The MX architecture and its usage within the Cinema-MX application

Phillip Lord

February 20, 2002

Abstract

This document describes the MX[1] architecture, and how it has been used in the Cinema-MX[2] application. Its meant both as an overview of the architecture, and also a introduction for those who might wish to develop new modules for the application.

1 Introduction

The MX architecture was designed to allow Cinema-MX (or other applications) to be extended straightforwardly and simply. This is achieved by splitting the application up into a series of small modules. These modules can then be put together to form the end application.

For the Cinema-MX application as well as allowing modularity we wanted to gain extensibility, allowing the user to integrate new modules into Cinema-MX, without requiring alteration of the main code base. To this end, XML has been used to define which modules should be loaded, and also to provide some configuration for these modules if required.

2 Requirements

There were a number of requirements for the MX architecture.

- The architecture should be relatively *light-weight*. That is it should be verbose, which would discourage programmers from making small modules.
- The basic architecture should be as customisable as possible.
- The architecture should be *fail fast*, that is it should break early, rather than run incorrectly.
- Dependency between modules should be explicit.
- Loading should be as rapid as possible.

[1] Modular, eXtensible
[2] Colour INteractive Editor for Multiple Alignments

3 Identifiers

The requirement that module dependency should be explicit creates a problem. One module must be able to refer to another. However if that class is referred to directly, then the class will be loaded immediately, when we might want to defer it.

To circumvent this problem, we use an identifier class, which can be used to refer to the module, the interface for this is shown in Listing 1.

```
public interface ModuleIdentifier
{
  public boolean isInterface();
  public String getClassName();
  public String getModuleName();
}// ModuleIdentifier
```

Listing 1: The ModuleIdentifier Interface

The two methods in this interface `public String getClassName()` and `public String getModuleName()` define the link between a module and a String which can be used to refer to it, which solves the problem of referring to a Module (by a name) and a Class.

One difficulty with this is that using a String (the module name) to refer to a module is not type safe, and will result in errors when the name is typed incorrectly. This difficulty is solved by extending the class `AbstractEnumeratedModuleIdentifier`. This uses the a variation of the theme of the Singleton design pattern to create an identifier class which is relatively type safe. An example of its usage from Cinema-MX is shown in Listing 2. This class uses reflection to translate the variable name of its instances, into the String that it uses for its Module name.

```
public class CinemaBootIdentifier extends AbstractEnumeratedModuleIdentifier
{
  private CinemaBootIdentifier( String className, String toString )
  {
    super( className, toString );
  }
  public static final CinemaBootIdentifier CINEMA_BOOT =
    new CinemaBootIdentifier( "uk.ac.man.bioinf.apps.cinema.CinemaBoot",
                      "Main Cinema Boot Class" );
  public static final CinemaBootIdentifier CINEMA_SHARED =
    new CinemaBootIdentifier( "uk.ac.man.bioinf.apps.cinema.CinemaShared",
                      "Shared Boot Class" );
} // CinemaBootIdentifier
```

Listing 2: The Abstract EnumeratedModuleIdentifier

The practical upshot of all of this is that to refer to for instance the main Cinema Boot Module, the CINEMA_BOOT instance can be used directly. Its still possible to type this incorrectly of course, but this will be detected at compile time. Its also possible to type the class name incorrectly when writing the identifier, but at least this needs to only be done once.

3.1 Abstract Identifiers

There is a final method in the `ModuleIdentifier` class, called `isInteface`. Normally this will return false, but it is possible to define a module which acts like an abstract class, and delegates its functionality to another module. This allows a degree of polymorphism for modules. This feature is not used very widely within Cinema-MX, as it turned out to be less needed than it appeared to be during design. For most purposes its easier to use “Optional Modules” which are described in Section 4.

