CAPLIN

Caplin Xaqua 1.0

Customizing The XMC

December 2009

CONFIDENTIAL

Contents

1 Preface

1.1 What this document contains

This document describes how to customize the Caplin Xaqua Management Console (XMC).

About Caplin document formats

This document is supplied in three formats:

- Portable document format (*.PDF* file), which you can read on-line using a suitable PDF reader such as Adobe Reader®. This version of the document is formatted as a printable manual; you can print it from the PDF reader.
- Web pages (*.HTML* files), which you can read on-line using a web browser. To read the web version of the document navigate to the *HTMLDoc_m_n* folder and open the file *index.html*.
- Microsoft HTML Help (*.CHM* file), which is an HTML format contained in a single file.
  To read a *.CHM* file just open it – no web browser is needed.

For the best reading experience

On the machine where your browser or PDF reader runs, install the following Microsoft Windows® fonts: Arial, Courier New, Times New Roman, Tahoma. You must have a suitable Microsoft license to use these fonts.

Restrictions on viewing .CHM files

You can only read *.CHM* files from Microsoft Windows.

Microsoft Windows security restrictions may prevent you from viewing the content of *.CHM* files that are located on network drives. To fix this either copy the file to a local hard drive on your PC (for example the Desktop), or ask your System Administrator to grant access to the file across the network. For more information see the Microsoft knowledge base article at
http://support.microsoft.com/kb/896054/.

1.2 Who should read this document

This document is intended for people who want to customize the XMC to monitor components of Caplin Xaqua.

1.3 Related documents

- **Caplin Xaqua: Getting Started With The XMC**

  Describes how to configure the Caplin Xaqua Management Console.
- **Caplin Xaqua: Monitoring And Management Overview**

  Describes the Caplin Xaqua Management and Monitoring solution and its place in the Caplin Xaqua architecture.
- **Caplin Xaqua Management Console: API Reference**

  The API reference documentation provided with the Caplin Xaqua Management Console.
- **Caplin Xaqua: Monitoring Socket Interface Specification**

  Describes the commands and responses of the Monitoring Socket Interface.
- **Caplin Liberator: Administration Guide**

  Describes how to install and configure Caplin Liberator.

1.4 Typographical conventions

The following typographical conventions are used to identify particular elements within the text.

Type	***Uses***
aMethod	Function or method name
aParameter	Parameter or variable name
/AFolder/Afile.txt	File names, folders and directories
`Some code;`	Program output and code examples
The `value=10` attribute is...	Code fragment in line with normal text
Some text in a dialog box	Dialog box output
`Something typed in`	User input – things you type at the computer keyboard
XYZ Product Overview	Document name
◆	Information bullet point
■	Action bullet point – an action you should perform

Note: Important Notes are enclosed within a box like this.
Please pay particular attention to these points to ensure proper configuration and operation of the solution.

Tip: Useful information is enclosed within a box like this.
Use these points to find out where to get more help on a topic.

1.5 Feedback

Customer feedback can only improve the quality of our product documentation, and we would welcome any comments, criticisms or suggestions you may have regarding this document.

Please email your feedback to documentation@caplin.com.

1.6 Acknowledgments

Adobe® Reader is a registered trademark of Adobe Systems Incorporated in the United States and/or other countries.

Windows is a registered trademark of Microsoft Corporation in the United States and other countries.

Java and *JMX* are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. or other countries.

2 Overview

The Caplin Xaqua Management Console (XMC) has been designed to allow customization of its displays and in particular the inclusion of user defined views. Configuration of the console is defined via a set of XML files. By implementing certain interfaces and writing a small amount of XML, a user can install their own views into the console.

User views are hosted as tabbed pages within the console, each tab can have its own name, icon, tooltip, menu options, toolbar options, and help text. Views are loaded dynamically and on demand for performance reasons. To add a new view to the console the user must perform the following actions (described in more detail in the rest of this document):

1. Create a `JPanel` derived class that implements the `View` Interface:

```
void init(Console console, Properties properties)
void save()
Properties getProperties()
List getActions()
Console getConsole()
void processMessage(String messageId, Map messageData)
```

This view should use the `Console` object passed into its `init()` method to obtain a `JMXConnection`. The actions (Swing Action objects) exposed via the `getActions()` method can be used to populate menu and toolbar options. Properties that configure the view can be loaded and saved via the `init()` and `getProperties()` methods.

