Java™ Application Framework Reference Guide

Version 2.4.2

Fermi National Accelerator Laboratory
Beam Division
Accelerator Controls Department

P.O.Box 500,
Batavia, IL 60510-0500

April 29, 2003

Andrey Petrov
apetrov@fnal.gov

Table Of Contents

Abstract

A framework is a reusable, semi-complete application that can be specialized to produce custom application.

Ralph E. Johnson, Brian Foote.
Journal of Object-Oriented Programming, 1998

This document describes Java™ Application Framework—a library of reusable components, meant to support the development of user applications in Accelerator Controls Department.

The main purpose of Application Framework is providing classes, that incapsulate several frequently used, complex, and critical routines, such as creation of generic graphical user interface (GUI), printing, sending e-mail, user authorization, access to controls data and global application properties, logging, debugging aids, etc. The second purpose is the development of a corporate application environment, that allows easily to write, debug, deploy and launch Java applications. In this connection, each user application has to implement several functions to be integrated in such an environment.

Manpower of the Controls Department is limited. In some time a significant amount of people may be involved in development of Java applications, and there is a need to relieve them from rewriting of tons of code. We do not have an intend to limit programmers in their work by the Framework, and this project is not comprehensive and free of bugs. Application Framework is just an attempt to begin the development of a corporate environment for Java applications. Our final target is providing the library of reusable classes, which generalize the experience of many people; support these classes and develop required documentation.

1

Overview

1.1 Application Framework Functions

Current version of Java Application Framework provides the following:

1. `JFrame` extension with generic graphical user interface:
   a. dynamic GUI generation based on heritable XML descriptors;
   b. fast setting of several `JFrame` attributes from XML descriptors (title, icon, startup modes etc.);
   c. convenient configuration for Java Help;
   d. saving and restoring frame properties (size, location, etc.) between sessions;
   e. predefined action listeners for some operations.
2. Application-wide Message Viewer.
3. Library of additional visual components (status bar, tool bar buttons, etc.).
4. Splash screen.
5. Universal image and data capture:
   a. standard dialog to choose areas on the frame;
   b. export to file;
   c. printing;
   d. sending by e-mail;
   e. copying to the system clipboard
6. Extended application properties:
   a. access to the server-side property database;
   b. local configuration files to define default property values;
   c. saving properties, changed by the user, on the server.

7. Connection to the Data Acquisition Engine (DAE):
   a. GUI for DAE connection and settings control;
   b. status bar icon that shows connection status;
   c. configuration of DAE connection through application properties.
8. Extended logging:
   a. GUI for logging control;
   b. logging configuration through application properties;
   c. additional logging handlers:
      - `AppixHandler` to send messages to the server-side database;
      - `MessageViewerHandler` to send messages to the Message Viewer.
9. A procedure for the application self-determination (according to the data from the database).
10. An implementation of Application Locks, in order to avoid concurrent execution of critical routines.

1.2 Requirements & Configuration

1.2.1 JDK

Java Runtime Environment 1.4.1 is required. The latest version of JRE can be found at *http://java.sun.com/j2se/downloads.html*.

1.2.2 Classpath

The Application Framework root package is `gov.fnal.controls.framework`. All native framework classes and images are packed in `framework.jar`. If you have an access to the original *gov*-tree, you may use it in your classpath. Note, however, that in case of *jar*-files you can not substitute `framework.jar` with `gov.jar`, because the last one does not contain images.

In a minimal configuration, Application Framework requires four additional jar files: `jhall.jar`, `mail.jar`, `activation.jar` and `jlfgr-1_0.jar`. If the application needs DAE connection, `jconn2.jar` must be included in classpath, as well as the classes from the *gov*-tree, that implement DAE connection.

Thus, there are 5 possible classpath configurations for the Application Framework:

Table 1. Application Framework Classpath

<table>
<tr><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th></tr>
<tr><th colspan="2">Without DAE Connection</th><th colspan="3">With DAE Connection</th></tr>
<tr><td colspan="5">jhall.jar</td></tr>
<tr><td colspan="5">activation.jar</td></tr>
<tr><td colspan="5">mail.jar</td></tr>
<tr><td colspan="5">jlfgr-1_0.jar</td></tr>
<tr><td colspan="2"></td><td colspan="3">jconn2.jar</td></tr>
<tr><td rowspan="2">framework.jar</td><td rowspan="2">Original gov-tree</td><td colspan="2">framework.jar</td></tr>
<tr><td>gov.jar</td><td>govcore.jar</td></tr>
</table>

For convenience, an archive `af-bunch.jar` contains all data from the original archives `jhall.jar`, `activation.jar`, `mail.jar`, and `jlfgr-1_0.jar`. `af-bunch.jar` can be found at *//daesrv/java/jars/multi*.

1.2.3 Application Index

Application Framework must be able to establish `http` connection with Application Index (APPiX) server, *http://www-bd-fnal.gov/appix*. This server is available for both on-site and off-site applications.

It is recommended to register the application in APPiX database, otherwise some services (central logging, heartbeat, operation locks, and self-determination) will be unavailable.

1.3 Writing An Application: Basic Steps

This is a brief list of steps, that must be followed. For the example, see §9.

The main goal of the Application Framework is to simplify the development of console Java applications. Framework provides a set of classes that implement some frequently used, complex, and critical routines. At least, two principal classes can be used directly in the application: `ApplicationManager` and `JControlsFrame`.

`ApplicationManager` supports interactions between the user application as a single whole and the system environment: it implements access to the application properties, access to APPiX database, DAE connection, page setup for printing, and determination of the main application class. Only one instance of `ApplicationManager` may exist on a single VJM. This class does not have public constructors. A static `init` method must be called at the program startup, from the application's `main` method.

`JControlsFrame` extends `javax.swing.JFrame` (through `AbstractControlsFrame`). It is assigned to play a role of the main application frame. `JControlsFrame` already has the generic graphical user interface (menu-, tool- and status

bars) on the content pane and provides, through this GUI, access to the most of internal routines (printing, DAE connection, logging configuration, etc.). This frame also supports the retention of some properties (such as frame size and location) between sessions. User applications may have any number of `JControlsFrame` instances.

Graphical user interfaces are defined by so-called *frame descriptors*. The frame descriptor is an XML (plain text) file created at the developer's discretion. Every extension of `JControlsFrame` may have one corresponding descriptor or does not have any. Descriptors are heritable, as well as Java classes linked with them. It is supposed, that `JControlsFrame` should be extended in the user applications in order to define a custom content on the content pane (the content pane in our case is an area between tool- and status bars). If the custom XML descriptor is not created, `JControlsFrame`'s generic GUI is used, otherwise this custom descriptor extends the generic one.

To use Application Framework:

1. Call `ApplicationManager.init` in the beginning of the application's `main` method and pass application parameters as an argument.
2. Make your frame the extension of `JControlsFrame`, rather than `JFrame`:
   a. describe the frame explicitly either as a public class, or as a subclass; avoid creation of anonymous classes;
   b. do not create menu-, tool- and status bars manually; use XML descriptors to define them;
   c. use XML descriptor to set the frame title, icon and other simple properties;
   d. for every frame, put all initialization code in `jbInit` method;
   e. do not call `jbInit` in the application; this method is called by the superclass constructor before the initialization of instance variables (§2.2);
   f. if your frame is resizable, do not set its size—the size will be set automatically by the Framework; do not set frame location on the screen;
   g. in most cases, do not set the default close operation.
3. Create the property file for you application:
   a. set `application.help.topic` and `application.help.url` properties in order to enable the application help;
   b. specify `application.title`, `application.version`, and `application.author` in order to have them on the splash screen and *About* dialog.

4. If you need DAE connection:
   a. do not create `DaqUser` manually, use `ApplicationManager.getDaqUser` method instead;
   b. set default values for `dae.name.fixed`, `dae.enabled`, `dae.connect`, and `dae.unlock.time` properties in the application property file.

2

Generic Frame

All applications designed for public usage must provide a standard and intuitive graphical user interface.