4 The Module

The `ModuleIdentifier`interface refers to a Class name. This Class should be a instance of the `Module` class. Its interface is shown in Listing 3. There are quite a few other methods in this class, but most of them have been elided here for the sake of simplicity.

```
public abstract class Module
{
  public void load() throws ModuleException {};
  public void start();
  public void destroy();
  public ModuleIdentifierList getRequiredIdentifiers();
  public Module getRequiredModule( ModuleIdentifier ident );
  public abstract String getVersion();
} // Module
```

Listing 3: The Module class

We can divide the methods shown here into three groups.

- Those related to dependency with other modules.
- Those directly to do with the function of the Module.
- And the other method.

Dealing with this in order. One of the requirements is for explicit dependency between modules. This is provided by the `public ModuleIdentifierList getRequiredIdentifiers()` method. In this method any modules which this module depends on should be identified. For example, see Listing 4, which comes from the `CinemaConsensusDisplay` module. This requires two other modules, namely `CinemaConsensus`, which actually takes on the task of calculating the consensus, and `CinemaSystemEvents`. The consensus display is threaded, and needs to know when the application is about to close, so that it can shut down cleanly.

```
public ModuleIdentifierList getRequiredIdentifiers()
{
  ModuleIdentifierList list = super.getRequiredIdentifiers();
  list.add( CinemaConsensusIdentifier.CINEMA_CONSENSUS );
  list.add( CinemaCoreIdentifier.CINEMA_SYSTEM_EVENTS );
  return list;
}
```

Listing 4: An example of getRequiredIdentifiers

The second method `public Module getRequiredModule(ModuleIdentifier ident)` actually allows access to these modules. Listing 5 comes again from the `CinemaConsensusDisplay` class

```
if( queue == null ){
  queue = new InvokerInternalQueue
    ( (CinemaSystemEvents)getRequiredModule
       ( CinemaCoreIdentifier.CINEMA_SYSTEM_EVENTS ) );
}
```

Listing 5: An example of getRequiredModule

The methods dealing with module functionality are hopefully largely self-explanatory. When the module is initially loaded, unsurprising the `public void load` method is called. During this time the module should perform any initialisation that is required. The rule at this time is that only initialisation that does not require other modules should be performed, as this may well not be available yet. Or in another way, while the `load()` method is running, there are no guarantee's about what the `getRequiredModule()` method will return (most likely it will return `null`).

Immediately after this time the `public void start()` method will be called. At this time it is guaranteed that the `getRequiredModule()` method will return any of the Modules identified, and that further all of their `load()` methods will have been called and have successfully completed.

This is actually simpler than it sounds, but it's designed to cope with a fairly complex dependency graph, and generally it just works. No checking is performed to ensure that the graph is acyclic The system will crash if you do this, but as per the design requirement it will fail immediately.

And finally the other method. This is meant to return a String identifying the version of the Module. This is not widely used. No specific semantics is required for this String, and generally the CVS version keyword has been used. This might be removed at a later date.

4.1 Other methods

There are a few other methods which are potentially of interest within the `Module` interface. Firstly the `Module` provides access to the `ModuleContext` class, which contains the `public Module getModule(ModuleIdentifier ident)`. This enables access to any other Modules in the system, beyond those named as required modules. As they are not required they may be unavailable, so checking the `public boolean isModuleAvailable(ModuleIdentifier identifier)` first is probably wise.

And finally the `ModuleContext` class gives access to the `public Object getConfig()` method. Of itself this is not that useful. Its used internally to provide XML configuration though, which is described in Section 5.2.

5 XML Loading and Configuration

The module system described so far provides a basic architecture. However some mechanism needs to be available to define which modules should be used. While it is possible to do this using Java directly this would require the user to possess a Java compiler to enable new modules, or reconfigure existing ones. By defining the loading and configuration in XML, it's possible to do this using a text editor.