2. Create an XML file named *view_<NameOfView>.xml* that defines the view, its classname, tabname, icon, menus, toolbar, help topic, and so on.

3. Edit one or more of the *console_<datasourceType>.xml* files to reference this view; that is, specify that a console of the given type should load and display this view.

4. Link in the help text.

3 Implementing Required Java Classes

There is one interface that must be implemented in order to create a new view. There is also an optional interface which is described below. The required interface is `com.caplin.view.View`.

More details on this interface can be found by looking at the **Caplin Xaqua Management Console: API Reference**, supplied with the XMC. The interface comprises the following methods:

- `void init(Console console, Properties properties);`
- `void save();`
- `Properties getProperties();`
- `List getActions();`
- `Console getConsole();`
- `void processMessage(String messageId, Map messageData);`

The class that implements this interface should extend a suitable `JComponent` class such as `javax.swing.JPanel`. So for example, the declaration of a class that implements a view for a component would have the following declaration:

```
public class ExampleView extends JPanel implements View
```

This class should be specified in the *view_<NameOfView>.xml* file (see Creating a Help Guide 12 for further information). In this file, the developer may specify a number of actions to be performed. Each of these actions need to be implemented in the Java code (see Actions 6).

3.1 Actions

Each `<Action Name="..."/>` tag in the *view_<NameOfView>.xml* file needs to correspond to an action in the Java code. This should be achieved by constructing an inner class within the class that implements `com.caplin.view.View`. This inner class should extend `javax.swing.AbstractAction`.

So for example, if a developer creates a *view_example.xml* file with the action tag `<Action Name="SayHello"/>`, then the Java code in the inner class could be similar to the following:

```
class RefreshAction extends AbstractAction
{
  public RefreshAction()
  {
    this.putValue(Action.ACTION_COMMAND_KEY, "SayHello");
    this.putValue(Action.NAME, "Say Hi");
    this.putValue(Action.SHORT_DESCRIPTION,"Say hi to the user");
    this.putValue(Action.MNEMONIC_KEY,
                  new Integer(java.awt.event.KeyEvent.VK_F5));
    this.putValue(Action.SMALL_ICON,
                  ResourceManager.getInstance().getImage(
                  "resources/images/Refresh16.gif"));
  }
  public void actionPerformed( ActionEvent e )
  {
    //perform processing to achieve goals of action
  }
}
```

The first line of the constructor tells the XMC to call this class to perform the action described in the `<Action .../>` tag. Notice that the two strings are the same. This is very important as otherwise the XMC will not be able to tell what class should be used to perform the task, and will output an error to the command prompt and act as if the user never asked for the action to be performed.

The second line tells the XMC what text to display on the drop down menu.

The third line tells the XMC what text to display in the tooltip for the action. The tooltip will be displayed for both a drop down menu option and a toolbar button.

The fourth line specifies a keyboard shortcut key to use for this action. The XMC will listen for this keystroke and will call the class when it receives it.

The fifth line specifies an icon to be used for this action. If this action is to be added to the toolbar then the icon will act as a button. If the action is to be added to a drop down menu, then the icon will be displayed next to the text as specified in the second line.

The code in `actionPerformed(ActionEvent e)` performs the action. The XMC calls this method when the user requests the action corresponding to this class (by selecting a menu item or clicking a toolbar button).

3.2 Implementing ConsoleListener

The interface `com.caplin.console.ConsoleListener` should be implemented if the view needs to perform processing when the XMC starts up or shuts down. The interface should also be implemented if the view needs to be notified when another view is updated or the JMX connection changes.

A detailed description of each method to be implemented is available in the JavaDoc which is supplied as standard with the XMC. For completeness the method declarations are also supplied here, and are as follows:

- `void viewChanged(View view);`
- `void viewLoaded(View view);`
- `void connectionStateChanged(boolean connected);`
- `void closing();`

3.3 Inter-view communication

Since each view class is loaded dynamically and on demand (via the class name in the associated *view_<NameOfView>.xml* file), you should not directly access one view from another.

Inter-view communication is handled via the following three methods, two on the `Console` class and one on the `View` class.

Console Class: showView()

```
/**
* Shows the given view, the view must be defined in the console
* XML configuration file
* @param toViewClassName
* @return true if view shown
*/
boolean showView(String toViewClassName);
```

Console Class: postMessage()

```
/**
* Posts a message to the given view, used for inter-view communication
* @param toViewClassName the class name of the view to receive the message
* @param messageId a String identifying the message
* @param messageData a Map of name, value pairs representing messages data
*/
void postMessage(String toViewClassName, String messageId, Map messageData);
```

View Class: processMessage()

```
/**
* Processes a message from another view, used for inter-view
communication
* @param messageId a String identifying the message
* @param messageData a Map of name, value pairs representing messages data
 */
void processMessage(String messageId, Map messageData);
```

This mechanism is used throughout the standard views supplied with the XMC, to link items to the associated MBean tab in the Explorer view. The following code snippet shows how to perform this link from a `View` class:

```
protected void onShowInExplorer( ObjectName objName )
{
  getConsole().showView("com.caplin.view.explorer.ExplorerView");
  HashMap dataMap = new HashMap();
  dataMap.put("MBeanName", objName);
  getConsole().postMessage("com.caplin.view.explorer.ExplorerView",
  "ShowMBean", dataMap);
}
```

4 Writing the Required XML

If a custom view is to be created for a particular component (for example Liberator), then a developer will need to complete the tasks described in View XML 9 and Console XML 11 in order to get the XMC to use the custom view.

4.1 View XML

A new file called *view_<NameOfView>.xml* will need to be created. This file defines what a view will look like. The file takes the following form.

Example Code

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE View SYSTEM "dtd/View.dtd">
<View>
  <Class>com.company.view.OverviewViewTab</Class>

  <Name>Overview</Name>
  <LongDescription>This tab gives an overview of the new component...
  </LongDescription>

  <HelpTopic>overview.mainpage</HelpTopic>

  <Icon>resources/myIcon.gif</Icon>

  <Menus>
    <Menu Name="Tools">
      <Action Name="PerformAction" />
    </Menu>
  </Menus>

  <ToolBar>
    <Action Name="PerformAction" />
    <Divider/>
    <Action Name="PerformOtherAction" />
  </ToolBar>
</View>
```

XML Tag Descriptions

`<Class>`

The `<Class>` tag defines the Java class that will be used to display this view (see Implementing Required Java Classes 5). The XMC calls this class when a user double-clicks the component in the main page of the XMC. If it cannot find/load the class, then it will output an error message to the command prompt and fail to launch the component (as if the user had performed no action).

`<Name>`

The `<Name>` and `<LongDescription>` tags are used by the XMC to display information about this view to the user. The text entered between the opening and closing `<Name>` tags is displayed in the tab for this view.

`<LongDescription>`

If the user hovers over the tab, then the text between the opening and closing `<LongDescription>` tags is displayed as a tooltip. This gives the user more information about the view without having to open it.

`<HelpTopic>`

The `<HelpTopic>` tag is used by the XMC to reference the appropriate section in the help guide for this view. A developer can easily add help for a custom component to the XMC. This is covered in more detail in Creating a Help Guide 12.

`<Icon>`

The `<Icon>` tag is used to specify the location of an image file that the XMC displays next to the name of the view (see the `<Name>` tag). This file can be a relative path or an absolute path. The base directory will be the directory in which the XMC was started.

`<Menus> and <Menu>`

The `<Menus>` tag lists the menus that are displayed by the XMC. Each menu is defined by a `<Menu>` tag and each menu option by an `<Action />` tag (see Actions 6). In the example above, the XMC will create a drop down menu called Tools. The display name for each menu option is defined by Java code (see Implementing Required Java Classes 5).

`<Divider>`

It is possible to have dividers in drop down menus and toolbars. For example, if you want to separate out different types of operation in a single menu, then you could add a `<Divider />` tag between two `<Action />` tags. This causes the XMC to insert a horizontal line between the two items in the drop down menu. In the code example above, a `<Divider />` tag is inserted in a `<Toolbar>` section to separate two toolbar buttons.

`<Toolbar>`

The `<ToolBar>` tag is used by the XMC to display buttons on the toolbar. The developer also needs to implement this action in the class specified for this view (see Implementing Required Java Classes 5). When the user clicks a button the XMC calls the relevant Java code (see Actions 6).

When a `<Divider />` tag is used to separate buttons from each other, the XMC inserts a small vertical line between the buttons.

4.2 Console XML

First the developer needs to go to the correct XML file for the component. This file will be called *console_<ComponentName>.xml* (for example, *console_liberator.xml*). This file tells the XMC what views to create for the component. The format of the file will be similar to the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ConsoleConfig SYSTEM "dtd/ConsoleConfig.dtd">

<ConsoleConfig>
  ...

  <Views>
    <View Id="ExplorerView">
      <Properties>
        <Property Name="SplitPos" Value="...."/>
        <Property Name="SortOrder" Value="...."/>
      </Properties>
    </View>
    <View Id="OverviewView"/>
    <View Id="PermissionView"/>
    <View Id="PageEditorView"/>
  </Views>
</ConsoleConfig>
```