The class `gov.fnal.controls.framework.JControlsFrame`, an indirect successor of `javax.swing.JFrame`, implements so-called *generic graphical user interface*. Generic GUI has many of customary graphical widgets predefined, as well as action listeners for them. Developers can extend `JControlsFrame` in order to create the custom GUI, inheriting generic frame appearance and behavior. Both generic and custom GUIs are defined in text files, called *interface descriptors*.

The components, not defined in the interface descriptor (e.g., elements on the content pane), must be initialized and arranged in `jbInit` method. This method is called automatically by `JControlsFrame` during its creation.

2.1 Interface Descriptors

All generic graphical components on `JControlsFrame` are created dynamically at the runtime using *reflection*. The definition of these components is stored in text files (interface descriptors). Each `JControlsFrame` successor may have only one interface descriptor, or does not have it at all. The descriptor must be placed in the same directory as the corresponding class, and have the same file name with *.xml* extension. Frame descriptors for anonymous classes are not supported. If the frame is described as a named subclass, the descriptor's file name should be:

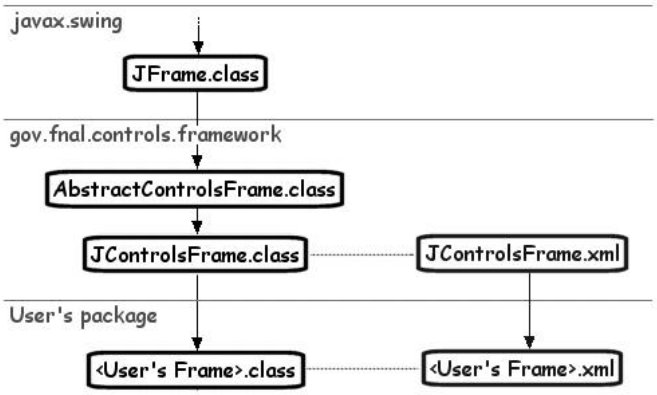
```
<class>$<subclass>.xml
```

for example, `MyClass$MyFrame.xml`.

Interface descriptor format is discussed in §1.

Interface descriptors are heritable, like Java classes. For example, if `MyFrame.class` extends `JControlsFrame.class`, `MyFrame.xml` will extend `JControlsFrame.xml`. Although XML standard does not support inheritance itself, a special method for descriptors has been developed. Interface

descriptor inheritance gives an opportunity to generalize GUI development. In a simplest case, the programmer may use the generic interface without any changes—only an extension of `JControlsFrame.class` is required. Then, it is possible to create an extension of `JControlsFrame.xml` in order to describe required changes in GUI (e.g., removing unnecessary elements, adding new ones, changing frame title, etc.). Next figure illustrates these two threads of inheritance:

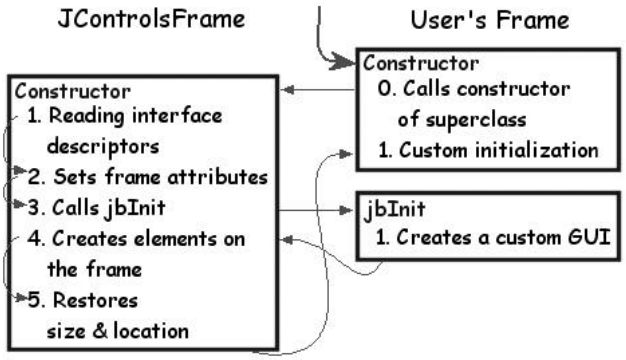


On this diagram, an abstract `AbstractControlsFrame.class` implements all functions of the generic frame, but does not have the descriptor. `JControlsFrame.class` extends `AbstractControlsFrame.class`; it does not implement any new functions, but has the corresponding descriptor, `JControlsFrame.xml`. If the user extends `JControlsFrame.class`, the new class inherits both functions and the GUI. Of course, it is possible to extend `AbstractControlsFrame.class` to inherit all functions, but create a new frame descriptor from the beginning.

Since extension of `JControlsFrame` is a preferred way, only this class is used below in this guide.

2.2 Frame Initialization

During the initialization, `JControlsFrame` reads interface descriptors and sets frame attributes and IDs of menu-, tool- and status bars. Later these values may be altered in `jbInit`. The following diagram illustrates frame initialization sequence:



Application Framework shows a splash screen during initialization of the first `JControlsFrame` on current VJM.

There are several important points, related to the initialization procedure:

1. `jbInit` is called automatically by the superclass constructor; you should not call it by yourself.
2. `jbInit` is called before the initializers of instance variables are evaluated. That means that if you use a global instance variable in `jbInit`, it must be initialized there, rather than by an initializer placed at the variable definition or in the constructor. Those two initializers will be called after `jbInit` finishes.
3. By the moment when `JControlsFrame` calls `jbInit`, it has the interface descriptors already loaded, but all the widgets are not instantiated yet. So, you can change `menuBarID`, `toolBarID` and `statusBarID`, but `getWidget` method always returns `null`. If you want to access particular widgets, you should do this in the constructor.

2.3 Default Visibility

The way to call `setVisible` from `jbInit` to make the frame visible is inappropriate, because there are several hidden routines being performed after that. `JControlsFrame` has an attribute `visibleOnStartup` that defines whether the frame must become visible immediately after the initialization is completed. Use the frame descriptor in order to set this attribute, or call `setVisibleOnStartup` from `jbInit`.

2.4 Default Close Operation

Unlike in `javax.swing.JFrame`, the `defaultCloseOperation` attribute in `JControlsFrame` is set automatically during the initialization. For the first `JControlsFrame` instance on current VJM the value of `defaultCloseOperation`

is `EXIT_ON_CLOSE`, otherwise—`DISPOSE_ON_CLOSE`. You may change the default value at any time from the program.

2.5 Element Access

Graphical widgets, dynamically created on the frame, have no direct references in `JControlsFrame`. In other words, there are no variables that can be used to get those components. Instead of direct access, access by ID is used.

Each element has an ID (text string). In the interface descriptor each ID is unique. On the frame this is not necessarily true: in general, more than one instance of each element may be created. E.g., it may be several buttons "Print". The elements with one ID may have different entities, such as buttons and menu items. In spite of this, all of them have the same behavior and similar appearance: one icon (if icon is possible), text, visibility, enable and select statuses.

To access an element, `getWidget` method should be used:

```
public ControlsWidget getWidget( String widgetID )
```

This method returns a `ControlsWidget` object. It allows to handle the set of elements as a single whole. The following methods are defined for `ControlsWidget` instance:

```
void setText( String )          String getText()
void setEnabled( boolean )      boolean isEnabled()
void setVisible( boolean )      boolean isVisible()
void setSelected( boolean )     boolean isSelected()
Object[] getComponents()
```

Menu-, tool- and status bars have IDs, as well. `JControlsFrame` supports the following methods:

```
void setMenuBarID( String )     String getMenuBarID()
void setToolBarID( String )     String getToolBarID()
void setStatusBarID( String )   String getStatusBarID()
```

`JControlsFrame` constructor calls `set...` methods with parameters coming from the interface descriptor. Since `jbInit` method is called after that, these IDs may be altered in `jbInit`. Use `null` value to hide a specific bar.

The content pane does not have an ID. To access it, use `getContentPane` method, as usually.

***Note**: `getWidget` method does not work for menu-, tool-, status bars and the content pane.*

2.6 Event Handling

Events from the dynamically created elements are handled through `javax.swing.ActionMap`. By default, `JControlsFrame` has a map, that may be get and altered using `getActionMap` and `setActionMap` methods.

`ActionMap` keeps the correspondence between event handlers and their keys. In our case, the event key is a value of the component's `actionCommand` attribute. These attributes are defined in the interface descriptor. If `actionCommand` is not set, element ID is used instead.

Following example shows how to write action listeners. This particular code replaces a default event handler for the "Exit" menu item.

```
protected void jbInit() throws Exception {

  AbstractAction newExitAction = new AbstractAction() {
    public void actionPerformed( ActionEvent e ) {
      System.out.println( "Exit pressed" );
    }
  };

  getActionMap().put( "ExitAction", newExitAction );
}
```

Some elements of the generic GUI have predefined action listeners. Setting new `ActionMap` does not remove them. The only way to replace or remove predefined action listeners is to use `put`, `clear` or `remove` methods of `javax.swing.ActionMap`.