5.1 Loading

As described in Section 3.1, each module referred to by a `ModuleIdentifier`. In order to load first the ModuleIdentifier must be made available to the system. The `Identifier` directive can be used to this end, as shown in Listing 6. This code assumes that the `AbstractEnumeratedModuleIdentifier` has been used. For further information see the `module.dtd` file in the source, which is heavily documented. Its worth remembering that the `AbstractEnumeratedModuleIdentifier` can contain identifiers for many different modules, so relatively few of these statements are needed. In Cinema-MX the modules are grouped into functional units. The overhead of loading an `ModuleIdentifier` is very low (a few objects for each additional one), so there is not really any problem in loading these, even if the module is not used in the end.

```
<identifier>
  <enumeration>
    <class>uk.ac.man.bioinf.apps.cinema.utils.CinemaUtilityIdentifier</class>
  </enumeration>
</identifier>
```

Listing 6: The Identifier Directive

Having made the `ModuleIdentifier` available, the module itself can be loaded or started from within the XML, using the `load` and `start` directive. The module is referred to by the name returned by the identifier. For example, the code in 7 shows loading and starting of the module that provides the "status bar" in Cinema-MX.

```
<load>
  <name>CINEMA_STATUS</name>
</load>
<start>
  <name>CINEMA_STATUS</name>
</start>
```

Listing 7: Loading a Module

5.2 Configuration

As well as loading modules its also possible to configure them. At the current time, the configuration can come in one of two forms, which are a properties list, or a tree structure. For example in Listing 8 the configuration which is used for input module, is shown. It defines "parsers" which are used to output sequence data. This configurability means that it's to possible to add new "parsers" (perhaps inappropriately named as they are used for both input and output of sequences) Cinema-MX, by altering the configuration for this module.

```
<properties>
  <paramname>PIR</paramname>
    <value>uk.ac.man.bioinf.io.parsers.PIRProteinAlignmentParser</value>
  </param>
  <param>
    <paramname>MOT</paramname>
    <value>uk.ac.man.bioinf.io.parsers.MotProteinParser</value>
  </param>
</properties>
```

Listing 8: Configuring Parsers

The second type of configuration is a simple tree structure, which can be seen in Listing 9. In this case the menu system is being configured. In this case most of the configuration has been elided. Generally speaking the properties configuration is to be preferred because its much simpler, but the tree structure is much more versatile, and means that tricks, such as providing additional semantics to the keys of the properties lists are not necessary.

```
<tree>
  <!-- The File Menu -->
  <node>
    <value>File</value>
    <node>
      <!-- Provides the open alignment -->
      <name>SEQ_INPUT</name>
    </node>
    <node>
      <!-- Provides the save alignment -->
      <name>SEQ_OUTPUT</name>
    </node>
    <node>
      <!-- Provides the exit menu -->
      <name>CINEMA_CORE_GUI</name>
    </node>
  </node>
</tree>
```

Listing 9: Configuring the menu system

In order for the modules to access this configuration two methods (see Listing 10) are provided by the `XMLModule` class which all the Cinema-MX modules extend from. The standard Java `Properties` class has been used here. Sadly Java does not provide a standard Tree class, so a simple one has been provided.

```
public Properties getConfigProperties();
public ConfigNode getConfigTree();
```

Listing 10: The XML methods

6 Cinema-MX Modules

Although the MX architecture provides the ability to define modules, and their interaction with each other, they do not provide any specific Cinema functionality. It would be possible to provide all of this functionality through the MX system, by accessing specific modules. However for convenience the `XMLModule` class has been extended, to give access to a number of different method, through the `CinemaModule` class. [3] Additionally there is a more specific `CinemaCoreGui` class, which gives access to more methods, which gives direct access to the widgets used to build the basic Cinema-MX frame. Essentially the rule is extend the `CinemaModule` unless you really need the `CinemaGuiModule` as the latter is less likely to remain stable.

```
public abstract class CinemaModule extends XMLModule
   implements AlignmentEventProvider
{
  public SequenceAlignment getSequenceAlignment()
  public void setSequenceAlignment( SequenceAlignment seq )
  public ColorMap getColorMap()
  public void setColorMap( ColorMap map )
  public AlignmentSelectionModel getAlignmentSelectionModel()
  public void setAlignmentSelectionModel( AlignmentSelectionModel model )
  public void setSequenceTitleColor( GappedSequence seq, Color colour )
  public void clearSequenceTitleColor( GappedSequence seq )
  public void sendStatusMessage( String message )
} // CinemaModule
```

Listing 11: The Cinema Module

The interface of the `CinemaModule` is shown in Listing 11. Direct access is provided to the alignment being shown, to the `ColorMap`. [4] The other methods give access to other information associated with the view, including the Selection Model, the colour associated with the sequence, and a status message which appears at the bottom of the Cinema-MX frame.

6.1 Cinema-MX modules in use

The MX architecture works best if the individual `Module`'s are relatively small. To give some idea of how this works in Cinema-MX, the `Module`'s are described in Table 6.1.

[3] As it happens, this has been implemented by accessing modules through the MX architecture. All of the functionality provide by the `CinemaModule` is actually delegated to a `XMLModule` called `CinemaCoreView`, while all of the functionality provide by the more specific `CinemaGuiModule` is serviced by the `CinemaCoreGui` module.

[4] In the interests of international co-operation, it should be noted that the shorter spelling of the word "Color" was used. In the interests of flag waving jingoism, it should be noted that it hurt, it really hurt.

CinemaModule	Function
CinemaColorFactory	Generates `ColorMap` instances, and menu items for their selection.
CinemaCommand-LineParser	Parses the command line, and acts on it.
CinemaConsensus	Provides calculation of consensus sequences
CinemaGroupModule	Group sequences, for editing, viewing, and analysis.
CinemaMenuBuilder	Generate menu items on the basis of the XML configurations
CinemaMotifModule	The MotifManager dialog, and output
CinemaMultiple-ConsensusViewer	View consensus sequences of groups, and there variance.
CinemaPersist	Save information between instantiations of Cinema-MX
CinemaRegexp	Regular Expression searches down sequences
CinemaResizeElements	Resize sequence cells
CinemaSlaveViewerModule	Generate viewer frames for use by other modules.
CinemaSplash	Adds massive functionality in the form of a Splash screen.
FormGroupsByPrints	**Experimental!** Queries the PRINTS_S database to display the PRINTS motifs.
PhylipInvoker	**Experimental!** Displays a phylogenetic tree.
CinemaGuiModule	
AbstractSequenceInput	Input sequence by some route.
AbstractSequenceOutput	Output sequence by some route.
CinemaColorSelector	Select `ColorMap`.
CinemaConsensus-Display	Displays the consensus sequences.
CinemaCoreView	Support for CinemaModule class
CinemaGo	Well everybody hacks some times
CinemaHackMenu	More professionally displayed as “in development” in the menu system. Its quicker to add here than through XML.
CineamMenuSystem	Uses CinemaMenuBuilder to build menu, then displays it
CinemaSequenceMenu	Right click on sequence button menu.
CinemaStatusInformation	Prints “cursor here” information in status bar.
Others	
CinemaCoreGui	Extends directly from the `Module` class, and provides support for the CinemaGuiModule class.
XMLBootModule	Extends from directly from `Module`. Loads1 parsers for reading XML configuration.
CinemaBoot	Extends from `XMLBootModule`. Additional support for Cinema-MX XML loading.
CinemaFilePersist	Extends from `CinemaPersist`. Save persistence data to file.
FileSequenceInput	Extends from `AbstractSequenceInput`. Load from file.
FileSequenceOutput	Extends from `AbstractSequenceOutput`. Save to file.

Table 2: The modules in use within the Cinema-MX application. The modules are organized by their super class, either the `CinemaModule`, the `CinemaGuiModule`, or `Others`, which are extended from some other module.