For every view defined in the `<Views>` section, an accompanying file needs to be created (see View XML 9). The developer should also add any required properties for the new view in the `<Properties>` section of the `<View>` tag. The XMC will create a `java.util.Properties` object from the `<Properties>` section, and pass the object to each view (see Implementing Required Java Classes 5 for further information).

5 Creating a Help Guide

As a final but important section on creating a customized view, it is important to create a help guide for the new view, that users can refer to in order to learn how to use the view or if they encounter problems. The XMC uses JavaHelp (http://java.sun.com/products/javahelp/) to provide an interactive help guide. This guide does not go into the details of how to create the various files required, instead it describes how to modify the help files that are distributed with the XMC to reference the help files that a developer needs to create.

In the *view_<NameOfView>.xml* file the developer needs to specify a help section for the view. The XMC will then use this to provide help to the user on the current view they are using.

The developer will also need to add entries to the following files:

- *resources/help/ParentHelpMap.jhm*
- *resources/help/ParentHelp.hs*

5.1 ParentHelpMap.jhm

In this file, the developer will need to add a link to the HelpSet file (*file.hs*) that should be created for the custom view.

The file will look similar to the following:

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE map
  PUBLIC "-//Sun Microsystems Inc.//DTD JavaHelp Map Version 1.0//EN"
    "http://java.sun.com/products/javahelp/map_1_0.dtd">

<map version="1.0">
  <mapID target="toplevelfolder" url="images/toplevel.gif" />
  <mapID target="pricemaster" url="PM4_help/IdeHelp.hs" />
  <mapID target="explorer.tab" url="html/explorer.html" />
</map>
```

The developer needs to add another `<mapID ... />` tag to point to the new HelpSet file created for the custom view.

5.2 ParentHelp.hs

This file is quite large but the developer only needs to edit one section within the file.

Under the `<view>` section there are <`subhelpset>` tags; the developer needs to add the help set file that has been created for the custom view to this section. For example:

```
...

<subhelpset location="CMCHelp.hs" />
<subhelpset location="PM4_help/IdeHelp.hs" />
<subhelpset location="customView_help/IdeHelp.hs" />

...
```

The location of the help set file should be specified if it is not in the same directory as *ParentHelp.hs*. An example of this can be seen above. The help set file *IdeHelp.hs* is located in a directory under the current one called *PM4_help*.

5.3 CustomHelpTOC.xml

In the Table of Contents for the custom view, two `<tocitem>` tags should be present; one for an introduction section to the custom view, and one providing the actual help for the custom view. An example of this can be seen in the file *CMCHelpToc.xml*. In this file the table of contents items are as follows:

```
<tocitem text="Introduction">
  <tocitem text="Management Console" target="top"/>
</tocitem>

<tocitem text="Tabs" target="tab.folder">
</tocitem>
```

The target values are defined in *CMCMap.jhm*.

The developer should not need to make any further changes to provide help for the custom view.

6 File Locations

Java class files should be rooted in the same directory as the *xmc.jar* file (that is, if your class is `com.acme.MyView`, then you should place the class file in the directory *com/acme* under the *CaplinXaquaManagementConsole* directory).

XML configuration files go in the *conf* directory and help files should be rooted in the *resources/help* directory (which must be created under the *CaplinXaquaManagementConsole* directory).

7 Code Examples

7.1 Example View class

The following code example implements the `View` class.

```
package com.caplin.view;

import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.ArrayList;
import java.util.Enumeration;
import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.logging.Logger;

import javax.swing.AbstractAction;
import javax.swing.Action;
import javax.swing.JButton;
import javax.swing.JPanel;

import com.caplin.console.Console;
import com.caplin.console.ConsoleListener;

/**
 * A simple Test class that implements View:
 *
 * <pre>
 *
 *  - logs the calls to View methods
 *  - logs property values passed in
 *  - uses property to load/save background colour
 *  - exposes action to set background color
 *  - listens and logs console events
 *
 * </pre>
 *
 */

public class TestView extends JPanel implements View
{
  private static Logger log = Logger.getLogger(TestView.class.getName());

  private Console console;
  private List actions;
  private Color backColor;

  public TestView()
  {
    backColor = Color.WHITE; // set default value

    // load list of actions to be exposed
    actions = new ArrayList();
    actions.add(new SetBlueAction());
    // put something on panel
    JButton b = new JButton("make it white!");
    add(b);
    b.addActionListener(new ActionListener()
    {
      public void actionPerformed( ActionEvent e )
      {
        backColor = Color.white;
        setBackground(backColor);
        console.setDirty(TestView.this, false);
      }
    });
  }

  /**
   * Called by console to initialize this view
   */
  public void init( Console console, Properties properties )
  {
    this.console = console;
    log.info("---");

    // add delay to simulate time taken to load state from server
    delay();

    // try to read color property
    String colorProperty = (String)properties.get("Color");
    if (colorProperty != null)
    {
      if (colorProperty.equals("White"))
      {
        backColor = Color.WHITE;
      }
      else if (colorProperty.equals("Blue"))
      {
        backColor = Color.BLUE;
      }
    }

    setBackground(backColor);

    // listen and log console events
    console.addConsoleListener(new ConsoleListener()
    {
      public void viewChanged( View view )
      {
        log.info("viewChanged");
      }

      public void viewLoaded( View view )
      {
        log.info("viewLoaded");
      }

      public void closing()
      {
        log.info("closing");
      }
      public void connectionStateChanged( boolean connected )
      {
        log.info("connection changed");
      }
    });
    // log all properties passed in
    Enumeration en = properties.keys();
    while (en.hasMoreElements())
    {
      String name = (String)en.nextElement();
      String value = (String)properties.get(name);
      log.fine(name + "= " + value);
    }

  }

  /**
   * Called by console to save this view (that is, save data to
   * persistent storage) if it has set its state to dirty
   */
  public void save()
  {
    log.info("---");
    delay();
  }

  /**
   * Helper for 1 second delay
   */

  private void delay()
  {
    try
    {
      Thread.sleep(1000);
    }
    catch (InterruptedException e1)
    {
      e1.printStackTrace();
    }
  }

  /**
   * Called by console to retrieve the set of properties that must be saved
   * in the xml for this view
   */
  public Properties getProperties()
  {
    log.info("---");

    // save background color as property
    Properties properties = new Properties();

    if (backColor == Color.WHITE)
    {
      properties.put("Color", "White");
    }
    else if (backColor == Color.BLUE)
    {
      properties.put("Color", "Blue");
    }

    return properties;
  }
  /**
   * Called by console to retrieve the list of actions exposed by this view.
   * The action names of these actions are used, via the view xml file, to
   * populate menu and toolbar options
   */
  public List getActions()
  {
    log.info("---");
    return actions;
  }

