to save an output to the local file system and for streaming over the network. Unfortunately, the implementation for streaming allows for errors, putting emphasis on streaming frames to get as much as possible over the line in a time sensitive manner. This means that if data is lost, or not transmitted fast enough, it can be skipped. Also, we want to be able to save the output to a file, and this would require another conversion to happen on the client's computer. Since we intend not to have the client dealing with anything other than raw data and files, this is an unacceptable solution. The project now uses the existing FileDataSink to save the file to the server's hard drive, and then transfers it byte by byte to the client.

Client-Server MovieMaking Timeline:

