# JIDE Desktop Application Framework Developer Guide

# Table of Contents

# What is JIDE Desktop Application Framework

JIDE Desktop Application Framework (JDAF) facilitates the creation of robust cross-platform desktop application using Java. JDAF will enable you to create quality, scalable and maintainable desktop applications using Java with unprecedented OS integration and end-user satisfaction.

We believe Java is the definitive technology for developing cross-platform desktop applications. But we also realize that there are some important realities to confront and issues to resolve so you the developer will be positioned for success with your next desktop solution. Below we discuss our motivations for this product.

## An Architectural Framework

In the web development community, development metaphors and architectures abound. For example Jakarta Struts for Web Applications. One columnist counted 50+ web architectures available to the developer at one point[1]. Yet on the desktop, there are precious few architectures. To a large extent, developers are left to reinvent the wheel for each desktop application solution. Why shouldn't desktop developers have an architecture that promotes good design and a proven structure for application development?

JIDE Desktop Application Framework responds by providing a cohesive development framework following the popular Model-View-Controller (MVC) paradigm. The MVC paradigm is well understood by developers and proven to be effective for producing low-coupled yet highly-cohesive object-oriented architectures.

## Base Functionality

Making a small frame-based Swing application is fairly straightforward. Even new Java developers can get there fairly quickly with the Java Tutorials. Most developers are comfortable with classes like JFrame and the normal palette of Swing Components. You may even be familiar with our JIDE family of Swing Components. But, things start to get out of hand very quickly when an application gets larger. For example, introducing the concept of multiple documents starts one down a path of requirements such as data management, menu wiring, dialogs negotiations, and window management. This behavior is expected by users but has little to do with core application functionality, yet it takes a significant amount of time and resources to accomplish. Compound this with the challenges of cross-platform deployment such as OS Guidelines integration and varying Java VM versions and implementations, and you suddenly

[1] Web Frameworks and IDE - Part 3, Yakov Fain, JDJ Oct. 2005, Vol.10, Issue10

have a non-trivial venture. Often, underestimating the cost of providing this functionality can cause release delays or compromises in quality and usability.

Various groups offer some helpful tools. JSR-296 Swing Application Framework looked promising by addressing the nominal needs of a Java application but has been abandon. On the other hand, there is the option to re-use solutions like Eclipse or NetBeans Platforms to gain benefits from their architectures. However, while these RCP platforms do provide base functionality, they were not originally intended as generic desktop solutions, but as "tools" platforms. As a result, applications inherit infrastructure that may either be overkill or ill-suited, and applications built on these platforms tend to look like the platform themselves, being less intuitive to general users. The learning-curve and complexity of working with these frameworks is also expensive due to a history entrenched in design requirements for complex development environments.

There remains the need for a generic yet capable desktop application development solution.

JIDE Desktop Application Framework delivers. It was designed from the ground up to be a platform for non-trivial desktop application development. It provides out-of-the-box application functionality. Full support for data management, view management, file handling, application-related Actions that handle single or multiple documents, editing, window management, printing and help integration, and even command line parsing and console application APIs. All this capability means that you can focus on your application requirements and let the framework do the rest.

## Better Desktop Integration

Java applications have historically wrestled with being well received on the desktop. One could argue that a "write-once, run everywhere" desktop application may have been counter-intuitive, and contributed to the lack of acceptance of Java on the desktop because user expectations on their particular desktop platform where not met.

Java developers are forced to make pragmatic decisions when choosing cross-platform strategies. Usually the result is a UI that reflects what the developer is accustomed to (or what the tools allow them to do), or perhaps what is most convenient and economical. Java purports taking care of cross-platform UI details via Swing, but often developers expect more from Swing that it was designed to do. These factors explain why Mac users are confronted with applications having a "main window" with an integrated menu bar at the top, and wonder what "Exit" is doing in the File menu, and why dialogs look so strange. Windows users wonder where the "Exit" command is and where the toolbars are, etc.

Swing has come a long way. Look and Feel fidelity continues to improve with each Java release. But there are many cross-platform issues beyond Swings ability to facilitate. Solely relying on Java API's (Swing or SWT) for application development will result in architectures that by default stand in stark defiance to certain prominent OS platform guidelines, hence breaking fundamental user expectations by design. The greater therefore is one of *cross-platform integration* and *user expectation*.

Each OS vendor provides *OS Guidelines* detailing what is expected of a desktop application on their platform. Guidelines are concerned with creating a consistent *application experience* for their users. While guidelines do specify some user interface conventions; button spacing, fonts, etc., most of the specifications are focused on application experience.

Guidelines are less concerned with how windows render, but instead with how windows represent documents, how they interact with other document windows, how they integrate with menus, what happens when they open and close, what dialogs are presented, and with what kind of information. Guidelines are less concerned with how menus render, but with what menus should be available, in what order, with what capabilities, and how they should act in various states of the application.

Native developers easily abide by these guidelines due to native tooling and know-how. AWT/Swing, even SWT are irrelevant to this point because "component-level" look and feel is expected by the user, and as Java developers we rely on Swing to deliver this. There is another world of look and feel that needs to be considered. We have identified this characteristic as *Application Look And Feel,* and believe it to be a fundamental requirement for modern, cross-platform Java desktop development, least we propagate the mistakes of the past.

JDAF supports this application look and feel concept by introducing a unique feature called a "Managed Application UI". This industry-first technology places the control of the desktop application's UI on the framework. This allows the Application UI to be adaptive to the platform so that OS guidelines can be respected. For each OS the Application UI emulates native OS document window presentation characteristics, provides recommended dialog presentations, negotiations with the file system, provides recommended menu and toolbar configurations, standard icon themes, and other unique OS-guidelines-recommended integration features and nuances. And like Swing Look And Feel, you get this functionality for free. The net effect is that the framework manages your UI automatically. You need only work at the Component-level.

With a "Managed Application UI" your product provides an intuitive, native-feeling application experience. Finally, you have a truly cross-platform solution!

## Exploring JIDE Desktop Application Framework

A desktop application, in its most abstract form, displays and/or manipulates data. The JIDE Desktop Application Framework (JDAF) facilitates this abstraction by using Model-View-Controller (MVC) architecture. A *DataModel* is the model, a *DataView* is the view, and the *DesktopApplication* is the controller. Put another way, a DesktopApplication controls DataModels and DataViews.

While the framework is extensive in capability, much of the functionality just works. Besides defining your DataModels and DataViews, using the API to create an OS-Guidelines compliant application can take as little as a few lines of code:

```
DesktopApplication application = new GUIApplication("My Application", MyDataModel.class, MyDataView.class);
application.run(args);
```

There is a wealth of configuration capabilities available before you call *run(),* but in a sense, it's that simple. Providing application-specific behavior is accomplished by attaching various listeners, customizers, Actions, factories and other service objects to your DesktopApplication instance.

Here is an example of creating an application knowing even less of JDAF, having only a Java component as a "view":

```
GUIApplication.run("My Application", MyPanel.class);
```

You may also subclass and configure the application it in the constructor.

The following sections will take you on a brief tour of the fundamental classes in JDAF. Later in the document we discuss creating particular types of applications and then migration strategies from existing codebases to JDAF.

Unless mentioned specifically, all configuration options happen against the application object, prior to calling *run().* When *run()* is called, the application lifecycle takes control of the JVM lifecycle.

## DesktopApplication

The DesktopApplication is the "controller" portion of the MVC architecture. It is the focal-point of your application. *DesktopApplication* is an abstract class that defines the desktop application lifecycle inside of your *main()* function.

There are two direct concrete subclasses; *ConsoleApplication* and *GUIApplication*. A third usability GUIApplication subclass is the *FileBasedApplication*, which facilitates a file or document-centric application. These classes may be used directly or extended; whichever is more convenient or appropriate. A number of listeners and service objects allow configuration of the application's behavior at runtime.

To start a DesktopApplication, you call *run(String[] args).* To terminate a DesktopApplication, you call *exit(),* though this is usually invoked from the UI.

### ApplicationLifecycleListener

To monitor and react to the basic lifecycle of a DesktopApplication instance, you may register an *ApplicationLifecycleListener*. This listener is called at critical times during the life of the application with the following notifications:

- APPLICATION_OPENING
- APPLICATION_OPENED

- APPLICATION_CLOSING
- APPLICATION_CLOSED

Each event method is passed an *ApplicationLifecycleEvent*, having a reference to the DesktopApplication instance. The APPLICATION_OPENING and APPLICATION_CLOSING event methods may throw an *ApplicationVetoException*, in which case the application is terminated or kept from terminating, respectively. There is an *ApplicationLifecycleAdapter* available as a convenience implementation.

The ApplicationLifecycleListener is a great place to perform general application tasks, because it straddles the application. For example:

- *Show a Splash Screen* - The `APPLICATION_OPENING` event preempts Swing class-loading, so a splash screen will open snappily, particularly if an AWT is used instead of a Swing class. `APPLICATION_OPENED` can then be used to close the splash screen.
- *Check Security* - Use the `APPLICATION_OPENING` event to do security tasks such as user license checking, LDAP authentication, configure application states based on environment, or even introduce your JIDE license information in an encapsulated way. This is also a good place to set an alternate look and feel if desired.
- *Initial GUI Behavior* - The `APPLICATION_OPENED` event is a great place to perform some default initial behavior. The GUI is loaded and ready to go. Show a welcome screen using *StandardDialog* from JIDE Dialogs or use the *TipOfTheDayDialog* component from JIDE Components to aid in application instruction.
- *Application Cleanup* - The `APPLICATION_CLOSING` or `APPLICATION_CLOSED` events are a great place to perform clean up chores such as storing layout data from the JIDE Docking Framework or storing other application state, saving preferences, removing temp files, or any other clean up functionality.

### Application Data

A DesktopApplication, as a Controller, needs to be able to originate and manage application models and views. Here are the primary controller methods for the application:

- *public DataModel newData(Object)* - introduces a new data into the application
- *public DataModel openData(Object)* – introduces an old data into the application
- *public void saveData(DataModel)* – persists a data
- *public void resetData(DataModel)* – re-opens a data
- *public void closeData(DataModel)* – removes the data from the application

The *newData()* and *openData()* methods use *criteria* to determine what kind of data to open in the application. The resulting data is represented as a *DataModel*. *DataViews* are created independently of the model in an implementation dependent manner. More detail on this is presented after we discuss the Model and View layers, or see *MVC Flow.*

## DataModel

To implement the *model* portion of our MVC architecture, we use the *DataModel* interface. Application data can take many forms. Data can be a unique object structure, effectively modeling the domain space, or a predefined asset, like character data, a byte image buffer, or some other formatted block, or a proxy to data, such as a JDBC ResultSet. DataModel makes any data suitable to the framework.

Put another way, DataModel is simply an application interface to your data, which can be any plain Java object. And you should add properties to your implementation as is appropriate to you data modeling needs.

A DataModel has a *name* and a *semantic* name property:

- *public String getName();*
- *public String getSemanticName();*

DataModel names must be unique for each DataModel in the DesktopApplication. The *name* is used as the unique readable identification of the data. The DesktopApplication ensures that the name is a unique when it is created for each DataModel. The *semantic* name is used to denote a “meta name” to the user, such as “Document”, “Project”, “Drawing”, etc. Semantic names do not have to be unique, providing a name-spacing level of identification for your models.

When a DataModel is originated from the DesktopApplication you may provide input of some kind by which to determine which DataModel to create. We call this the input *criteria*. This is stored in the created DataModel and may be accessed via.