  /**
   * convenience method for passing console instance to other classes
   */
  public Console getConsole()
  {
    return console;
  }

  /**
   * Called by the console to send messages from other views,
   * not implemented here
   */
  public void processMessage( String messageId, Map messageData )
  {
  }

  /**
   * An example action to set the background color to blue
   *
   */
  class SetBlueAction extends AbstractAction
  {
    public SetBlueAction()
    {
      this.putValue(Action.ACTION_COMMAND_KEY, "SetBlue");
      // this is the action name used in the view xml to
      // populate menu and toolbar options
      this.putValue(Action.NAME, "Set Blue");
      // menu item name
      this.putValue(Action.SHORT_DESCRIPTION, "Set the background blue");
      // tooltip
      this.putValue(Action.MNEMONIC_KEY,
                   new Integer(java.awt.event.KeyEvent.VK_B));
    }

    public void actionPerformed( ActionEvent e )
    {
      backColor = Color.BLUE;
      TestView.this.setBackground(backColor);
      console.setDirty(TestView.this, true);
    }
  }

}
```

7.2 Example View XML configuration file

The following is an example of a View XML configuration file.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE View SYSTEM "dtd/View.dtd">

<View>
  <Class>com.caplin.view.TestView</Class>
  <Name>A Test View</Name>
  <LongDescription>This is a test view</LongDescription>
  <HelpTopic>testview</HelpTopic>
  <Icon>/conf/folder_view.png</Icon>

  <Menus>
    <Menu Name="Edit">
      <Action Name="SetBlue"/>
    </Menu>
  </Menus>

  <ToolBar>
    <Action Name="SetBlue"/>
  </ToolBar>
</View>
```

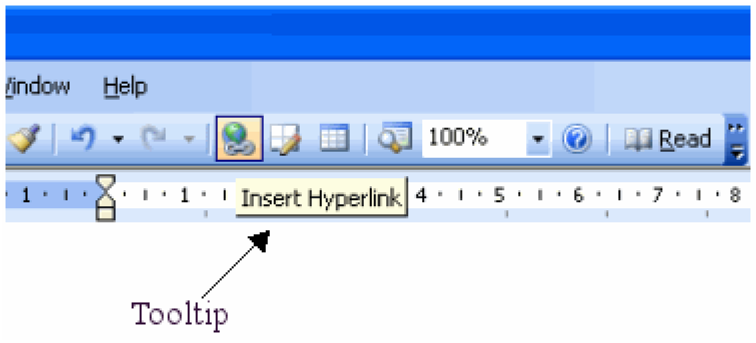
7.3 Example Console XML configuration file

The following is an example of a Console XML configuration file with test view added.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ConsoleConfig SYSTEM "dtd/ConsoleConfig.dtd">
<ConsoleConfig>
  <Properties>
  </Properties>
  <Views>
    <View Id="datasource_overview"/>
    <View Id="datasource_peers"/>
    <View Id="datasource_logging"/>
    <View Id="liberator_users"/>
    <View Id="liberator_objects"/>
    <View Id="explorer"/>
    <View Id="test"/>
  </Views>
</ConsoleConfig>
```

8 Glossary

This section contains a glossary of terms, abbreviations, and acronyms relating to the XMC.

Term	Definition
Tooltip	This is a common feature used in Java to provide extra information without cluttering up the user interface. If a user moves a mouse pointer over a component that has a tooltip, and lets the pointer rest over the component for a second or two, then a text description of the component appears at the end of the mouse pointer. This is called a tooltip. 
View	A view is defined as a panel displayed in the XMC for an individual component. A view is implemented as a tab by default (for example, the Explorer view is implemented as a tab).
XMC	Caplin Xaqua Management Console

Single-dealer platforms for the capital markets

CAPLIN

Contact Us

Caplin Systems Ltd

Triton Court

14 Finsbury Square

London EC2A 1BR

Telephone: +44 20 7826 9600

Fax: +44 20 7826 9610

www.caplin.com