Table 2. Elements with predefined listeners

Menu Bar	*Tool Bar*	*Status Bar*	*Action*
Export To File			Opens capture dialog and saves selection to a file
Send Mail			Opens capture dialog and sends selection by e-mail
Print			Opens capture dialog and prints selection
Page Setup			Opens the standard Page Setup dialog
Cut			Cuts selection to the system clipboard
Copy			Copies selection to the system clipboard
Copy Special...			Opens capture dialog and copies selection to the system clipboard
Paste			Pastes data from the system clipboard
Select All			Selects all data on the currently focused component
DAE Connection			Opens DAE Connection dialog; disabled if DAE routines are unavailable
Message Viewer			Opens Message Viewer; status bar button is hidden if there are no new messages.
Application Properties			Shows application properties loaded from the database
Logging			Opens logging configuration dialog
Help Topics			Opens application help; enabled only if `application.help.topic` and `application.help.url` properties are set
About			Shows information window
Exit			Calls default close operation for the current frame; see §2.4

In the generic GUI, all items without predefined action listeners are hidden.

2.7 Persistent Frame Properties

Frame size, location and tool bar position are persistent by default. That means that Applications Framework keeps these values for every `JControlsFrame` instance between sessions and tries to restore them at the startup. `JControlsFrame` supports the following methods:

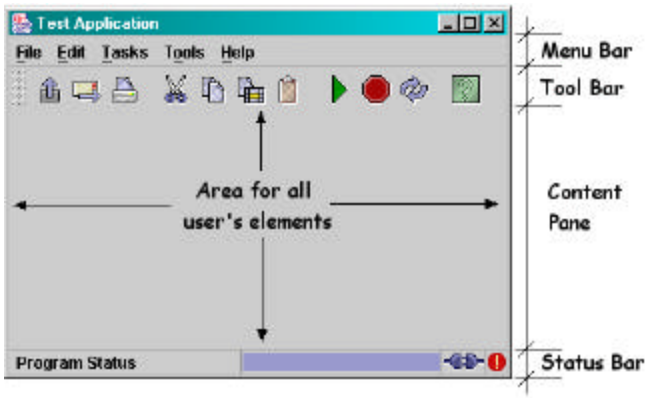
```
void setRestoreSize( boolean )        boolean isRestoreSize()
void setRestoreLocation( boolean )    boolean isRestoreLocation()
```

Frame properties are stored in a file *$USER_HOME$/ControlsFrames.properties*. To use a different file, put the full file name in `frame.property.file` system property. For example:

```
-Dframe.property.file=/usr/home/jdoe/.ControlsFrames
```

2.8 Appearance

`JControlsFrame` looks as follow:



There are four areas on the frame: **menu bar**, **tool bar**, **content pane** and **status bar**. Menu-, tool- and status bars are created by Application Framework. Content pane is a `JPanel` for all user components, such as labels, buttons, trees etc.

2.8.1 Menu Bar

Element ID = `MenuBar`.

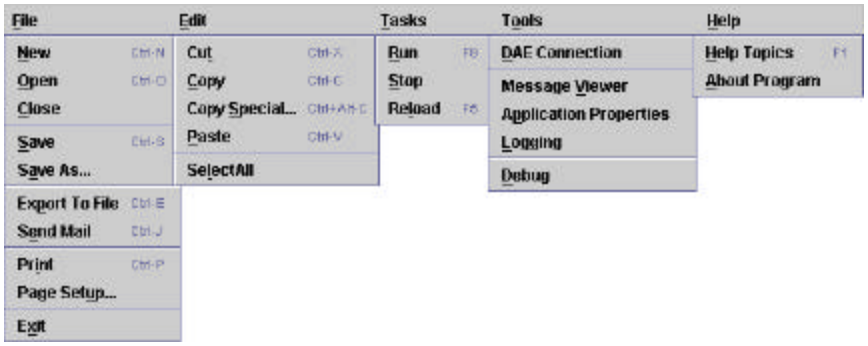


Table 3. Generic menu bar items

Name	*Element ID*	*ActionCommand*	*Function*
File	`File`		
New	`New`	`NewAction`	*
Open	`Open`	`OpenAction`	*
Close	`Close`	`CloseAction`	*
Save	`Save`	`SaveAction`	*
Save As...	`SaveAs`	`SaveAsAction`	*
Export To File	`Export`	`ExportAction`	Opens capture dialog to save image or text to a file
Send Mail	`Send`	`SendAction`	Opens capture dialog to send image or text by e-mail
Print	`Print`	`PrintAction`	Opens capture dialog to print image
Page Setup...	`PageSetup`	`PageSetupAction`	Opens standard Page Setup dialog
Exit	`Exit`	`ExitAction`	Calls default close operation for the current frame
Edit	`Edit`		
Cut	`Cut`	`CutAction`	Cuts selection to the system clipboard
Copy	`Copy`	`CopyAction`	Copies selection to the system clipboard

Generic menu bar (continued).

Name	*Element ID*	*ActionCommand*	*Function*
Copy Special...	`CopySpecial`	`CopySpecialAction`	Opens capture dialog to copy image or text to the system clipboard
Paste	`Paste`	`PasteAction`	Pastes data from the system clipboard
Select All...	`SelectAll`	`SelectAllAction`	Selects all data on the currently focused component
Tasks	`Tasks`		
Run	`Run`	`RunAction`	*
Stop	`Stop`	`StopAction`	*
Reload	`Reload`	`ReloadAction`	*
Tools	`Tools`		
DAE Connection	`DaeConnection`	`DaeConnectionAction`	Opens DAE Connection dialog
Message Viewer	`MessageViewer`	`MessageViewerAction`	Opens application-wide message viewer
Application Properties	`AppProperties`	`AppPropertiesAction`	Shows application properties loaded from the database
Logging	`Logging`	`LoggingAction`	Opens GUI for logging configuration
Debug	`Debug`	`DebugAction`	**
Help	`Help`		
Help Topics	`HelpTopics`	`HelpTopicsAction`	Calls application Java Help
About...	`About`	`AboutAction`	Opens information dialog

* To be defined by the developers.
** Will be implemented later.

2.8.2 Tool Bar

Element ID = `ToolBar`.

Table 4. Generic toolbar items

Icon	*Element ID*	*Icon*	*Element ID*
	`Export`		`Paste`
	`Send`		`Run`
	`Print`		`Stop`
	`Cut`		`Reload`
	`Copy`		`HelpTopics`
	`CopySpecial`		

2.8.3 Status Bar

Element ID = `StatusBar`.

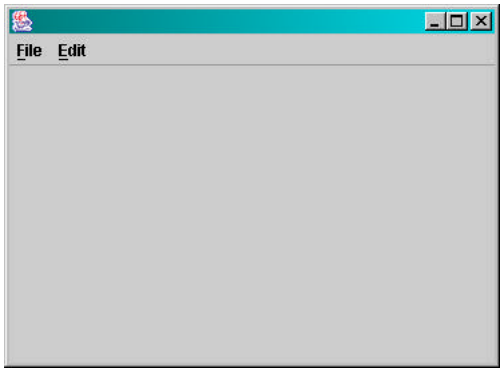
Table 5. Generic status bar items

#	*Element ID*	*Icon*	*Class*	*Function*
0	`Status`		`gov.fnal.controls.framework.↵ swing.JControlsStatusItem`	Status message
1	`Progress`		`gov.fnal.controls.framework.↵ swing.JControlsProgressBar`	Progress bar
2	`DaeStatusButton`		`gov.fnal.controls.framework.↵ swing.JDaeStatusButton`	DAE Connection
3	`MAButton`		`gov.fnal.controls.framework.↵ swing.JMessageAlarmButton`	Message Viewer

2.9 Simplified GUI

The simplified graphical user interface is intended for secondary windows. It is implemented in `gov.fnal.controls.framework.JSimpleControlsFrame`.

This GUI has only controls, related to the image and text capture and clipboard operations. Simplified GUI does not have tool- and status bars. Menu bar includes two items: “File” and “Edit”:



`JSimpleControlsFrame` can be customized using an XML descriptor.

2.10 Generic Dialogs

Application Framework has three classes that implement generic dialogs: `gov.fnal.controls.framework.JControlsDialog0`, …`1`, and …`2`. These classes are successors of `AbstractControlsDialog`, and they differ only in the set of control buttons. The generic dialogs are similar to the generic frame. Instead of menu-, tool- and status bars they have a button bar, defined in the interface descriptor. Content pane is the same. By default, the generic dialogs are modal (i.e., other windows may not be used while the dialog is open).

`JControlsDialog0`

`JControlsDialog1`

`JControlsDialog2`

Events from the control buttons are handled through the `ActionMap`, as described in §2.6. “Ok”, “Cancel”, and “Apply” buttons have `OkAction`, `CancelAction`, and `ApplyAction` action commands assigned, correspondingly. Button events are handled internally and generate two second-level actions, `CommitAction` and `RollbackAction`:

Table 6. Generic dialog actions

Button	*Initial Action*	*Generated Actions*
Ok	`OkAction`	`CommitAction` → dialog closes
Cancel	`CancelAction`	`RollbackAction` → dialog closes
Apply	`ApplyAction`	`CommitAction`

By default, the second-level keys are not linked with handlers in `ActionMap`. User can either write second-level action handler(s), or redefine original first-level handler(s). The first way is preferable:

```
protected void jbInit() throws Exception {

  AbstractAction commitAction = new AbstractAction() {
    public void actionPerformed( ActionEvent e ) {
      // a code to save data
    }
  };
  AbstractAction rollbackAction = new AbstractAction() {
    public void actionPerformed( ActionEvent e ) {
      // a code to restore data, rollback changes, etc.
    }
  };

  getActionMap().put( "CommitAction", commitAction );
  getActionMap().put( "RollbackAction", rollbackAction );

}
```

In order to prevent the dialog from closing, commit and rollback handlers may generate an `gov.fnal.controls.framework.helpers.OperationCancelledException`.

Generic dialogs have three attributes, that define default actions. These attributes can be assigned through the interface descriptors or in `jbInit`:

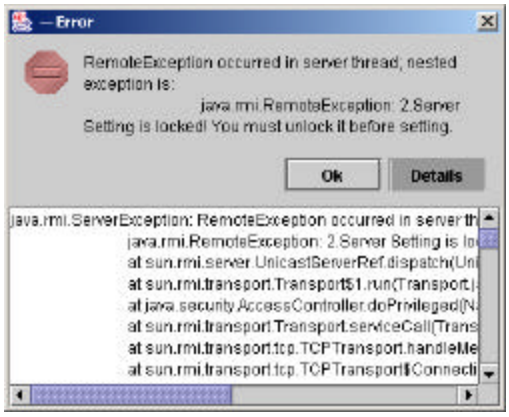
Table 7. Default actions in generic dialogs

Attribute	*Function*	*Default Values**		
		…0	…1	…2
`defaultAction`	When *Enter* is pressed on the dialog	**C**	**O**	**O**
`cancelAction`	When *Esc* is pressed on the dialog	**C**	**C**	**C**
`windowAction`	When the dialog is closed through the window control icon	**C**	**C**	**C**

* Default values for `JControlsDialog0`, `…1`, and `…2`, correspondingly
(C) = `CancelAction`, **(O)** = `OkAction`.

2.11 Extended Message Box

The class `gov.fnal.controls.framework.JControlsMessageBox` implements a modal message box with "Details" button:



In order to activate `JControlsMessageBox`, use the following static methods:

```
public int showError( String message, String details )
public int showError( Throwable error )
public int showInformation( String message, String details )
public int showWarning( String message, String details )
```

All these methods return a button code: `OK_RESULT`, `CANCEL_RESULT`, `YES_RESULT`, or `NO_RESULT`. In case of `showError(Throwable error)`, the exception's stack trace is used as message details.

3

Interface Descriptor Format

3.1 Document Type Definition

Document Type Definition (DTD) for the interface descriptor can be found at *http://www-bd.fnal.gov/appix/gui.dtd*. This DTD has no public ID. The valid document type declaration must be included in every interface descriptor file:

```
<!DOCTYPE frame SYSTEM "http://www-bd.fnal.gov/appix/gui.dtd">
```

The XML parser used by Application Framework must have online access to this DTD in order to validate document structure and to read default attributes.

3.2 Descriptor Structure

Each interface descriptor element is linked to one or several Java classes. Element attributes correspond to the class instance fields. They are passed through appropriate `set...` methods. For example, if an XML element has attribute named `visible`, the value of this attribute will be passed to the class instance by calling `setVisible`. There are two exceptions of this rule:

- Attribute `id` is the element's primary key; it is never passed to the object;.
- Attribute `class` specifies the list of class names. It is used to create the object. In most cases, the value of this attribute is taken from DTD and does not have to be defined in XML file.

All elements, except of `frame`, must have unique ID.

When Application Framework sets fields, the appropriate `set...` method must have only one parameter of one of the following types:

`boolean` or `java.lang.Boolean`
`char` or `java.lang.Char`
`int` or `java.lang.Integer`
`javax.swing.Icon`
`javax.swing.KeyStroke`

To set an icon, `JControlsFrame`'s `getImageIcon` method is used.

Each element may have zero, one, or several subelements. In most cases, subelements are added to the parent element with `add` method of the parent. Exceptions are the `frame` element and separators.

Subelements may be defined explicitly:

```
<menu id="mainMenu">
    <item id="saveItem"/>
</menu>
```

or by reference:

```
<menu id="mainMenu">
    saveItem
</menu>
<item id="saveItem"/>
```

In the last case, the parent element has a list of children IDs (separated by spaces and/or line feeds). Subelements may be defined anywhere in the current document or its predecessor. It is possible to mix explicit declarations and declarations by reference in one element.

Several references to one elements may exist.

Special identifier `!separator` defines menu and toolbar separators. `addSeparator` method is used to place it on parent element.

Some elements are polymorphous: their classes depend on parent classes. For example, `class` attribute for the `item` element is defined in DTD as:

```
class %ClassMap; "gov.fnal.controls.framework.JControlsButton;↵
  javax.swing.JMenu=javax.swing.JMenuItem"
```

This expression says, that the default class for the element is `JControlsButton`, but if its parent is an instance of `JMenu`, element's class will be `JMenuItem`.

3.3 Descriptor Inheritance

The previous chapter gives the diagram of descriptor inheritance (see §2.1). Below is a detailed explanation of this procedure.

Element IDs are unique across all sequence of interface descriptors. If an element with the same ID is defined in the child document, this declaration overrides the parent declaration. It does not matter, whether this element is on the top level or it is a subelement. If overriding takes place:

- Attributes, defined in the child document, replace attributes from the parent document.
- Attributes, defined in the parent document and not redefined in the child one, remain without changes.
- Subelements, declared in the child document, replace subelements, declared in the parent document. To place subelement list from the parent document, a special identifier `!default` can be used in the child document.

Example:

Parent document	*Child document*	*Result*

`<item id="item1"`
`visible="true"`
`enabled="false"`
`item1_1`
`item1_2`
`</item>`
	`<item id="item1"`
	`enabled="true"`
	`text="a_text">`
	`item2_1`
	`!default`
	`item2_2`
	`</item>`
		`<item id="item1"`
		`visible="true"`
		`enabled="true"`
		`text="a_text">`
		`item2_1`
		`item1_1`
		`item1_2`
		`item2_2`
		`</item>`

3.4 Root Element

Root element of the interface descriptor is `frame`. It defines `JControlsFrame` attributes, such as menu-, tool- and status bar IDs, but does not have ID itself:

```
<frame menuBarID="MenuBar" toolBarID="ToolBar"
statusBarID="StatusBar">
```

When Application Framework creates a `JControlsFrame` instance, it reads `menuBarID`, `toolBarID` and `statusBarID` properties, creates instances of these bars and puts them on the frame.

`frame` element conforms to the rules of inheritance, described above.

3.5 List Of Elements

Table 8. GUI Elements

Element name	*Description*	*Corresponding class(es)*	*Attributes*[1]	*Sub-elements*

`frame`	Root element; describes main application frame	`gov.fnal.controls.framework.↵ JControlsFrame`	`menuBarID`
			`toolBarID`
			`statusBarID`
			`restoreSize`
			`restoreLocation`
			`visibleOnStartup`
			`title`
			`icon`
				`menu_bar`
				`tool_bar`
				`status_bar`
				`menu`
				`item`
				`radio`
				`check`
				`status`
				`progress`
				`widget`

`menu_bar`	Menu bar	`javax.swing.JMenuBar`	`id`[2] `class`[3]	`menu` `widget`

`tool_bar`	Tool bar	`gov.fnal.controls.framework.↵ JControlsToolBar`	`id`
			`class`
				`item`
				`radio`
				`check`
				`widget`

`status_bar`	Status bar	`gov.fnal.controls.framework.↵ JControlsToolBar`	`id`
			`class`
				`status`
				`progress`
				`widget`

`menu`	Menu	`javax.swing.JMenu`	`id`
			`class`
			`text`
			`mnemonic`
				`item`
				`radio`
				`check`
				`widget`

`item`	Menu- or tool bar item	`gov.fnal.controls.framework.↵ JControlsButton`
		or
		`javax.swing.JMenuItem`
			`id`
			`class`
			`text`
			`mnemonic`
			`actionCommand`
			`accelerator`
			`icon`
			`enabled`	

GUI Elements (continued)

Element name	*Description*	*Corresponding class(es)*	*Attributes* [1]	*Sub-elements*

`radio`	Menu- or tool bar radio item	`gov.fnal.controls.framework.`↵ `JControlsRadioButton`
		
		or
		
		`javax.swing.`↵ `JRadioButtonMenuItem`
			`id`
			`class`
			`text`
			`mnemonic`
			`actionCommand`
			`accelerator`
			`icon`
			`enabled`
			`selected`	

`check`	Menu- or tool bar check item	`gov.fnal.controls.framework.`↵ `JControlsCheckButton`
		
		or
		
		`javax.swing.`↵ `JCheckBoxMenuItem`
			`id`
			`class`
			`text`
			`mnemonic`
			`actionCommand`
			`accelerator`
			`icon`
			`enabled`
			`selected`	

`status`	Status bar item	`gov.fnal.controls.framework.`↵ `JControlsStatusItem`	`id`
			`class`
			`text`
			`enabled`
			`alignment`
			`resizable`
			`width`	

`progress`	Status bar progress gauge	`gov.fnal.controls.framework.`↵ `JControlsProgressBar`	`id`
			`class`
			`text`
			`enabled`
			`resizable`
			`width`	

`widget`	Arbitrary component	To be defined in `class` attribute	`id` `class`	

[1] Only principal attributes are shown here; in fact, any instance fields, that meet the requirements in §3.2, may be set.
[2] `id` attribute is required.
[3] `class` attribute is required, but for all elements except of `widget` it is defined in DTD.

4

Managing Properties

Application Framework provides a unified system of property distribution. It may be useful to store application properties and to control behavior of the application from one point: e.g., in order to change the default DAE name for all instances of the application.

4.1 Getting Properties

To access the set of properties from the application, `ApplicationManager.getProperties` static method should be used. This method returns `java.util.Properties` object. The application can not only read values, but modify and add new ones, as well. Note, that `setProperties` method is not supported by `ApplicationManager`, so all changes must be made in the existing object.

In general, the property name may be random. However, to avoid a conflict between different applications, it is recommended to use package name in the beginning of the property name. For example: `gov.fnal.controls.applications.monitor.DeviceTable.SortOrder`.

4.2 Storing Properties

There are three locations, where the properties are stored and can be initially defined: global property file, application property file and the property server.

- Global property file is the file `/gov/fnal/controls/framework/ApplicationFramework.properties`. It defines values, that are common for the whole Application Framework. Users may not change this file.
- Every application can have its own property file. Name of this file must correspond to the name of application's main class (class, that contains `main` static method), and have *.properties* extension. Application property file is optional.
- Property server is a set of servlets connected to the database.

Application Framework never modifies global and application property files. All changes, made at the runtime inside the application, are saved on the server.

Properties are retrieved in a strict order:

(1) global property file → (2) application property file → (3) property server

On every step, new values overwrite old ones. Thus, data from the property server always have the highest priority.

4.3 Property Server

Property server (Property Repository) is a set of servlets. Properties themselves are stored in the database. The web interface to manage the repository is available at *http://www-bd.fnal.gov/appix/select/props*.

Every record, that describe property in the database, has two additional attributes: application and user. So, properties may be application-specific and/or user-specific. Property name is not a primary key. That means, for example, that two records with the same name may exists, if one of them is application-specific, and another one—is not.

When the servlet retrieves a set of properties by the Framework's request, application-specific and user-specific properties have higher priority, than others. All properties changed by user applications at the runtime are stored as user-specific, but not application-specific.

4.4 List Of Used Properties

Following properties are used by the current version of Application Framework (properties of logging handlers are not shown):

Table 9. Application Framework properties

Name	*Defined in:*	*Function*
`.level`	G	Global logging level
`appix_server`	G	APPiX server URL
`application.author`	A	Application author(s)
`application.help.topic`	A	Help topic ID

Application Framework properties (continued)

Name	*Defined in:*	*Function*
`application.help.url`	A	URL of application help file
`application.title`	A	Application name
`application.version`	A	Application version
`dae.connect`	G	`1`, if DAE must be connected at the application startup
`dae.enabled`	G	`1`, if DAE connection is enabled for this application
`dae.name`	S	Default DAE name
`dae.unlock.time`	G	Default settings unlock time in minutes; zero value locks settings, negative values unlock settings forever
`framework.heartbeat`	G	`1` to allow Heartbeat Service
`framework.version`	G	Current Application Framework version
`gov.fnal.controls.framework.level`	G	Application Framework logging level
`handlers`	G	List of available logging handlers
`message.autoopen`	G	`1` if Message Viewer must be automatically opened when new message comes up.
`protected.logging.handlers`	G	List of logging handlers, that may not be changed through logging configuration GUI
`user.email.address`		Email address of the current user
`user.email.addressee`		Last used email addressee
`user.export.delimiter.index`		Last used text delimiter for the export utility
`user.export.file`		Last export file
`user.export.qualifier.index`		Last user text qualifier for the export utility
`user.print.center`		`1` if image must be centered while printing
`user.print.fit`		`1` if image must fit the page while printing

Application Framework properties (continued)

Name	*Defined in:*	*Function*
`user.print.transparent`		`1` to remove background while printing
`user.smtp_server`	S	SMTP server name

(G) = global property file
(A) = application property file
(S) = property server.

4.5 Setting Properties For Application

1. Create text file with the same name as your main class and *.properties* extension. Place this file in the same directory, as your main class.
2. Every property definition must occupy on line and have the following format:

   `<name>=<value>`

3. Set `application.title`, `application.version` and `application.author`.
4. Set `dae.enabled` and `dae.connect` properties, if needed.

5

DAE Connection

5.1 General Description

Application Framework provides an advanced procedure for the Data Acquisition Engine (DAE) connection.

To create the DAE connection, in general, the application must obtain an instance of `gov.fnal.controls.daq.acquire.DaqUser`. Each instance of this object corresponds to an opened connection. `DaqUser` describes the DAE client: a combination of user, node, and service. In our case, the *service* is an application, and Application Framework provides full information about it, including privileges. The information is being taken from APPiX database through servlets.

Application Framework provides two ways to manage the DAE connection: a programmatic way, and through the generic user interface.

The programmatic way allows to:

- set the default DAE name (for debugging);
- connect to DAE every time at the startup;
- have a listener to handle DAE-related events (connect, disconnect, etc.).

The generic GUI allows to:

- connect and disconnect the DAE at the user discretion;
- change the DAE name (this value is persistent for every user);
- lock and unlock settings.

5.2 System Requirements

In order to work with DAE, the application must have two items included in the classpath: `govcore.jar` and `jconn2.jar`. As far as `govcore.jar` is a subset of *gov*-tree, it may be substituted by either `gov.jar`, or by the original *gov*-tree

(see §1.2.2). If these libraries are not included, DAE connection is unavailable and all related controls are hidden.

The client machine must have access to the DAE; in most cases this is unavailable off site.

The application may connect only to the DAE, that is assigned to serve specific application types. The default system-wide DAE name is Ok in most cases. At present time, the following servers are working with general purpose console user programs:

- *dse08.fnal.gov* (default)
- *dse09.fnal.gov*
- *dse10.fnal.gov*

You can use the local DAE, as well.

5.3 DAE-Related API

DAQ user should not be created explicitly in the application. A shared `DaqUser` instance is provided by the following static methods:

```
ApplicationManager.getDaqUser()
ApplicationManager.getDaqUser( boolean forceConnection )
```

Both of these methods return an `Object`, that is actually `DaqUser` instance, or `null`. The first method is equal to `getDaqUser(false)`. The Boolean argument switches two modes:

- `forceConnection==false`:
  - if the connection has already been opened, the method returns `DaqUser` instance immediately;
  - if the connection is in progress, the method waits until a thread finishes and returns either valid instance, or `null`;
  - otherwise, `null` is returned immediately.
- `forceConnection==true`:
  - if the connection has already been opened, the method returns `DaqUser` instance immediately;
  - if the connection is in progress, the waits until a thread finishes and returns either valid instance, or `null`;
  - if the connection is enabled but closed, the method initiate the connection, waits until the thread finishes and returns either valid instance, or `null`;
  - otherwise, `null` is returned immediately.

As long as the user can disconnect from DAE using GUI controls, it is impossible to guarantee, that `DaqUser` instance will be valid all the time. To avoid this problem, a `DaeConnectionListener` should be used to inform the application when the DAE connection is changed:

```
ApplicationManager.getDaeSupport().addDaeConnectionListener(…);
```

5.4 Configuration

By default, the application is disconnected from DAE, but the connection can be established using DAE Configuration dialog.

The are several application properties (see §1), that may be used:

Table 10. DAE-related properties

Property Name	*Value format*	*Default value*	*Function*
`dae.connect`	`0/1`	`0`	Sets whether DAE connection should be opened at the startup
`dae.enabled`	`0/1`	`1`	Sets whether DAE connection (and related GUI controls) should be available for the application
`dae.name`[1]	URL	`dse08.fnal.gov`	DAE name
`dae.name.fixed`	URL	*none*	Fixed DAE name for debugging
`dae.unlock.time`	integer	`0`	Settings control: • **>0:** settings are enabled for the specific time (in minutes); • **=0:** settings are disabled; • **<0**: settings are enabled for unlimited time. Settings unlock time is counted from the moment when DAE connection is established.

[1] If DAE name is changed through GUI, the new value is stored on the server as a user-specific property.

Properties `dae.enabled`, `dae.connect`, `dae.unlock.time`, and `dae.name.fixed` can be assigned in the application property file. Meanwhile, the default value

for `dae.name` is stored on the property server, and it **can not** be re-defined through the application property file.

Two values for DAE name are provided in order to make the management more flexible. Application Framework tries to use `dae.name.fixed` first. If this property exists, its value is used in DAE connection, and the user is not allowed to change it; in other words, the user may connect only to the specified server in this case. Otherwise, Application Framework uses `dae.name`, and allows the user to change it through GUI. The default value for `dae.name` always exists on the property server. If the user enters new DAE name, it will be stored on the server and will be used later in **all** framework-based applications without `dae.name.fixed`, started by this user, as the default DAE name.

5.5 Settings Control

Application Framework provides a support for disallowing settings, similar to VMS. By default, settings are locked (disabled). Settings may be unlocked (enabled) for a certain period of time, or forever, using GUI. The maximum allowed unlock time is subject of account's and node's privileges. GUI shows only allowed values. Unlock forever is available only for very limited number of people. For those who have permissions to unlock settings forever, settings are always unlocked at the application startup.

For debugging, developers may unlock settings by default. To do this, application property `dae.unlock.time` should be specified in the property file. Positive values of this property are considered to be unlock duration in minutes. A negative value means "unlock forever". Zero value locks (disables) settings. If settings are unlocked before the connection, the unlock time "count down" begins only after the connection is opened.

Settings may be locked/unlocked from the Java code using:

```
ApplicationManager.getDaeSupport().lock()
ApplicationManager.getDaeSupport().unlock(int duration)
```

Do not change the default settings status in production versions!

The users must have settings locked and unlock them manually, in order to avoid mistakes. MCR is the only exception, so far.

5.6 Security Issues

In order to define whether the program can do settings, three types of permissions are checked: account's, node's and service's. The set of service's permissions is defined in Application Index for every program (see APPiX

Guide, §6.3). Please note, that the DAE uses only actual access classes (MCR, Development, ...), but not *pseudo-classes* (PUBLIC and INSIDER).

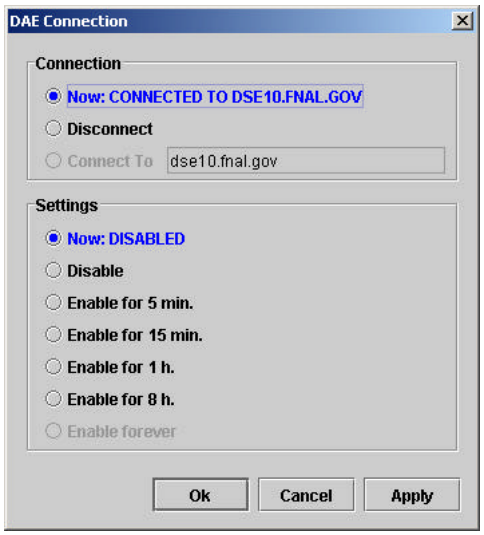
Applications not registered in Application Index use a default service permissions, `0x7fffffff`. This value allows the application to do everything, so the actual type of access in this case is defined using node's and account's privileges. Developers are required to use this default service privileges only during debugging and for applications, that can not be registered in Application Framework (e.g., servlets).

5.7 GUI Controls

The generic GUI has an icon on the status bar, that shows current status of DAE connection:

 DAE disconnected.
 DAE connected; settings disabled.
 DAE connected; settings enabled.

To open DAE Connection dialog, click this icon, or use "Tools/DAE Connection..." menu.



The top part of the dialog is designed to switch the connection on and off, and choose DAE name. The bottom part is used to control settings.

6

Image And Text Capture

6.1 Service Description

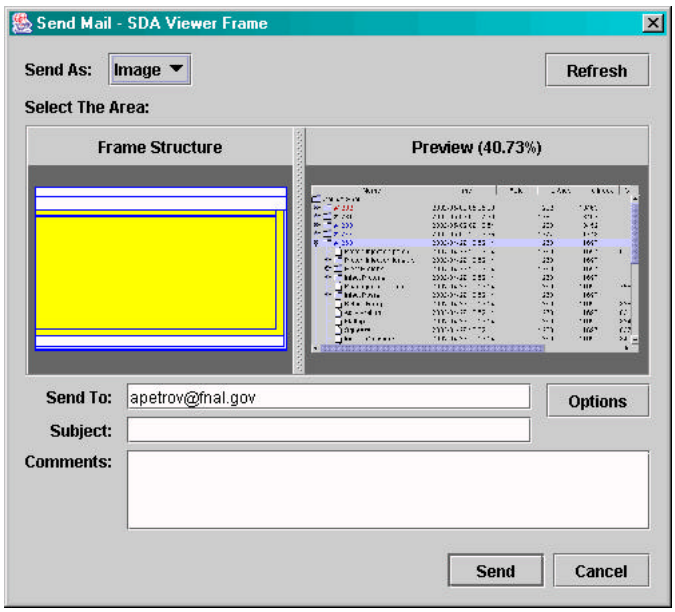
The capture utility allows to get current data from the selected area on the frame as an image, or convert it to a text. This utility can work with any extension of `AbstractControlsFrame`: it does not need to know *a priori* what is placed on the content pane in the specific implementation, and every time analyzes this content “on the fly”.

Captured data may be saved in a file, sent by e-mail, printed, or copied to the system clipboard. Capture mode (image or text) is selected by the user, except of printing—the printing utility always handles images.

In the image mode, any area on the frame may be chosen for the processing. In the text mode—only components, that support text conversion. Fields delimiter and qualifier can be specified by the user.

6.2 Selecting Data

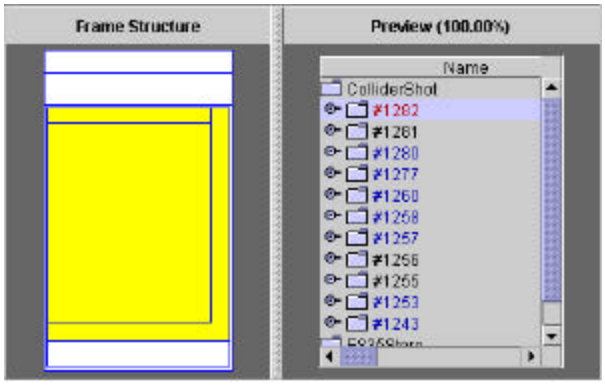
The Capture Dialog provides a graphical interface to select areas on the frame. Only one area my be selected at one time. For example, this is the screen shot of the Capture Dialog for e-mail:



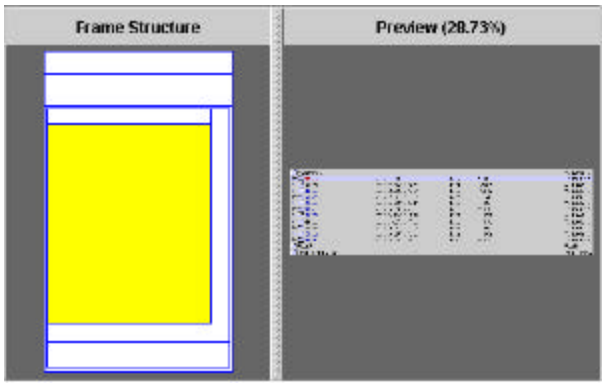
Left panel shows the current frame layout. The selected area is highlighted in yellow. Right panel shows a preview of the selected area.

Every frame has a number of elements placed on it in hierarchical order. If you select the element that is parent for other elements, all children will be selected, too.

If you have `JScrollPane` on you frame, and it renders only a part of the object (tree, table, list), you can select an internal component of this scroll pane (this component is `JViewport`) and get the whole object:



`JScrollPane` *is selected. Only visible part is captured.*



`JViewport` *is selected. Whole table is captured.*

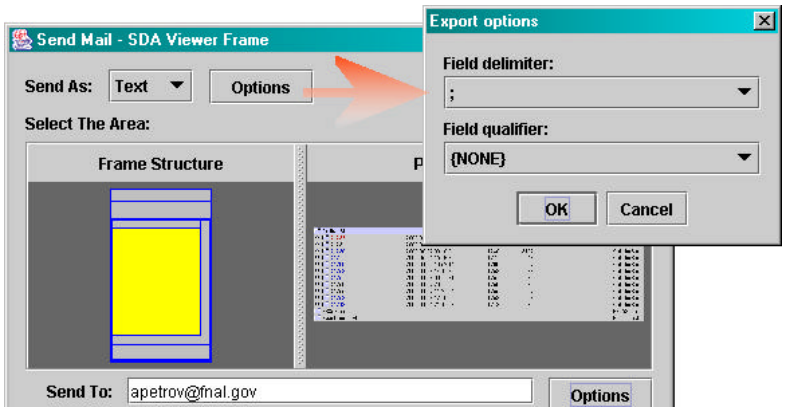
Image selection may not exceed 1000×1000 pixels. Otherwise it will be truncated. Truncated borders are marked with a broken red line.

6.3 Text Conversion

Content of several object can be converted to a text. These objects are:

- `javax.swing.text.JTextComponent`
- `javax.swing.JTable`
- `javax.swing.JTree`

When text mode is selected, all areas on the frame, that do not contain listed components, are disabled (gray). If you get data from either `JTree` or `JTable`, field delimiter and field qualifier can be specified. Field delimiter separates data of different cells. Field qualifier—surrounds data from the cell. Text conversion has no limitation on data size.



7

Logging

7.1 Service Description

Application Framework supports JDK 1.4 logging API: *http://java.sun.com/j2se/1.4.1/docs/guide/util/logging/index.html*. Generic framework classes themselves use logging API to create a log of the initialization process and watch for some important internal events and exceptions. The generic framework logger is named `gov.fnal.controls.framework`. Developers should use their own loggers in applications.

Application Frameworks provides a way for logging API configuration, that replaces the native Java logging configuration through the configuration files. Logging configuration is stored as properties (§1). From the application, those properties may be changed in Logging Configuration Dialog.

Application Framework provides two additional logger handlers:

- `gov.fnal.controls.framework.logging.AppixHandler` sends logging messages to the APPiX database; these messages may be viewed at *http://www-bd.fnal.gov/appix/select/props*;
- `gov.fnal.controls.framework.logging.MessageViewerHandler` sends logging messages to a Message Viewer window.

Logging services are provided only for those applications, that are registered in APPiX database.

7.2 Logging Configuration

Initial logging configuration is defined in the Global Property File (see the table on the next page).

Table 11. Logging properties

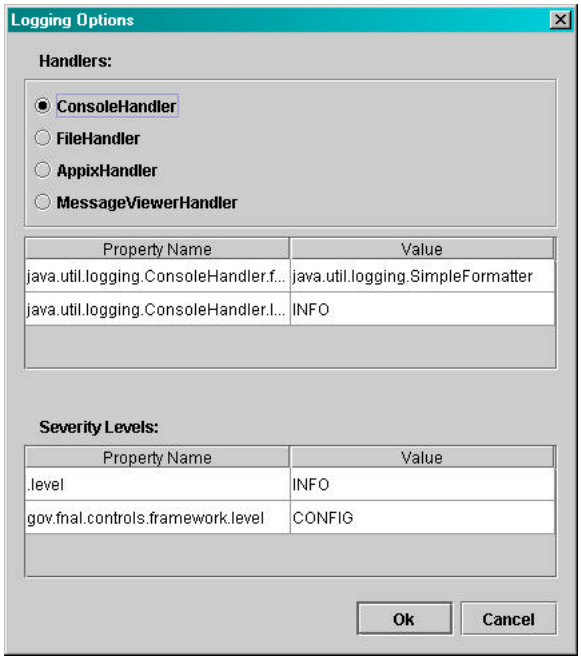
Property Name	*Default Value*	*Function*
handlers[1]	①.ConsoleHandler, ②.AppixHandler, ②.MessageViewerHandler	List of available handlers
protected.logging.handlers	①.FileHandler	List of logging handlers, that may not be altered through GUI
.level	INFO	Global default severity level for all
gov.fnal.controls.framework.level	CONFIG	Severity level for Framework loggers
①.FileHandler.level ①.FileHandler.pattern	OFF %h/java%u.log	FileHandler configuration
①.FileHandler.limit	50000	
①.FileHandler.count	1	
①.FileHandler.formatter	①.XMLFormatter	
①.ConsoleHandler.level	INFO	ConsoleHandler configuration
①.ConsoleHandler.formatter	①.XMLFormatter	
②.AppixHandler.level	WARNING	AppixHandler configuration[2]
②.MessageViewerHandler.level	OFF	ConsoleHandler configuration
②.MessageViewerHandler.formatter	①.SimpleFormatter	

① = java.util.logging
② = gov.fnal.controls.framework.logging

[1] FileHandler is now excluded from the default list of handlers, though its properties still exist.
It has been found, that unconditional usage of FileHandler may cause hanging on several systems, where user's home directory is located on a separated file system— even if the severity level for this handler is OFF.

[2] AppixHandler always uses XMLFormatter as a formatter.

To alter these default values, the application property file should be used. At the runtime, configuration of the particular logger handler may be changed in Logging Configuration Dialog. Use “Tools/Logging” menu item to open it.



The top panel shows the logging handler configuration; the values may be changed through the GUI (except of those for *protected logging handlers*). The bottom panel shows severity levels.

7.3 How To Use Logging

1. Create an instance of your logger. Choose an appropriate name for the logger. It is recommended to use the package name.
2. Set severity level for your logger using application property file. For example, if logger name is `gov.fnal.controls.applications.monitor`, add the following line in *<application_name>.properties* file:

```
gov.fnal.controls.applications.monitor.level = ALL
```

3. Every time, when you need to place a log message from your application, use some function of the logger (`log`, `severe`, `warning`, `info`, `config`, `fine`, `finer`, or `finest`).
4. You can change severity level by editing the application property file, or assigning new value through the logging configuration dialog.

Example:

```
package gov.fnal.controls.applications.monitor;

import java.util.logging.Logger;
import java.util.logging.Level;

private Logger logger = Logger.getLogger(
  "gov.fnal.controls.applications.monitor"
);

public class Test {

  public Test() {
    logger.log( Level.INFO, "Class is created" );
  }

  public void testLogger( String message ) {
    logger.severe( message );
    logger.warning( message );
    logger.info( message );
    logger.config( message );
    logger.fine( message );
    logger.finer( message );
    logger.finest( message );
  }

}
```

7.4 Heartbeat Service

Application Framework provides Heartbeat Service, that notifies the server every time when the application is started or terminated, and every 6 minutes during the execution. By default, this service is enabled. To switch it off, the property `framework.heartbeat` must be set to `0`. Heartbeat service is provided only for applications, registered in APPiX database.

Application usage log is available at: *http://www-bd.fnal.gov/appix/select/heartbeat*. List of live applications: *http://www-bd.fnal.gov/appix/select/live?select=1*.

Every application can have one of three statuses:

- **Working** — if start notification is received,
- **Terminated** — if end notification is received, or
- **Died** — if no notifications were received last 15 minutes.

8

Operation Locks

8.1 Service Description

Application lock service is designed to prevent concurrent execution on critical routines. Applications registered in Application Index are able to set a special system-wide named flag on the server. Before setting this flag, a server utility check whether it is already set by another program, and reject the current request, if so. All existing flags are linked to running program instances. Operation locks can be released explicitly, by request, or they are removed automatically upon the normal program termination or death.

8.2 Operation Locks API

In order to request an operation lock, the program should use `setOperationLock` abstract method of Application Manager. The argument of this method is a lock name, which is an arbitrary string. The method is either sets the requested lock on the server and finishes quietly, or it returns an exception. The cause of the failure can be:

- operation lock with the same name is already set by another program (multiple requests from one programs are Ok);
- the application is not registered in Application Index;
- lack of connection to the server.

Since the first situation is quite possible, the exception should be always handled in the program. Use the exception's message to find a plain-text explanation of reason.

To unlock an operation, use `releaseOperationLock` static method. It finishes successfully, even if the lock has not been set.

Example:

```
public void lockMyOperation() {
  try{
    ApplicationManager.setOperationLock( "foo" );
  } catch (Exception ex) {
    JControlsMessageBox.showError( ex );
  }
}

public void unlockMyOperation() {
  try{
    ApplicationManager.releaseOperationLock("foo" );
  } catch (Exception ex) {}
}
```

9

Reference Implementation

The class `gov.fnal.controls.examples.FrameworkExample` implements most of the Application Framework functions:

- Dynamically generated GUI;
- DAE Connection (with `DaeConnectionListener`);
- Operation locks;
- Application properties;
- Logging;
- Loading resources.

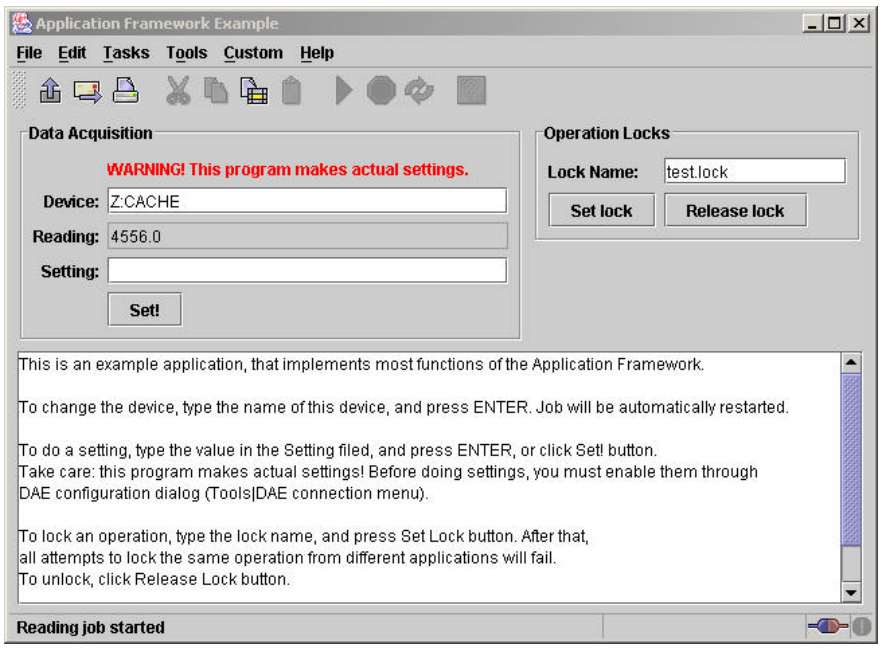
To start the example, use the references in Z directory of Application Index. The program may be launched as either STDALONE or JWS-AUTO type.

For the manual start, the program classpath must include `gov.jar`, `framework.jar`, `af-bunch.jar`, and `jconn2.jar`. On Windows console, type something like:

```
%JAVA_HOME%\bin\java -classpath p:/jars/gov.jar;↵
  p:/jars/split/framework.jar;↵
  p:/jars/multi/af-bunch.jar;p:/jars/jconn2.jar↵
  gov.fnal.controls.examples.FrameworkExample
```

Here, `p:` drive is mapped to *daesrv**java* directory.

The user interface of this program looks as follow:



At the startup, the application starts the DAQ job and shows current reading values for the specified device (Z:CACHE on the snapshot). If the device name is changed, the job is automatically restarted. In order to do the setting, a float value should be typed in “Setting” field. In case of Z:CACHE device, the setting value is immediately translated to the reading value, so it will appear in the “Reading” field. For all users, except of MCR, settings must be unlocked manually through DAE Connection dialog (“Tools | DAE Connection” menu).

The “Operation Locks” panel allows to test the operation lock service.

Appendix I. Release History

- **Release 2.3.36, 04/02/2003 - LAST VERSION**
  - Operation Lock service is implemented
- **Release 2.3.11, 10/29/2002**
  - Manual definition of user's home directory (for `GUIPersistencyManager`) added.
  - `FileHandler` excluded from the default handler list.
  - Self-determination procudure for "JWS-AUTO" applications is implemented.
- **Release 2.3.9, 10/21/2002**
  - Heartbeat service added.
- **Release 2.3.7, 10/15/2002**
  - `JTabbedPane` set enabled for capture utility.
  - `JSimpleControlsFrame` introduced.
- **Release 2.3.6, 10/10/2002**
  - Changes in system environment:
  - `xerxes.jar` removed from classpath
  - `jlfgr-1_0.jar` added to classpath
  - Menu bar changed
  - Status bar changed: connection and message viewer buttons added
  - Message Viewer created
  - DAE connection routine completely updated
  - Self-determination procedure updated, APPiX connection added
  - Two new logging handlers: `AppixHandler` and `MessageViewerHandler`
  - Capture utility update:
  - Last file name is stored between sessions
  - Printing options are stored between sessions

  - Print To Fit mode for big images
- **Release 2.2.3, 05/03/2002**
  - First productional version