- *public String getCriteria();*

A DataModel facilitates the concept of being *new,* and can be flagged as being *dirty*, which advertises its candidacy for persistence.

- *public void init(DesktopApplication);*
- *public boolean isNew();*
- *public boolean isDirty();*
- *public void setDirty(boolean);*

A DataModel contains an UndoManager that can be used to record UndoableEdits specific to the DataModel.

- *public UndoManager getUndoManager();*

DesktopApplication can support multiple DataModels at a time or different types. To access a DataModel from the DesktopApplication, use *getDataModel(name), getDataModel(Class)* or any of the other *getX()* methods. In a DesktopApplication only one DataModel should have the application "focus" at a time. This notion is facilitated by the DesktopApplication methods:

- *public DataModel getFocusedDataModel();*
- *public void setFocusedDataModel(DataModel dataModel);*

Use the DesktopApplication method *setAllowMultipleOpens(false)* to allow only one DataModel open at a time.

### Data Persistence

Most important to the DataModel interface are those that facilitate the origination and persistence of data. These are:

- *public void newData()*
- *public void openData() throws DataModelException;*
- *public void saveData() throws DataModelException;*
- *public void resetData() throws DataModelException;*
- *public void closeData()*

The DesktopApplication calls these methods automatically with primary DataModels during the application lifecycle. You may call them at any time.

### DataModel Implementations

*AbstractDataModel* is a root implementation of DataModel. It contains bound properties that are used by the framework, and you can register PropertyChangeListeners as well. The currently bound properties are:

- PROPERTY_NAME
- PROPERTY_SEMANTIC_NAME
- PROPERTY_CRITERIA
- PROPERTY_DIRTY
- PROPERTY_NEW

A basic concrete extension is *BasicDataModel,* which supports a single *data* property.

- *public Object getData();*
- *public void setData(Object);*
- *public void dataChanged();*

Setting the *data* property causes the PROPERTY_DATA property change event to be sent. The *dataChanged()* method may be called to notify listeners that the value in the *data* property has changed, without setting the entire property. This is useful when the data object is a complex object as opposed to a primitive value.

BasicDataModel has empty implementations of the data cycle methods, with the exception of the *newData*() method, which by default will populate the *data* property with the *criteria* property if the data property is null. This allows a simple pass-through of information from the application's origination of the DataModel when using *DesktopApplication.newData(criteria)*. You can override the other methods as you need.

Another concrete implementation is *FileDataModel.* It is a BasicDataModel subclass that reads and writes its data value to a file. FileDataModel supports a *File* and *FileFormat* to control data binding to and from files. With this class, the data cycle methods do *not* need to be implemented at all, since the file functionality is implemented.

## Data Origination

The DesktopApplication methods *newData(criteria)* and *openData(criteria)* "originate" DataModel instances. These methods are called either by the UI via the NewAction or OpenAction, or may be called programmatically. DataModel origination is facilitated by adding one or more *DataModelFactory* instances. DataModelFactory has one method, *createDataModel(Object):*

```
application.addDataModelFactory(new DataModelFactory() {
    public DataModel createDataModel(Object criteria) {
        // return a DataModel implementation
  }
});
```

The *createDataModel(Object)* method takes some *criteria* as an argument, intended to be used to determine whether to vend a DataModel. This criteria value will be stored in the DataModel under the *criteria* property.

When a request for data is made of the DesktopApplication, each DataModelFactory is consulted with the provided criteria until a non-null DataModel instance is returned. In this way, multiple data of potentially different types can be introduced to the DesktopApplication.

While it is trivial to create a DataModelFactory, we provide some functional implementations that remove the need to create one in most circumstances.

- *BasicDataModelFactory* - BasicDataModelFactory instantiates a single DataModel type based on a single criteria value. You may supply the criteria and a DataModel class, otherwise it creates a *BasicDataModel*. If the criteria value is not defined, it *always* creates a DataModel. BasicDataModelFactory is a good "default" factory for simple data cases.
- *FileDataModelFactory* - FileDataModelFactory should be used to support DataModels that marshal their data to and from the underlying file system. FileDataModelFactory requires that criteria be resolvable to a Java File or FileFormat and that a *FileFormat* be registered to accept support for such data. The File, along with the FileFormat, is associated with a *FileDataModel* instance. Opening, saving, and reverting are facilitated through the File/FileFormat relationship.

  Working with files is covered more thoroughly later in this document in the "Making a Document-Centric Application" section.
- *HashedDataModelFactory* - A DataModelFactory that allows creation of multiple criteria to DataModel class mappings via an internal Map. This provides a reusable factory multiple DataModels need to be returned based on the criteria.
- *MappedDataModelFactory* - A DataModelFactory that allows data *types* to be mapped to DataModel classes. This provides a reusable factory when the type of the criteria distinguishes the DataModel to originate.

FileDataModelFactory, MappedDataModelFactory, and HashedDataModelFactory all provide the ability to supply a *java.util.Map* of initialization values. The key value pairs of the map are applied to the bean properties of the DataModel immediately after construction. Otherwise, a resources file using the [classname.property] resource key pattern can also be used to initialize parameters on a newly created DataModel, provided that the DataModel is your own subclass and has a *resources.properties* bundle file in the same package.

All DataModelFactory objects can be accessed from the DesktopApplication using *getDataModelFactories()*, or a single DataModelFactory can be accessed by type, using the *getDataModelFactory(Class)* method.

## Primary and Secondary DataModels

JDAF understands two fundamental states of a DataModel; *primary* and *secondary*. This is denoted at runtime by the *isPrimary()* method. If *isPrimary()* is true, the DataModel is considered a *primary* DataModel. This means it is focusable in the application. Only one focusable DataModel may be set in a DesktopApplication at a given time. This DataModel is accessible using the *getFocusedModel()* and *setFocusedModel()* methods. Focusing a DataModel also focuses its corresponding DataView. Hence we consider the DataView "primary" as well, for the sake of discussion. In a GUIApplication, primary DataViews are installed automatically by JDAF according to the application style, and the standard GUIApplicationActions, installed in the menubar and toolbar automatically, operate on notion of the primary models and views. For example, when "Save" is invoked, the primary DataModel is the target for the save.

Conversely, if *isPrimary()* returns *false*, the DataModel is considered a *secondary* DataModel. This means it is not focusable and therefore not automatically recognized by the DesktopApplication. Secondary DataViews are not handled by the managed UI and are not acknowledged in the managed data lifecycle. However, this is not a lesser status, just different. The benefits are many. First, you can orchestrate many DataModels and DataVews in a window in combination with a primary DataModels and DataViews. This is perfect for "docked" or "split" views or really any UI you want to include in the MVC. Also, as the developer you have total control over the DataModel lifecycle. You can tell it when to open, close, and save[2]. Finally, while JDAF manages the association between DataModel and DataView, in the context of GUIApplication, you control how the DataView is installed into the window. This is done by installing a *DataViewHandler. (*See *Making a GUI Application* for more details.)

The primary status of a DataModel shouldn't change during its lifetime. That is, primary DataModels should always return *true* from isPrimary() and secondaries should always return *false*. There are two default DataModel implementations that support secondary status by default: *SecondaryBasicDataModel* and *SecondaryFileDataModel*. These are similar in all ways to BasicDataModel and FileDataModel accept in their primary status is false.

## Monitoring the Data Lifecycle

Since a DesktopApplication manages one or more DataModels, it may be useful to be notified of DataModel activity. A *DataModelListener* can be added to the DesktopApplication to receive DataModel events. The *DataModelEvent* provides access to the DesktopApplication and the DataModel, and in some cases allows control flow based on the event. The events are as follows:

- *DATA_MODEL_INITIALIZED*

[2] GUIAppication does provide an *autoSaveSecondaryDataModels* property that called *saveData()* when secondary DataViews are closed.

- *DATA_MODEL_OPENING*
- *DATA_MODEL_OPENED*
- *DATA_MODEL_RESETTING*
- *DATA_MODEL_RESET*
- *DATA_MODEL_SAVING*
- *DATA_MODEL_SAVED*
- *DATA_MODEL_CLOSING*
- *DATA_MODEL_CLOSED*

As you can see, all but the *DATA_MODEL_INITIALIZED* event, which is a notification of *new* data, provides a pre/post operation opportunity. In all pre-operation methods (ending with 'ing), an *ApplicationVetoException* may be thrown to signify whether the action should be aborted.

A *DataModelAdapter* is available as a convenience implementation.

## DataView

To facilitate the *view* portion of the MVC architecture, we introduce the interface *DataView*. DataView defines how the user visually experiences and interacts with a given DataModel. For every DataModel created, there will be a corresponding DataView. The DesktopApplication manages this association of DataModel/DataView pairs.[3]

DataView is primarily concerned with facilitating the display and manipulation of a DataModel. It has four methods:

- *init(DesktopApplication)*
- *getViewComponent()*
- *updateView(DataModel) throws DataModelException*
- *updateModel(DataModel) throws DataModelException*

The *init()* method is called when the DataView is instantiated from a DataViewFactory. It allows the DataView to initialize itself and reference the application object. The *getViewComponent()* method allows

[3] A common vernacular for this DataModel/DataView combination would be a "*Document*". We have intentionally avoided this terminology for multiple reasons; to circumvent confusion with Java Text API Document, the W3C DOM Document, and our own DocumentComponent API. Furthermore, the term Document is very specific to a file-based application. While this type of application is supported, JDAF is significantly more abstract. Not only "Document" can be conceived but other more distributed models may be realized in the future, that otherwise would break the "Document" metaphor.

the interface to return an object that is suitable to the DesktopApplication implementation. For example, in a GUIApplication this is a *java.awt.Component*.

The *updateView()* method is called whenever the DataModel is new, opened or reset. This allows the DataView to reflect the state of the DataModel. This method may be called manually whenever the view needs to refreshed.

The *updateModel()* method is called when the DataModel is about to be saved to allow the DataView to update the state of the DataModel first. This method may be called manually whenever the model needs to be refreshed from the view.

### DataViewFactory

DataViews are dynamically introduced into the DesktopApplication using a *DataViewFactory*. The *createDataView(DataModel)* method takes a DataModel as a parameter. The idea is to let the factory decide which view would provide the visual interface to the model.

```
application.addDataViewFactory(new DataViewFactory() {
    public DataModel createDataView(DataModel data) {
        // return a DataView implementation
    }
});
```

The visual and interactive nature of the view is highly application dependent. It is up to the developer to develop these views and add factories that vend appropriate DataViews for a given DataModel in the DesktopApplication.

### *Provided DataViewFactory Objects*

JDAF comes with some convenience and application-specific DataViewFactory implementations:

- *BasicDataViewFactory* – This factory vends a DataView implementation depending on the Class you provide, irrespective of the DataModel. If no class is provided it vends a *DataViewPane* implementation.
- *BasicComponentViewFactory* – This factory vends an anonymous DataView by wrapping a standard JComponent.
- *MappedDataViewFactory* – This factory vends a DataView implementation based on the DataModel Class mapping you define. Convenient for multiple DataModel type implementations.
- *FileDataViewFactory* – This factory only responds to FileDataModels and vends a DataView based on a Class-mapping that associates FileFormats to DataViews. See the section “Making a Document-Centric Application” later in this document.

- *SemanticNameDataViewFactory* – This factory vends a DataView implementation based on the semantic name of the input DataModel. Convenient for multiple DataModel type implementations.
- *ConsoleViewFactory* – This factory is only supported by the ConsoleApplication, which is preconfigured with said factory. The Console Application API is covered more thoroughly later in this document.

All DataViewFactory objects can be accessed from the DesktopApplication using *getDataViewFactories()*, or a single DataViewFactory can be accessed by type, using the *getDataViewFactory(Class)* method.

### Multiple DataViews

In a GUIApplication, multiple DataViews may be associated to a single DataModel. This is accomplished by providing multiple DataViewFactory implementations dedicated to producing a single unique DataView for the same DataModel instance. If the DataModel is primary, the first DataView returned is considered the primary DataView and will be installed automatically in accordance with the application style. Any following DataViews are considered *auxiliary* views and, like secondary DataViews, require a *DataViewHandler* so that you may determine where in the UI to install the secondary DataViews in the UI. In order for this capability to work you must set the *setAllowMultipleDataViewsPerModel()*.

### Provided DataViews

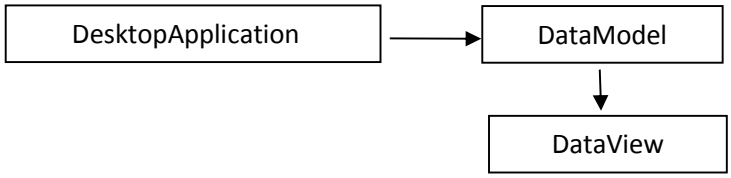
JDAF comes with some basic DataView implementations primarily to support the two fundamental application types:

- *DataViewPane* –a JPanel subclass. It may be subclassed to create a DataView for a GUIApplication or populated using a DataViewListener when using the BasicDataViewFactory without a class argument. Subclasses should use *initializeComponents()* to install Swing Components. (Note: Windowing is managed by the application automatically). The *updateModel()* and *updateView()* methods are used to synchronize the DataModel and DataView states, and are called automatically during the application lifecycle. DataViewPane provides other convenience facilities, like access to the GUIApplication, facilities for “dirtying” the associated DataModel, adding UndoableEdits to the DataModels UndoManager, and interfacing with the Edit Menu.
- *ConsoleView* - the lone DataView implementation used by ConsoleApplication. It is created automatically with each DataModel. ConsoleView is a virtual view behind the actual command prompt. It provides many typed read and write methods and interfaces with JIDE ObjectConverters to manage String-to-Object conversions. This class is integral part of the Console Application API.

## MVC Flow

Now that we have introduced the DesktopApplication (Controller), DataModel (Model), and DataView (View), it seems prudent to summarize how the DesktopApplication MVC architecture operates, just to be clear.

Concerning model origination, consider the following conceptual diagram:




The following basic sequence occurs:

1. The controller methods *newData(criteria)* or *openData(criteria)* originate a DataModel via a DataModelFactory using input criteria.
2. The DataModel *init(DesktopApplication)* method is called and criteria is bound to the DataModel.
3. The DataModel *newData()* or *openData()* methods are called, respectively.
4. The DataModel is used as an argument to originate as many DataViews via DataViewFactories. This results in a pair of DataModel/DataView(s).
5. The DataView(s) *init(DesktopApplication)* method is called.
6. The DataView(s) *updateView(DataModel)* method is called to update the view with the model.

During this cycle, the respective DataModel and DataView events are fired. There will always be a DataModel/DataView(s) pair. But there may be many DataModel/DataView pairs, of the same or different types, at any time. DesktopApplication contains *getX()* methods for retrieving these model and view objects in various ways. For example, *getDataModel(DataView)* and *getDataView(DataModel)* provide for this association.

### Data Lifecycle

A DesktopApplication needs to be able to create, load, save, reset, and close the application data. We call this the *Data Lifecycle.* The DesktopApplication manages the data lifecycle through five methods. DesktopApplication delegates calls to DataModels and DataViews as appropriate to the application implementation. Below is a table of the associated behavior:

| **DesktopApplication (Controller)** | **DataModel (Model)** | **DataView (View)** |
|---|---|---|
| newData(Object criteria) | newData() | updateView(DataModel) |

| | | |
|---|---|---|
| openData(Object criteria) | openData() | updateView(DataModel) |
| saveData(DataModel); | saveData(); | *updateModel(DataModel) |
| resetData(DataModel); | resetData(); | updateView(DataModel) |
| closeData((DataModel); | closeData(); | N/A |
| printData(DataModel) | [Implement PrintSource] | [Implement PrintSource] |

- DataView.updateModel(DataModel) is called *before* DataModel.saveData().

## Printing

To facilitate printing, the DesktopApplication uses a *Printer* object. Printer encapsulates the workflow of printing in Java. In order to use Printer, it must be supplied with *java.awt.Pageable* object.

When *printData(DataModel)* is called, the focused DataModel and DataView are examined to see if they are a *PrintSource*. If a PrintSource is resolvable, it is used to vend a Pageable instance, and the instance is set into the Printer to begin printing. By default *DataViewPane* implements the *PrintSource* interface. Simply provide the *Pageable* in the *getPageable()* method to enable printing.

The subject of Java Printing is outside the scope of this document, but JDAF provides one Pageable implementation called *ComponentPageable* which can be used to print a Swing component. This is a very minimal implementation for on-off light-weight work. It is up the developer to control pagination, for example. But it may suffice for a quick printing solution. We have planned to provide more robust output architecture in the future. It may also be possible to provide PrintSource interfaces to existing printing products.

In the DesktopApplication, Printer can be disabled by calling *setEnabled(false)*, or removing the Printer object altogether via *setPrinter(null)*. This should be done before a call to *DesktopApplication.run()*. In a GUIApplication, the later will cause the print-related items to be completely removed from the menus and toolbars. The former will cause them to become disabled. The absence of a *PrintSource* will also disable them.

## CommandLine

CommandLine implements a Singleton[4] that stores Java program arguments from the application *main()* method, and provides universal access to the arguments along with parsing capabilities. CommandLine can be used two ways; explicitly and implicitly. Explicit use of CommandLine means to capture the arguments of the main method using the class itself:

[4] Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley Professional Computing Series). http://mahemoff.com/paper/software/learningGoFPatterns/

```
public static void main(String[] args) {
    CommandLine.set(args);
    // create application…etc
}
```

The *CommandLine.set()* method can only be called once. Other calls are ignored. It parses the command line and creates a static CommandLine instance.

Implicit capture of the CommandLine is achieved by calling the *run(String[] args)* signature method of a DesktopApplication, like so:

```
public static void main(String[] args) {
    GUIApplication application = new GUIApplication("My App");
    application.run(args);
}
```

The *run(args)* method internally captures the CommandLine. With either method, CommandLine can be accessed using *CommandLine.get()* for the life of the JVM. CommandLine inherits all the parsing capabilities of CommandString, so the command string can be easily extracted, pattern matched, type converted, or checked for switches.

### CommandLine and Object Conversion

CommandString and hence CommandLine use the *ObjectConverter* API for converting Strings to Objects. It implements *ConverterContextSupport*, so the developer can set a custom library of ObjectConverters to use during the parsing of arguments.

DesktopApplication has a ConverterContext that can be used when parsing arguments from the CommandLine. This ConverterContext can be used by developers to provide access to the application instance during conversions. It is available via *getConverterContext*() from a given DesktopApplication. *Note, removing this ConverterContext is not recommended.*

## Environment Variables

For years the *System.getenv()* method had been deprecated and hard-blocked by an Exception, preventing developers from gaining programmatic access to environment variables. While Java 1.5 brought the feature back, now there is a function gap if trying to develop for "1.4 and above".

The Environment class solves this issue by providing a unified mechanism for accessing desktop environment variables on any 4+ Java2SE platforms. Environment is a Singleton. To access an environment variable use:

```
String var = Environment.getInstance().getVariable("myVariable");
```

## Actions

A DesktopApplication has an ActionMap so that application-specific functionality can be associated with the application. The ActionMap is actually an *ApplicationActionMap*. This map supports *PropertyChangeListeners*, so that Action additions and removals can be tracked.

*ApplicationAction* is an Action interface extension that simply adds an APPLICATION_KEY property, which is used to reference the DesktopApplication instance in the Action. When any Action is added to an ApplicationActionMap, the DesktopApplication instance will be referenced in the Action with this key so that it is available when *actionPerformed()* is called.

*AbstractApplicationAction* is a default implementation of the ApplicationAction interface. It additionally provides the methods *install()* and *uninstall()* which are called when the application instance is referenced to the Action. This provides a hook for behavior such as wiring or formatting.

In a GUIApplication, a GUIApplicationActionMap is used. This ActionMap facilitates the binding of resources to Actions based on their action key. The *ActionKeys* interface defines common menu and toolbar commands. By registering an Action using one of these keys, the Action will be bound with resources that reflect OS defaults such as names, icons, accelerators, and mnemonics. This behavior is facilitated by the *ActionResourcesBinding*. To change resource binding behavior, your Action may implement the *Resourcesful* interface and return its own ResourcesBinding object. See the section "Working with Actions" for more information.

*GUIApplicationAction* is best suited for a GUIApplication. It introduces the LARGE_ICON property and provides some beneficial access methods and some threaded behavior for better perceived performance.

*ConsoleCommand* is an ApplicationAction used by the ConsoleApplication to implement commands in the console.

JDAF comes with a complete set of Actions that provide base functionality both for GUIApplications and ConsoleApplications.

## Threaded Activities

GUIApplication provides a powerful mechanism for executing robust threaded tasks via the *ActivityManager*. Each GUIApplication has an *ActivityManager*. To access it, use *getActivityManager()*. ActivityManager provides a context for threaded execution using an *Activity* class. This API provides monitoring functions as well. You may query the ActivityManager for currently running threads using *getActivities()* and *isActivityRunning()*.

## Creating an Activity

To create a threaded task, subclass *Activity*. Activity represents a single job or work, and is as simple to implement as an Action. Simply implement *activityPerformed()*. In this method do a slice of work, as you would inside a loop. Activity manages all the details.

To run an Activity, use the ActivityManager.

```
application.getActivityManager().run(new Activity("My Activity", 100, 10) {
  public void activityPerformed() {
    // do a slice of work here
  }
});
```

Activities have a *name* by which they are identified in the ActivityManager. This name must be unique. If you do not define a name, the ActivityManager will name the Activity for you. The ActivityManager will also ensure Activity names are unique. The constructors allow you to specify the name, number of steps/iterations, and sleep time.

Use INDETERMINATE_STEPS for the sleep time to specify an indeterminate task or blocking activity. Use INFINITE_STEPS to specify execution with an unknown number fo steps. Alternately you may use the constructors that have no input for number of work steps. In this case, *activityPerformed()* is called once, and the activity finishes when the method completes. You can call *finish()* to normally quit the Activity before the steps are complete. Any exception in *activityPerformed()* will cause *cancel()* to be called safely (after activityPerformed() completes), though it may be called manually to abort the Activity from inside or outside the Activity, at any time. To keep an Activity from being canceled, use *setCanCancel(false).* This simply advertises that the Activity should not be canceled for the benefit of listeners. It does not swallow Exceptions.

To react to competed status you may implement *activityFinished()* and/or *activityCanceled()*. These are called on termination of the thread.

The *getStepsComplete()* gives you the current iteration. You can call *setStatusMessage()* and *setSecondaryStatusMessage()* as well to provide additional information to listeners. The application is accessible from the Activity using *getApplication().* Activity supports user properties, similar to Actions, as well and many more controls over the threaded behavior. See javadoc.

## Activity Progress

Activities should never invoke the AWT Event Thread. Activity supports *ProgressListeners* to report feedback to the GUI instead. By registering one or more ProgressListeners you can provide feedback to your GUI application in a thread-safe way. This creates a separation, like a Model/View, in the design of your threaded activities.

JDAF provides many powerful *ProgressListener* implementations that provide GUI feedback and blocking capabilities so that you achieve immediate productivity. Simply add one or more of these to your Activity as appropriate using *addProgressListener()*.

- *ActionBlock* - Block one or more Actions.
- *ComponentBlock* - Block one or more Components or Containers.
- *GlassPaneBlock* - Use to block DataViews and Windows.
- *DialogBlock* - Use to block using a dialog. Presents a Progress dialog by default.
- *ProgressFeedback* - Use to provide feedback in the UI. It supports expressing progress information to certain compatible Components, like labels and progress bars. Just set the name of a compatible Component (see javadoc) in your DataView with the same name as used in the ProgressFeedback constructor, and that component will reflect the progress of the Activity.

It is also simple to create your own ProgressListeners. There are three methods to implement:

- *progressStart()*
- *progressProgressing()*
- *progressEnd()*

Each method receives a ProgressEvent with the state of the progress and source. In this case, the source is the Activity.

Alternately, if your task is specialized enough for reuse, you can use the *GUIActivty*, which is a ProgressListener to itself, encapsulating both the threaded logic and GUI feedback in a single object. Just be mindful that this implementation is shared between the worker thread and the AWT Event Thread; i.e. do non-UI work in the Activty methods and UI work in the ProgressListener methods only.

You may also add global ProgressListeners to the ActivityManager. These will listen to all running Activity objects. These can be attached at any time using *addProgressListener()*. Note, when using the ProgressFeedback globally, be sure to use the name-based component binding method instead of explicitly defining a Component object, as this is more robust for a multi-window application.

## Activity Actions

An Activity can be fired from an Action using *ActivityAction.* ActivityAction will automatically block the Action until the task completes. This is handy in menus and in buttons.

```
Activity activity = new Activity("Try Me", 100, 10) {
  public void activityPerformed() {
   // do a slice of work here
  }
});
```

```
Action action = new ActivityAction(activity);
JButton button = new JButton(action);
```

An ActivityAction assumes the name of the Activity by default and automatically adds an ActionBlock to the Activity to control the enable state of the button or menu item.

## Custom Events

A DesktopApplication provides a centralized bus for creating a robust, cooperative, and maintainable messaging infrastructure for your application. This is done with the *EventManager*. To access it, use *getEventManager()*. EventManager facilitates decoupled communication between user controllers, models, views or any other object in your application.

EventManager supports a "publish and subscribe" model for event dispatch. A source object uses the manager to publish an event, and one or more recipients may subscribe to receive the event, which comes in the form of a *SubscriberEvent*.

Facilitating an event is very simple. For the EventManager to acknowledge an event, you must first add a unique event, like so:

```
EventManager eventManager = application.getEventManager();
eventManger.addEvent("MyEvent");
```

Events are simply Strings and must be unique. Objects can “subscribe” to receive this event by implementing one of the EventSubscriber interfaces, which has one method, *doEvent(SubscriberEvent event).* To subscribe to an event you do this:

```
eventManager.subscribe("MyEvent", eventSubscriber);
```

To send the event, you use your source object and the event like so:

```
eventManager.publish("MyEvent", source);
```

The actual sending of the event is accomplished by an *EventContext* internally. There is a DefaultEventContext which provides bifurcated (forked) event notification to subscribers depending on the EventSubscriber interface implemented. If your subscriber simply implements *EventSubscriber*, it will receive events on the publishing thread. If your subscriber implements *ThreadedEventSubscriber*, the event is received on a separate thread than the published thread. If your subscriber implements *GUIEventSubscrber*, the event is received on the AWT event thread. To facilitate a different type of publishing model, you may implement an EventContext and associate it with an event type when it is added:

```
EventManager eventManager = application.getEventManager();
eventManger.addEvent("MyEvent", new MyEventContext());

// or later with

eventManger.setEventContext("MyEvent", new MyEventContext());
```

The EventContext is sent with the SubscriberEvent. JDAF provides a few EventContext implementations:

| Class | Description |
|---|---|
| DefaultEventContext | Used by default if no EventContext is supplied when the event is registered. Bifurcates events based on subscriber implementation. Supports a userObject that can facilitate additional state information to subscribers. |
| GUIEventContext | All events are sent on the Swing Event thread. |
| FocusedWindowGUIEventContext | A GUIEventContext that only publishes events to JComponent subscribers in the front-most window. |
| FocusedDataViewEventContext | A GUIEventContext that only sends to the focused DataView, and optionally secondary DataViews in the front-most window |
| FocusedDataModelEventContext | Only sends to the focused DataModel |

Subscribers may subscribe and unsubscribe to single events or all events, or just events in a given EventContext class. Note that while there are *unsubscribe()* methods, the EventManager uses WeakReferences for subscribers, so there is no need to be concerned with explicit unsubscription for the sake of garbage collection. To completely clean the EventManager call *purge()*.

## Preferences

A DesktopApplication uses the Preferences API to manage application preferences internally. The *getPreferences()* method returns a java.util.Preferences node. The node is based on your application name

thus ensuring that multiple uses of the framework produce a unique node on a given platform. While this is a convenience feature, you may or may not desire to use this mechanism for your application.

An alternative is to manage your own preferences using a Properties file or other file type. The DesktopApplication method *getDataDirectory()* is useful in this regard as it will return the platform-appropriate location for storing such files based on OS conventions. This directory may be used for other purposes such as general session data or layout data.

In either case, JDAF does provide a facility for a Preferences dialog. Please see the section *Working with Dialogs*.

## Help

A DesktopApplication facilitates help integration via the interface *HelpSource*. A HelpSource implementation is set via *setHelp(HelpSource)*. The HelpSource interface provides a simple hook for executing your help system:

- *openHelp(Object hint)*

The hint argument is intended to influence the way the help is presented. HelpService defines some default constants that should be supported in an implementation:

- TOC_TOPIC
- INDEX_TOPIC
- SEARCH_TOPIC
- TUTORIAL_TOPIC

ConsoleApplication uses this to provide help for ConsoleCommands. GUIApplication provides a means for providing help throughout the UI using the *HelpSource.HELP_HINT_PROPERTY_NAME* value. A particular help topic can be defined like so:

1. In a *HelpAction* set the topic property. When no windows are open or no other help is defined, this topic is used. It is set to TOC_TOPIC by default.
2. As a client property in a DataView component. Use the *HelpSource.HELP_HINT_PROPERTY_NAME* as the property key and the topic as the value.
3. In dialogs, in the *DialogOptions* of a *DialogRequest* add a help button and then put the topic as a value in the *DialogOptions* using *put(HelpSource.HELP_HINT_PROPERTY_NAME, "topic")*.

For dialogs the help button will trigger the HelpSource with the topic. Otherwise, If the focused DataView topic is set, via the client property, it will be used. Finally, the topic in the *HelpAction* will be used.

The OS platform-specific help key will trigger the HelpAction or the HelpSource directly. This platform default key detection can be changed by implementing a HelpKey subclass and overriding *detect()*. To install the custom HelpKey, call *setHelpKey()* on the ApplicationMenuBarsUI object of the AppplicationUIManager.

## Versioning

A DesktopApplication can maintain a displayable version string.

*public String getVersion ();*

*public void setVersion (String version);*

The default value is "1.0". While this is a marginal treatment of application versioning, the value is used under some circumstances such as a default "About" dialog and an introduction prompt for a console application. In both cases, the behavior can be customized, so feel free to use or ignore this facility as you see fit.

## Vendor

A DesktopApplication can maintain a displayable vendor string.

```
public String getVendor();
public void setVendor(String version);
```

This value may be null or used to set the vendor of the application. This value is used in the default "About" dialog, but more importantly it is used on some platforms in the *dataDirectory* path (see *getDataDirectory()* ).

## Localization

It is common practice for desktop development to store String resources in properties files for localization. To provide some additional power and ease to working with resources, we have provided a powerful but simple resource management system through the *Resources* class.

To access String resources you use the *Resources* class like so:

```
String string = Resources.getString(getClass() , "key");
```

The Class defines the root location of the bundle.  Other signatures provide access by bundle name alone.

Without specifying the bundle name, JDAF uses the default name "resources" by default. Therefore files are called "resources.properties". This can be globally changed by using the System property:

```
-Djdaf.bundle.name
```

Resources is a factory for *Resources* objects. Each Resources object represents a root bundle at a particular package location. Resources contains typed access methods for primitive conversions and provides String-to-Object conversions using the *ObjectConverter* API.

You can also use a powerful formatting object called *ObjectFormat.* ObjectFormat can be an alternative to MessageFormat. It facilitates much more readable resource strings by using a simple but powerful expression language. You can mix-in MessageFormat syntax, also. Resources supports reading Strings using this UL-syntax or a custom syntax that you configure in the ObjectFormat. Resources also can "fold" resources on one another allowing resource Strings to reference other resource Strings in a bundle file. See the section "Object Format" towards the end of the document for UL-syntax examples.

Resources provides a powerful feature called "Polymorphic Resource Binding". This feature allows resources to be loaded into an Object based on a naming convention in the bundle. This is a great productivity tool for localization. Instead of initializing fields in the Object with calls to resource bundles, the fields can be bound to resources via a call to *Resources.bind(object)*. If the object obeys simple bean-naming conventions, resources in the form `[simpleClassName.fieldName]` will be set into the object. This includes detecting super class resources stretching over other bundles in differing packages.

Bean binding is but one kind of binding mechanism. Resource binding is flexible and extensible using the *ResourcesBinding* interface. A second signature allows for this object to be passed in to facilitate the binding mechanism. For example, resources can also be bound to Actions and Maps using the *ActionResourcesBinding* and the *MapResourcesBinding*, respectively.

### Localizing Basic Application properties

If you create a resources file (resources.properties) at the root of your application subclass, you can define the default properties of the application. Let us say the application subclass is called MyApplication. The properties file would look like this:

```
MyApplication.name="My Application"
MyApplication.version="1.0"
MyApplication.vendor="My Company"
```

If you use the application instance instead, for example GUIApplication, the resource file needs to be at the system root level (src root or jar root) and the file would look like:

```
GUIApplication.name="My Application"
GUIApplication.version="1.0"
GUIApplication.vendor="My Company"
```

When using resources in this manner, use the no-name constructor of a DesktopApplication, or pass null as the name.

### OS-Extended Variants

Since one focus of JDAF is convincing cross-platform application presentation and behavior, Resources must be OS-sensitive. For true cross-platform development there is an additional challenge not only of language support, but of OS guidelines. For example, internally JDAF supports OS-specific menu names, buttons names, icons, dialog messages, and more.

The Resource Bundle API could support this concept of OS/Platforms through its "variant" bundle name qualifier. However, there are no standards as to how platform variants are expressed, and more seriously the variant is the last qualifier in the bundle lookup chain, making it impossible to provide no-variant, non-location or default resource files that are OS-specific. To this end JDAF supports an extension to the resource mechanism, we call it the *OS-extended-variants* . You can place the OS variant on the end of any resource bundle file name, including after the normal variant, and it will respect the OS/platform. Given the OS-Variant WIN, all the following bundle names are legal:

resources_WIN.properties

resources_en_WIN.properties

resources_en_US_WIN.properties

resources_en_US_NewYork_WIN.properties

resources.properties

You may define the bundle name without the extended OS variant and it will operate as normal, irrespective of OS-platform.

Since JDAF supports certain OS/platforms via its Managed UI, the name of the properties files must match the respective variants supported by JDAF. To aid the developer in providing these files, Resources provides the static method *createResourceFiles()*. This method will create or update a set of resource files for a given Locale for each supported OS-extended-variants in JDAF. This ensures that Resources is synchronous with the active Application UI. Multiple calls to this method are non-destructive, so it can be called more than once to add support for new variants as OS-guideline support is added.

## Making A GUI Application

GUIApplication provides a robust application framework for the GUI desktop application. It provides a baseline user interface for you, automatically managing document windowing, standard dialog negotiations, standard menus, icons, and toolbars. JDAF supports the underlying platform via the *ApplicationUIManager* classes. Each manager class is written to emulate the respective OS-guideline

specifications. This feature imparts an intuitive, native feel to your application and additionally relieves you of the boiler-plate overhead usually associated with GUI application development.

## Application UI Technology

Before we get into developing a GUI Application, let us touch on the "Managed UI" feature. This feature automatically provides an OS-specific, guidelines-compliant application UI to manage your models and views, but you may find yourself interfacing with it at times, particularly if setting OS-specific options.

The call to *GUIApplication run()* will eminently start the Application UI. (The UI is started on the event thread, so is unnecessary to place *run()* on the Event thread.) The UI is managed by an *ApplicationUIManager.* Each supported OS has a respective ApplicationUIManager implementation. An *ApplicationUIManagerFactory* selects the appropriate ApplicationUIManager implementation, depending on the underlying OS, during construction of the GUIApplication instance. The *ApplicationUIManager.getVariant()* method in ApplicationUIManager determines which OS that the UI will facilitate. The constants are currently:

- JAVA_CROSS_PLATFORM
- LINUX_GNOME
- LINUX_KDE
- MAC_OS_X
- WINDOWS_XP

As Application UI support is added, the constants will grow.

While the System Properties identifies the OS via "os.name", as can our own *SystemInfo* class, the JDAF Application UI has specific OS-support and attempts to choose the most appropriate Look and Feel to achieve the best possible OS integration. In other words, these two technologies (JDAF and Look and Feel API) are working together to try to achieve the highest-degree of believability on a given OS/platform.

With this in mind, to make UI-based decisions on a given platform you should rely on the constants returned from the installed ApplicationUIManager *getVariant()* method. This also concerns variants in resource bundles as well. There are no standards for variants in the Resource Bundle API, so using JDAF variants is important (Use the *Resources.createResourceFiles()* to generate default bundle files with correct OS variants).

You may also use the *OSApplicationCustomizer* to make clean platform coding decisions that are driven from this logic.

## Application UI Highlights

While not all platforms are directly supported yet, we endeavor to introduce additional OS support in future updates. If an OS is not directly supported by the framework we use the *Java Cross-Platform* Application UI, which implements the Sun Java Look and Feel Guidelines.

Below we highlight what guidelines-recommended features can be expected when your application runs on a supported OS/platform.

### Java Cross-Platform Highlights

Guidelines for a Cross-Platform Java application are described in detail by the document Java Look and Feel Design Guidelines by Sun Microsystems. JDAF uses the CrossPlatformApplicationUIManager class to support features from that specification. Following are some highlights:

- Guidelines-compliant Dialog Presentation
- SDI Windowing Interface
- Compliant Window Titling
- API to the Java Look and Feel Graphics Repository *
- Large and Small Toolbar implementation
- Recommended Menu and Toolbar Items

*In order for this Application UI to work, the *Java Look and Feel Graphics Repository* jar (jlfgr1_0.jar) should be in the Java classpath. This jar can be downloaded from Sun at http://java.sun.com/developer/techDocs/hi/repository/. The Icons from this repository can be accessed using the *CrossPlatformIconsFactory*, though the managed UI uses them automatically.

A System property option is available that will force JDAF to use the CrossPlatformApplicationUIManager on *all* OS platforms. Use the System property “usecp=true”. While this option somewhat defeats the purpose of the application look and feel feature, it may be desirable for those who require a consistent UI, for perhaps corporate reasons. For general or commercial development, however, we recommend this not be done.

### GNU/Linux Highlights (Gnome)

Guidelines for GNU/Linux (Gnome) applications are described in detail by the web publication Gnome Human User Interface Guidelines. JDAF uses the LinuxGnomeApplicationUIManager class to support features from that specification and observances from other popular Gnome software, when this application UI is used. Following are some highlights:

- Guidelines-compliant Dialog Presentation including button icons and save-timing info
- SDI Windowing Interface and Gnome-MDI (using Tabs)

- Compliant Window Titling including traditional dirty mark (*)
- Large Gnome Toolbars and Icons
- Recommended Gnome Menus and Icons

*It is highly recommended that Java 6 be used on Linux, as there is a significant improvement in the Look and Feel fidelity of the GTKLookAndFeel. In earlier versions the general look is extremely poor and dramatically degrades the believability of a Gnome application. If this not possible, or another look and feel is desired, it may be desired to remove some of the automatic Gnome-ish attributes of the application UI. Icons in the menubars and buttons of dialogs can be removed, for example, using the manager.

While we do not directly support Solaris yet, some newer Solaris configurations use Gnome as the Desktop Manager. To cause JDAF to use the LinuxGnomeApplicationUIManager on Solaris, set the system property “usegnome=true”. JDAF uses the “GNOME_DESKTOP_SESSION_ID” environment variable to detect Gnome.

### GNU/Linux Highlights (KDE)

Guidelines for GNU/Linux KDE Desktop applications are described in KDE User Interface Guidelines. JDAF uses the LinuxKDEApplicationUIManager class to support features from that specification and observances from other popular KDE software, when this application UI is used. Following are some highlights:

- Guidelines-compliant Dialog Presentation including button icons and orientation
- SDI Windowing interface with popular round-robin window positioning
- Traditional window titling semantics with embedded dirty flags
- Common menu compliance with KDE icons and Special Go menu

*It is highly recommended that Java 6 be used on Linux, as there is a significant improvement in the Look and Feel fidelity of the GTKLookAndFeel. In earlier versions the general look is extremely poor and dramatically degrades the believability of a KDE application. Furthermore, we recommend that the latest version of a KDE distribution be used as earlier ones may not work as well with GTKLookAndFeel, which was designed for Gnome. If this not possible, or another look and feel is desired, it may be desired to remove some of the automatic KDE-ian attributes of the application UI. Icons in the menubars and buttons of dialogs can be removed, for example, using the appropriate manager.

A System property option is available that will force JDAF to use the LinuxKDEApplicationUIManager. Use the System property “usekde=true”. Otherwise, JDAF uses the “KDE_FULL_SESSION=true” environment variable to detect KDE.

## MAC OS X Highlights

Guidelines for a Mac OS X application are described in detail by the web publication Apple Human Interface Guidelines by Apple Computer. JDAF uses the MacOSXApplicationUIManager class to support features from that specification when running on Mac OS X. Following are some highlights:

- Automatic integration of recommended Java integration System properties (You do not have to do anything special)
- Guidelines-compliant Dialog Presentation *
- Guidelines-compliant File Alerts and Negotiations
- SDI Windowing interface with window dirty dot and other platform nuances
- API Access to "Brushed Metal" Window Background Effect and other properties **
- Guidelines-compliant Menus, including special Window menu behavior
- Single Application Menubar
- Unified Toolbar Support (on Leopard/Java5) ***
- Native OS X application menu bar integration of Actions

* We are working on animated "dialog sheets" next.

** Apple suggests that the brushed-metal window background effect only be used on applications that show a single window (think iTunes®.) You can set this behavior using the DesktopApplication method *setAllowsMultipleOpens(false)*. And set the background effect using the OSXProperties method *setBackgroundEffect()*. It accepts the following constants:

- BACKGROUND_EFFECT_NORMAL
- BACKGROUND_EFFECT_BRUSHED_METAL
- BACKGROUND_EFFECT_BRUSHED_METAL_ROUNDED

*** Normally toolbars are turned off for OS X because they are non-standard. If you turn them on, Leopard OS guidelines suggest using the "unified toolbar" guidelines which has the toolbar merged with the window title bar and has some button styles and layout specifics. JDAF will take care of all the details with the caveat that you must set *OSXProperties.setBackgroundEffect (UNIFIED_TOOLBAR)* immediately in your main method. You may then optionally use the methods in the *MacOSXApplicationToolBarsUI* to apply toolbar button styles while building your toolbars.

Note: there are some Apple guideline recommendations that currently are impossible to provide. Both bugs in Apple's Java implementation and lack of API on the Java side prevent them. We are constantly looking for ways to get around these issues. Still, we believe our first offering is high-fidelity and we will continue to improve it with each update.

### Windows XP Highlights

Guidelines for a Windows XP application are described in detail by the web publication The Official Guidelines for User Interface Developers and Designers by the Microsoft Corporation. JDAF uses the WindowsXPApplicationUIManager class to support features from that specification as well as conventions from popular native software, when running on Windows XP. Following are some highlights:

- SDI or classic MDI windowing interface. MDI with full-frame expansion features*
- Classic or Modern Icon themes depending on Classic mode detection
- Guidelines-compliant file negotiations, including illegal file name protection
- Guidelines-compliant Menu and Toolbar Items
- Inline recent documents with fuzzy path formatting
- Popular window arrangement features **
- Conventional Windows Hotkey integration (invisible accelerators)
- Automatic use of the WinLAF Project, if jar present. ***

* The WindowsXPApplicationUI uses an SDI windowing interface by default. To get a classic Windows MDI interface, use the application style *MDI_ APPLICATON_STYLE* when creating a GUIApplication. If TDI is desired instead you can bitwise OR the style FORCE_WINDOWS_XP_ TDI_APPLICATION_STYLE with the MDI style. Alternatively, you can use TDI_APPLICATION_STYLE, but this wholesale sets TDI on all platforms, which may not be desirable.

** The both SDI and MDI WindowsXPWindowGuidelines facilitate the special “Arrange” window commands popular in many MS Office applications. This functionality will arrange all open windows evenly in a grid pattern. This function can be configured using the *WindowArranger* object via the *getWindowArranger()* method.

*** A popular pre-Java 6 project for tweaking the Swing Windows Look and Feel is the WinLAF Project. This jar (winlaf-0.5.1.jar) modifies the Windows Look and Feel to make it more functional and accurate on pre-Java 6 systems. We highly recommend downloading and adding this jar to the classpath if your application will be using the traditional Windows Look and Feel. The WindowsXPApplicationUIManager will automatically load this jar if appropriate.

## GUIApplicationLifecycleListener

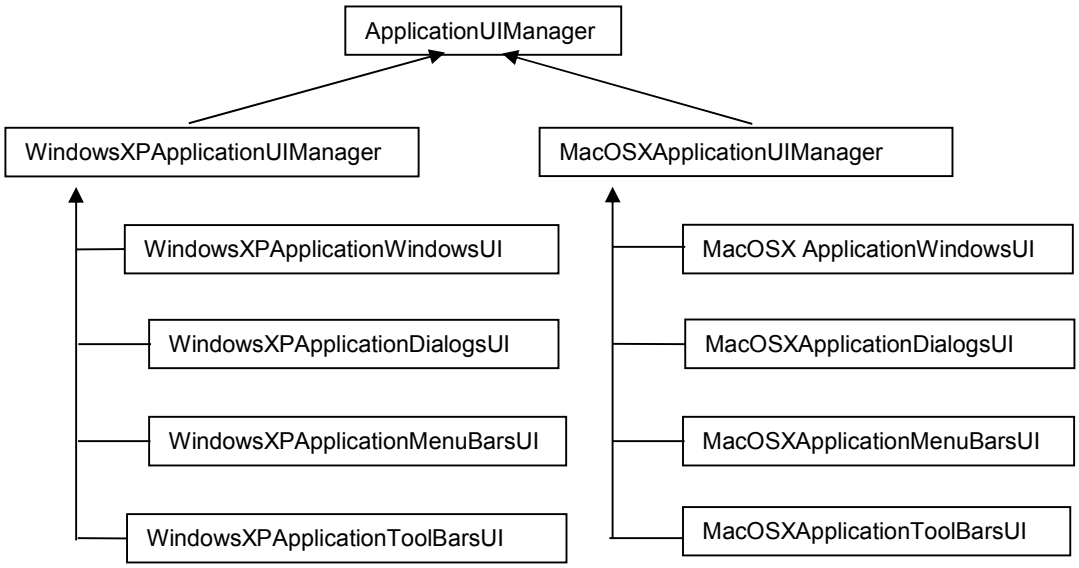
A special ApplicationLifecycleListener subclass is available for GUIApplications. The *GUIApplicationLifecycleListener* contains an additional event method called g*uiOpening(),* which can be used to detect when the UI is about to be displayed for the first time. This differs from *applicationOpening()* which is called prior to any engagement of Swing resources.

## Accessing the Application UI

While the application UI is managed automatically, there are cases when accessing the UI directly is desired. The ApplicationUIManager is composed of a set of ApplicationUI classes, each dedicated to implementing guidelines-compliancy in areas of the UI.

The root application UI classes are *ApplicationWindowsUI*, *ApplicationDialogsUI, ApplicationMenuBarsUI* and *ApplicationToolBarsUI*. To work with the UI programmatically, access the ApplicationUIManager from the GUIApplication using *getApplicationUIManager()*, then use the respective get methods: *getWindowsUI()*, *getDialogsUI()*, *getMenuBarsUI()*, and *getToolBarsUI().* If TDI is being used, *getTabbedUI()* will return a *TDIApplicationUI* object.

Using the Abstract Factory Pattern[5], each OS-specific ApplicationUIManager manages dedicated subclasses that implement OS-specific UI behavior. For example:




### Extending an Application UI

Generally, the behavior of the application UI is suitable in most situations; much like the Swing Look and Feel is generally suitable. However, the behavior of a managed UI can be overridden for a given OS if desired.

When a GUIApplication is constructed, it relies on an *ApplicationUIManagerFactory* to create the appropriate ApplicationUIManager, based on the underlying OS. To control which ApplicationUIManager

[5] Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley Professional Computing Series). http://mahemoff.com/paper/software/learningGoFPatterns/

is vended, you would need to subclass the ApplicationUIManagerFactory and return it from a DesktopApplication subclass via *defaultApplicationUIManagerFactory()*.

The factory has various *createX()* methods for each supported OS/platform. These methods can be overridden to provide extended implementations of a given ApplicationUIManager object.

### *JDAF-managed Swing Replacements*

Since JDAF manages the basic UI framework, you loose a degree of control over the types of implementation classes. (This is not a loss per se, as JDAF is making intelligent cross-platform decisions that otherwise would not be practical to make in normal code). If you require your own implementation of a basic Swing class managed by JDAF automatically, such as JFrames or JIDE library components such as DockableFrames or DocumentPanes, we have provided a mechanism. You may supply your subclass of the desired Swing object as a System property according to the following table in which case JDAF will honor your requirement in the most compatible way. Please use these System properties:

| Property Name | Root Class | Comments |
|---|---|---|
| jdaf.window | javax.swing.JFrame | Data Windows for use in Non-Docking and Non-Action Framework Applications |
| jdaf.toolbar | javax.swing.JToolBar | Use only in Non-Action Framework Applications.<br>See jdaf.dockable.menubar |
| jdaf.menubar | javax.swing.JMenuBar | Use only in Non-Action Framework Applications. See jdaf.dockable.menubar |
| jdaf.mdi.window | javax.swing.JInternalFrame | Use in MDI on Windows OS only |
| jdaf.tdi.tabbedpane | com.jidesoft.swing.JideTabbedPane | Use only in TDI Applications |
| jdaf.framed.tabbedpane | com.jidesoft.swing.JideTabbedPane | Used for Tabs in SplitApplicationUI using FramedApplicationFeature |
| jdaf.dockable.toolbar | com.jidesoft.action.CommandBar | Use only in Action Framework Applications |

| Property Name | Root Class | Comments |
|---|---|---|
| jdaf.dockable.menubar | com.jidesoft.action.CommandMenuBar | Use only in Action Framework Applications |
| jdaf.tdi.documenttabbedpane | com.jidesoft.document.DocumentPane | Use only in TDI Applications when DocumentPane is specified |
| jdaf.tdi.documentcomponent | com.jidesoft.document.DocumentComponent | Must implement the (JComponent, String, String, Icon) constructor |
| jdaf.dockable | com.jidesoft.docking.DefaultDockableHolder | Use when using Docking Framework only |
| jdaf.dockable | com.jidesoft.action.DefaultDockableBarHolder | Use when using Action Framework only |
| jdaf.dockable | com.jidesoft.action.DefaultDockableBarDockableHolder | Use when using Docking Framework and Action Framework |
| jdaf.mdi.dockable | com.jidesoft.docking.DefaultInternalFrameDockableHolder | Use for Docking Framework for internal frames on MDI Windows OS only. Note that jdaf.dockable or jdaf.dockablebar is used for the application window. |
| jdaf.dockableframe | com.jidesoft.docking.DockableFrame | Subclass for DockableFrames when using Docking Framework |

Note that all subclasses should have an empty constructor, unless documented otherwise in the comments. Some resources such as menus, dialogs and buttons do not support this feature because runtime implementation choices keep the class from being a predictable root implementation.

## The GUIApplication Class

Making a GUI application generally starts by instantiating an instance of GUIApplication and configuring its properties. Since the UI is managed, the remaining concerns for the developer are:

1) Determining the application style

2) Determining what DataModel(s) the data will be managed with

3) Adding DataViews for the DataModels

4) Adding application-specific Actions for Menus and Toolbars

Developing printing and integrating help are optional steps. If printing or help are not going to be supported in your application, we recommend calling the GUIApplication methods *setPrinter(null)* and *setHelp(null)*, respectively. The UI will configure to completely remove these options as appropriate to the OS.

You create an application instance like so:

```
GUIApplication application = new GUIApplication ("My Application");
// configure application…
application.run(args);
```

There are also static implementations of *run()* that provide an interesting development option focused on convenience. For example:

```
GUIApplication.run( "My App" , new MyApplicationFeature() {
    public void install() {
        // configure application
    }
});
```

You may of coarse subclass GUIApplication and provide the configuration in the constructor.

However you approach the use of GUIApplication, JDAF promotes a "set it and forget it" style of property configuration. We call these "pre-run" options because they are intended to define the general nature of the GUIApplication, and then be committed with a call to *run()*. Unless otherwise noted, all application configuration options in the framework are pre-run properties and normally should not be changed during the lifetime of the application.

GUIApplication constructors support varied sets of convenience for defining the basic attributes of the application. Keep in mind that the application style can *only* be set via a constructor because it sets up fundamental UI constructs internally. Beyond that, a JDAF application requires a minimum of one DataModelFactory and one DataViewFactory in order to run. The possibilities emerging from the

combination of DataModelFactory and DataViewFactory objects provides for many varied application designs.

You may of coarse treat GUIApplication as a bean and use the *setX()* and *addX()* methods to configure the object.

## Setting the Application Style

An overarching application style is defined via a constant in the constructor of the GUIApplication. This style determines the general nature of the application as it regards the primary DataModel presentation in the UI. The following constants are available.

| Constant | Comments |
|---|---|
| DEFAULT_APPLICATION_STYLE | Without passing a constant, this is the default style used. The effect is an SDI windowing interface on all OS'.<br>Note that by SDI we are speaking of the style of window management, not the number of windows allowed. The *setAllowsMultipleOpens()* method controls whether one or many DataModel can be open simultaneously. In the SDI style, each DataModel is opened in its own window. |
| MDI_APPLICATION_STYLE | Use this constant to set a "Multiple Document Interface" (MDI) style for the application.<br>MDI is only recognized on some OS'. Hence on Windows OS the UI will have a "main window" and manage data windows in that frame. On Gnome-Linux a tabbed interface (TDI) will be used. On OS X, the option is ignored and a normal SDI interface is used. |
| FORCE_WINDOWS_XP_TDI_ APPLICATION_STYLE | The MDI classic interface is default when specifying the MDI_APPLICATION_STYLE on Windows XP.<br>Some consider the Windows classic MDI interface a dated interface paradigm. To others it is a familiar part of life on Windows. For those who would like the tabbed look instead, |

| Constant | Comments |
|---|---|
| | you can bitwise-or this option with the MDI style to force Windows to use a TDI interface. |
| TDI_APPLICATION_STYLE | Use this style to set a "Tabbed Document Interface" (TDI) on all OS'.<br>This is a useful style particularly when using the JIDE Docking Framework for a "tools" style application. Otherwise, care should be used when utilizing this style for general application development. While it is acceptable on most OS', on some, like OS X, it is a strange paradigm for general use. |
| SPLIT_APPLICATION_STYLE | Combine this style to surround the primary DataView workspace with SplitPanes on all four sides.<br>This style is useful when UI content needs to be static in the window, outside of the DataView content.<br>This is a common style for email clients, catalog type application that have navigation, or any situation that needs to share the dynamic content with some kind of static content. |

### Using Split Application Optional Style

Bitwise-or the SPLIT_APPLICATION_STYLE with any other style mentioned above to provide split panes on the NORTH, SOUTH, EAST, and WEST of a primary DataView windows. To manage this capability, use the *SplitApplicationUI* object. It is accessible using:

```
SplitApplicationUI splitUI = application.getApplicationUIManager().getSplitUI();
```

Use *setSplitMode()* to set the split configuration. Currently this supports:

- VERTICAL_SPLIT
- HORIZONTAL_SPLIT

Use *setSplitVisible()* to hide and show a split component. Use *setDividerLocation()* to set the divider location. SplitApplicationUI uses sides to identify the location of the split component. The sides are NORTH, SOUTH, EAST and WEST.

A WindowCustomizer can be used to manually add Swing Components to the splits, or a DataViewHandler can be implemented, if the component is a DataView. To add components use *setUserComponent(window, userComponent, side)*.

```
getApplicationUIManager().getWindowsUI().
        addWindowCustomizer(new WindowCustomizer() {
                public void customizeWindow(ApplicationWindowsUI windowsUI,
                                Container window) {
                        windowsUI.getSplitUI().setUserComponent(window,
                        myComponent, SplitApplicationUI.WEST);
                }
                public void disposingWindow(ApplicationWindowsUI windowsUI,
                                Container window) {
                }
            });
```

Here is a DataViewHandler implementation of installing to the split UI:

```
public class SplitPaneHandler implements DataViewHandler {
            public void openDataView(Container window, DataView dataView,
                                        DataModel dataModel) {
                getApplication().getApplicationUIManager().getSplitUI()
           .setUserComponent(window,
                (Component)dataView.getViewComponent(), SplitApplicationUI.WEST);
            }

            public void closeDataView(Container window, DataView dataView) {
                getApplication().getApplicationUIManager().getSplitUI()
                 .removeUserComponent(window.SplitApplicationUI.WEST);
            }

            public boolean isUseNewWindow(ApplicationWindowsUI windowsUI) {
                return false; // whatever is appropriate
            }
      }
```

### FramedApplicationFeature

As an alternative to direct SplitApplicationUI interaction, JDAF provides a modular and clean MVC side-framing facility via the *FramedApplicationFeature*. This feature requires the use of secondary DataModels and allows any kind of DataView to be side-framed as well as supporting multiple DataViews on a given

side and a ToggleFrameAction to toggle visibility. The idea is to let you simply work with DataModels and DataViews and let the feature manage the installation of framed DataViews.

For a DataModel's view to be routed to a side, one of the *addFrameMapping()* methods must be called. There are various signatures of this method, but they all result in creating a *FrameConfiguration*. The FrameConfiguration defines how the DataModel is introduced. The following example creates two framed components called “Frame1” and “Frame2”.

```
public class MyApplication extends GUIApplication {
  public MyApplication() {

   // feature
   FramedApplicationFeature feature = new FramedApplicationFeature();
   feature.addFrameMapping(“Frame1”,
       SplitApplicatonUI.WEST,
        SecondaryBasicDataModel.class,
       DataView1.class);
   feature.addFrameMapping(“Frame2”,
       SplitApplicatonUI.WEST,
       SecondaryBasicDataModel.class,
       DataView2.class);
   addApplicationFeature(feature);
  ...
 }

 //…DataView implementations
}
```

When a window opens, the default behavior for the feature is to originate all DataModels matching a FrameConfiguration and install their DataViews. You may change whether *openData()* or *newData()* is used by calling *setOriginationMode()* to *OPEN_DATA_ORIGINATION_MODE* (default) or *NEW_DATA_ORIGINATION_MODE*.

If you desire to not initially originate DataModels when the window opens, you may call the FrameConfiguration *setAutoInstall(false)*. Then call *toggleFrame(frameName)* with the frameName, or use *openData()* or *newData()* manually. We provide the *ToggleFrameAction* to do such a thing for you from menus and toolbars, and it maintains a checked state as well.

Should you need to define criteria for your mapped DataModel, call *setCriteria()* on the FrameConfiguration. This criteria will be used to originate the DataModel. Be sure to pass this same criteria in if you call openData(criteria) or newData(criteria) manually on the GUIApplication or via OpenAction or NewAction calls, as this is the connection to the FrameConfiguration.

In this final example, we use a DockableConfiguration directly and set it up so that the feature will not originate the DataModel on startup. The user execution of the ToggleFrameAction with cause it to show and hide.

```
public class MyApplication extends GUIApplication {
    ...
    FramedApplicationFeature feature = new FramedApplicationFeature();
    newFrameConfiguration config = newFrameConfiguration();
    config.setFrameName("Welcome");
     config.setAutoInstall(false);
     config.setInitSide(SplitApplicationUI.WEST);
     config.setCriteria("Hello World!"); // criteria ref
     config.setDataModelClass(SecondaryBasicDataModel.class);
     config.setDataViewClass(MyDataView.class);

          feature.addDockableMapping(config);
       ...
          getActionMap().put("frame1", new ToggleFrameAction("Welcome"));

     // install in menu and toolbar
  }
```

The DataModel and DataView mappings can be managed outside the feature as well, solely relying on the criteria match to route the DataView to a side. Internally, a *HashedDataModelFactory* and *SemanticNameDataViewFactory* are used by the feature. Just make sure the same criteria set in the factory is set in the FrameConfiguration.

Another benefit of FramedApplicationFeature is that is uses JTabbedPanes to host multiple DataViews on a given side. The JTabbedPane may be customized using a *FramedApplicatonFeature.TabbedPaneCustomizer*:

```
feature.addTabbedPaneCustomizer(new FramedApplicatonFeature.TabbedPaneCustomizer() {
    public void customizeTabbedPane(Container tabbedPane, int side) {
    }
});
```

The FramedApplicationFeature does not support tab-dragging or docking. This functionalitty requires use of the JIDE Docking Framework. (You can use the *DockingApplicationFeature* to integrate the framework into JDAF. The usage is similar to the FrameApplicatonFeature so your migration is be minimal.)

## Setting Fundamental GUIApplication Properties

There are some fundamental properties you can set to control the general behavior of your application.

- *Name* – there is no set method for this as it is either set in the constructor or from the resources file under the key {ClassName}.name
- *setVersion()* – Use to represent the application version. This is a simple string of the version. This can be loaded from resources as well under the key {ClassName}.version
- *setVendor()* – Use to reflect the vendor of the application. This value is used in the data directory path on some platforms and otherwise in the default about dialog. This can be loaded from resources as well under the key {ClassName}.vendor
- *ult about dialog implem* – True by default. Use this option to define whether multiple DataModels are allowed in your application. This option is true by default, allowing multiple DataModels/DataViews. Set to false to allow only one DataModel at a time. If false, when a new DataModel is introduced and one is pre-existing, the old one is disposed before the new one is introduced. To force the DataViews to open in the same window call setReuseWindows(true) in the ApplicationUIManager. Otherwise a new window is produced each time.
- *setNewDataOnRun()* – True by default, accept on Windows XP MDI. Use this option to force a new DataModel to be opened initially. This can be set to false.

  While this property is managed automatically depending on the OS, this setting to true ensures that this behavior happens for all OS platforms.

  NOTE: Keep in mind that this may keep a UI from opening. If the interface is in TDI style and/or is using the JIDE Docking Framework JDAF will open the window. You may register a *WindowCustomizer* to detect the window since no DataViewEvent will be generated.
- *setUseThreadedOrigination()* – If set to true, new DataModels are introduced on a separate thread. This can be desirable if a UI needs to maintain its responsiveness while an expensive DataModel instantiation is taking place.
- *setExitApplicationOnLastDataView()* –OS and application-style-dependant; normally true. Use this setting on SDI application styles (SDIApplicationWindowsUI only) to determine whether the JVM will terminate when the last window is closed. For example, if TDI style is used, and this option is false, the last tab can be closed in the last window and the window/JVM remains open. Otherwise, shutting the last tab shuts the window, which shutdown the JVM. Normally, the default setting of this option is appropriate, but your requirements may be different.
- *setAutoSaveSecondaryDataModels()* – True by default; this option ensures that secondary DataModels are saved if they are dirty, before a window closes. Otherwise, since they are out of

the primary Data cycle (menuing system, etc) you need to manually call *saveData(DataModel)* on them. For example inside a DataViewListener *dataSaved()* event of the primary DataView.

- *setAllowMultipleDataViewsPerModel()* – False by default. Use this option to allow multiple DataViews to be vended for the same DataModel.

JDAF selects the most appropriate application UI depending on the underlying OS. The benefit of this feature is the intuitive, native feel it provides your application. Each OS has a unique ApplicationUIManager subclass, allowing the OS to be treated as a first class citizen.

The downside is that this can make OS-dependant coding decisions less cut and dry. JDAF supports a helper class called *OSApplicationCustomizer* that provides some resilience when making OS-based coding decisions, as well as some practical benefit. OSApplicationCustomizer is designed to be used as an inner class as an alternative to "if" blocks. It provides a *customizeApplication()* method for each ApplicationUIManager implementation. By implementing the respective methods you allow JDAF to select the most appropriate method at runtime.

```
new OSApplicationCustomizer() {
 public void customizeApplication(CrossPlatformApplicationUIManager ui) {
        // set Java Cross-Platform-Specific settings
 }
 public void customizeApplication(LinuxGnomeApplicationUIManager ui) {
        // set Linux-Specific settings
 }
 public void customizeApplication(MacOSXApplicationUIManager ui) {
        // set Mac OS X-Specific settings
 }
 public void customizeApplication(WindowXPApplicationUIManager ui){
        // set Window XP-Specific settings
 }
}.customize(application);
```

Notice how the OS-specific ApplicationUIManager is passed in as a typed object. As more OS platforms are supported, the available *customizeApplication()* methods will be expanded.

## Setting the Swing Look and Feel

This is a good time to discuss the relationship between Swing Look and Feel and JDAF Managed UI. One of the goals of JDAF is to facilitate the best possible cross-platform OS integration. This is accomplished using the combined effects of the *system default* Swing Look and Feels and our Managed Application UI. In other words, these two technologies work together to facilitate a high-degree of

believability on a given platform. Be aware that changing the Look and Feel will not change the behavior of the Application UI. So normally we recommend that on platforms where the System look and feel is appropriate, go with it, as this will yield the greatest believability on a given OS.

However, there are always exceptions. For example, in the event that JDAF does not support a given OS/platform, the Java Cross-Platform Application UI is used, which conforms to the Sun Java Look and Feel Guidelines and hence uses the Metal Look and Feel. Changing the Look And Feel here may be desirable.

In the event that your project requires a consistent Look And Feel on all platforms, you can set the "usecp" System property, and all platforms will use this Cross Platform Application UI.

Another clear example is Windows XP, where one encounters varying application UIs, even within Microsoft's own product lines. (In the Vista Guidelines, Microsoft is encouraging more consistency). If you are familiar with JIDE products then you are aware of our many Look and Feels available for Windows OS. These Look and Feels give your application that polished look on Windows.

With this knowledge, if it is necessary to change the Look and Feel in JDAF you will need to set the following option so that when the Managed UI is started it doesn't override your settings:

```
application.getApplicationUIManager().setSetsLookAndFeel(false);
```

Note: This setting is turned off automatically if you define the "swing.defaultlaf" System property. We do not recommend doing this unless you are branding your application to look the same on all platforms, in which case you should also pass the option "usecp". This will force the Cross Platform Application UI to be used on all platforms.

Now we can set a special Look and Feel on Windows, for example:

```
if(SystemInfo.isWindowsXP()) {
    LookAndFeelFactory.setDefaultStyle(
                                LookAndFeelFactory.OFFICE2003_STYLE);
    LookAndFeelFactory.installDefaultLookAndFeelAndExtension();
}
```

Make sure you do this before *run()* is called.

Additionally, part of the managed UI behavior is the population of standard icons for a given OS. So just changing the look and feel may not complete the transformation you desire. In other cases, the standard Look and Feel many be acceptable, but you wish to have a different set of icons. JDAF provides a class for bulk swapping out the icons in an application; the *IconTheme*. This example swaps out the icons on Windows XP to use the classic Windows icons:

```
IconTheme theme = new StandardIconTheme(
                    StandardIconTheme.WINDOWS_MODERN_VARIATION);
theme.install(application);
```

This should be called in a pre-run scenario. To manage your own IconTheme we provide *MapIconTheme.* But you certainly may implement your own or your own. IconTheme is discussed later in this document.

## Application Data Cycle

A GUIApplicationAction is installed for each controller method in the GUIApplication ActionMap. These actions are respectively installed into the standard menus and toolbars when the UI starts. The following table explains the action/method association.

| Action | DesktopApplication Method |
|---|---|
| NewAction | *newData(Object)* |
| OpenAction | *openData(Object)* |
| SaveAction | *saveData(DataModel)* |
| ResetAction/ RevertAction | *resetData(DataModel)* |
| PrintAction | *printData(DataModel)* |
| CloseAction | *closeData(DataModel)* |

These methods may be called programmatically as well.

### Creating Different DataModels and DataViews

The *NewAction* and *OpenAction* classes are loaded by default in the File menu of the GUIApplication. They call the GUIApplication controller methods *newData(criteria)* and *openData(criteria)*, respectively. These methods cooperate with the *DataModelFactory* objects to originate DataModels. The Actions have a *criteria* property. The value of this property what is passed to the controller methods and hence to the DataModelFactory implementations when the Action is invoked. (The resulting DataModel stores this criterion in the *criteria* property. In the case of *BasicDataModel* it is forwarded to the data property as well.)

Both NewAction and OpenAction allow for a *queuedDataRequestKey* to be set via *setDialogRequestKey()*. If you cache a DialogRequest in the ApplicationDialogsUI with this key, it will be invoked and the value from the *DialogResponse* will be used as the criteria for Action, and hence the DataModel origination. For example; this code sets up an application where the criteria can be selected from a combo box.

```
public class MyApplication extends GUIApplication {
   public MyApplication() {
```

```
        // queue a dialog request
        String[] criteria = new String[]{"Model1", "Model2","Model3"};
        StandardDialogRequest request = new StandardDialogRequest(
                    new CriteriaPane(criteria), DialogRequest.OK_CANCEL_DIALOG);
        StandardDialogRequest.addQueuedDialog(application, "criteria", request);

        // reference in action
        ((NewAction)application.getActionMap().get(ActionKeys.NEW)).
                                            setDialogRequestKey("criteria");

        // use a hashed data factory to map the values to DataModels
        HashedDataModelFactory dataFactory = new HashedDataModelFactory();
        dataFactory.addDataModelMapping(criteria[0], Model1.class);
        dataFactory.addDataModelMapping(criteria[1], Model2.class);
        dataFactory.addDataModelMapping(criteria[2], Model3.class);
        addDataModelFactory(dataFactory);

        // use a mapped view factory to map the DataViews
        MappedDataViewFactory viewFactory = new MappedDataViewFactory();
        factory.addDataViewMapping(Model1.class, View1.class);
        factory.addDataViewMapping(Model2.class, View2.class);
        factory.addDataViewMapping(Model3.class, View3.class);
        addDataViewFactory(viewFactory);

    // run application
    run();
    }

    public class CriteriaPane extends DialogPane {
        JComboBox criteriaBox = new JComboBox();
        public CriteriaPane(String[] criteria) {
            criteriaBox.setModel(new DefaultComboBoxModel(criteria));
        }
        protected void initializeComponents(DialogRequest request) {
            add(criteriaBox);
        }
        protected void commitComponents(DialogResponse response) {
            response.setValue(criteriaBox.getSelectedItem());
        }
    }
```

```
    // DataModel implementations …
    // DataView implementations…
}
```

The remaining data cycle controller methods operate on the "focused" DataModel. *SaveAction* calls *saveData(focusedDataModel)*, *ResetAction* calls *resetModel (focusedDataModel)*, etc. In the case of the GUIApplication, the focused DataModel is defined as the primary DataModel whose associated DataView is in the front-most window.

## Managing Models

The simplest approach to managing data models is the use of the BasicDataModel to hold your data object in its *data* property. By default, BasicDataModel sets the criteria used to create the model object into the data property. Otherwise, it is somewhat like an adapter in that it provides empty implementations of the data cycle methods from *AbstractDataModel*. You can use BasicDataModel as is, or subclass it and implement one or more data methods. An alternate strategy would be to add a DataModelListener to receive notifications of all DataModel lifecycles, and modify the model data externally.

If file-based persistence of the data is desired, a FileDataModel should be used, via the FileDataModelFactory. We recommend using the FileBasedApplication or the FileHandlingFeature if file functionality is desired, since there are many other issues when working with Files in an application besides the actual marshaling of data. (See "Making a Document-Centric Application" for more details.)

Subclassing AbstractDataModel is also appropriate in which case you may add properties and bean methods to fully provide a model for your data.

To integrate your model into the application, you can set your class in the BasicDataModelFactory and it will be instantiated. Just make sure there is a zero-argument constructor in your DataModel implementation. If there is more than one type of data, use the HashedDataModelFactory or the MappedDataModelFactory or multiple BasicDataModelFactorys. Otherwise, a custom DataModelFactory implementation can be used so that application-specific logic can be employed to vend the appropriate DataModel implementation.

Whatever DataModelFactory/DataModel implementation is chosen, be sure to balance it with an appropriate DataViewFactory that will respond to the DataModel to vend a proper DataView.

## Managing Views

Each DataModel produced will need at least one corresponding DataView implementation so that the data can be represented in the UI. A DataViewFactory vends a DataView based on the DataModel.

If you desire more than one DataView per DataModel, you must set the *setAllowMultipleDataViewsPerModel()* property and then provide more than one DataViewFactory that returns a unique DataView instance for the same DataModel as other factories. In this scenario, the first DataView of a primary DataModel, designated by a primary property of true, will be installed automatically according to the application style. Any consecutive DataViews, or DataViews belonging to secondary DataModels, designated by a primary property of false, will require a *DataViewHandler,* so that you may install the DataView where you desire.

Provided that a DataViewHandler is installed, primary or secondary DataModels may also have DataViews added to them during the application lifecycle. Use the ApplicationUIManager methods *openDataView()* and *closeDataView()* to manage these auxiliary DataViews.

> *Keep in mind that DataViewFactories are still used to vend the DataView. So be sensitive to the factories you have installed, particularly if they anonymously vend DataViews without a sensibility concerning the primary state of the DataModel in the UI. For this reason the JDAF factories provide an active property, so that certain factories may be disabled, for example to skip the re-vending of a primary DataView.*

GUIApplication uses Swing, your DataView implementation will need to facilitate a Swing component. Therefore subclassing JPanel and implementing DataView is a logical strategy. While implementing DataView is an application-specific option, using DataViewPane is an advantageous starting-point. DataViewPane is a DataView implementation based on JPanel. Simply subclass and override the following methods:

- *public void initializeComponents();*
- *public void updateView(DataModel);*
- *public void updateModel(DataModel);*

Use *initializeComponents()* to construct your UI.

Use *updateView(DataModel)* to move the data from the DataModel into the UI.

Use *updateModel(DataModel)* to move the information from your UI to the DataModel.

The application framework will call these methods as appropriate. Keep in mind that you may call the *updateX()* methods at any time, as appropriate in your application. We recommend that you use the *EventManager* to provide this behavior. Also, when calling *updateView()* be sure to use a *EventContext* that fires the event on the event thread, such as the *GUIEventContext*.

It may be desired to decorate the sides of a DataView. JDAF will always host a DataView component as the CENTER component inside a JPanel *BorderLayout*. Therefore it is always safe to use *getParent()*. We recommend decorations be added either in a DataViewListener DATAVIEW_OPENING event or the *initializeComponents()* method of *DataViewPane*, because at these times the DataView is installed. If called prematurely, the component may not be yet installed and the container will be null.

ApplicationWindowsUI calls *pack()* automatically prior to showing the window, so you may wish to call *setPreferredSize()* on your DataView component if the size is not acceptable.

Alternately, you can set all DataViews to a given size by calling:

```
application.getApplicationUIManager().getWindowsUI().setPreferredWindowSize(someSize);
```

DataViewPane provides other useful methods. You can use *getApplication()* to access the GUIApplication. Use *makeDirty()* to signify that the DataModel should be stored. You can use *postUndoableEdit()* to add an UndoableEdit to the UndoManager of the corresponding DataModel. Call *installEditables()* to register components with the edit menu. Call *setDefaultFocusComponent()* to set the component that should receive the focus event when the DataView opens.

DataViewPane is also a *PrintSource*. If you wish to provide printing, override the *getPageable()* method. To advertise that printing is available conditionally, you may additionally implement *isPrintable()*.

Whatever DataView implementation is used, a DataViewFactory must be added that will instantiate a given DataView based on the DataModel. If there is only one DataView type in your application, you can use the BasicDataViewFactory and simply provide your Class. BasicDataViewFactory will also create a default DataViewPane, in which you can use a DataViewListener *dataViewOpening()* event to populate it. In any case, make sure there is a zero-argument constructor in your DataView class. Alternatively, a review of the GUIApplication constructors shows that your DataView class can simply be passed in as a constructor argument.

If there is more than one type of DataModel, you can use the MappedDataViewFactory which allows you to map multiple DataModel classes to DataView classes. Or you may provide a custom DataViewFactory implementation or provide multiple DataViewFactory implementations that respond appropriately to the possible DataModel types produced by the DataModelFactory implementations.

If the view is dependent on a file type, you should use the FileDataViewFactory. This will let you associate a FileFormat with the DataView class, similar to using a FileDataModelFactory. In fact both implementations should be used together and agree on FileFormats.

### DataViewListener

GUIApplication supports a DataViewListener for monitoring the presentation of DataViews. This is particularly important since the windowing of the application is managed automatically, and the details of how windows are managed may change from OS to OS. DataModelListener provides a consistent interface no matter how the windowing system is managed. The following events are fired:

- DATA_VIEW_OPENING
- DATA_VIEW_OPENED
- DATA_VIEW_ACTIVATED
- DATA_VIEW_DEACTIVATED
- DATA_VIEW_CLOSING
- DATA_VIEW_CLOSED
- DATA_VIEW_MINIMIZED
- DATA_VIEW_MAXIMIZED
- DATA_VIEW_NORMALIZED

These events provide a similar behavior to AWT/Swing window events. A *DataViewAdapter* is available as a convenience implementation.

### Secondary and Auxiliary DataViews

JDAF manages all primary DataViews and installs them in a prominent position accoding to the application UI style. However, if a DataView's DataModel is secondary (*isPrimary() == false*) or the DataView is not the first DataView produced in a multi-view DataViewFactory scenario, then it is up to the application developer to install the DataView. This is accomplished by installing a *DataViewHandler* into the ApplicationUIManager like so:

```
application.getApplicationUIManager().addDataViewHandler(
        DataModel.class, new MyDataViewHandler());
```

A DataViewHandler is installed by associating it either with your DataModel implementation class or your DataView implementation class. If the handler is to be used for all views of a given DataModel implementation, use that DataModel class. If the handler is to be used for a particular DataView, use that DataView class. DataViewHandler has three methods:

- openDataView(Container, DataView, DataModel)
- closeDataView(Container, DataView)
- isUseNewWindow(ApplicationWindowsUI)

The *openDataView()* and *closeDataView()* are called to install and uninstall the DataView. The *isUseNewWindow()* is used to determine whether the DataView needs to open a new window or join the front window. The *container* is the window to install the DataView in.

Consider the following example, which uses the SPLIT_APPLICATION_UI and a DataViewHandler that installs a navigation view into the left side of the primary DataView window.

```
public class NavApp extends GUIApplication {
 public NavApp () {
     super(SPLIT_APPLICATION_STYLE);
     addDataModelFactory(new BasicDataModelFactory(
                    SecondaryBasicDataModel.class));
     addDataViewFactory(new BasicDataViewFactory(TreePane.class));

  // set window size, since there is no primary DataView
     ApplicationWindowsUI windowsUI =
                    getApplicationUIManager().getWindowsUI();
  windowsUI.setPreferredWindowSize(windowsUI.
                    getPreferredMaximumWindowSize());
     // add DataViewHandler
     getApplicationUIManager().addDataViewHandler(
                                  SecondaryBasicDataModel.class,
                                  new NavViewHandler());

}

// implement DataViewHandler
class NavViewHandler implements DataViewHandler {
       public void openDataView(Container window, DataView dataView,
                           DataModel dataModel) {
         // get SplitApplicationUI and set component on left
              SplitApplicationUI splitUI = getApplicationUIManager().getSplitUI();
              splitUI.setUserComponent(window,
                                  (Component)dataView.getViewComponent(),
                                  SplitApplicationUI.WEST);
              splitUI.setDividerLocation(window, SplitApplicationUI.WEST, 100);
       }

       public void closeDataView(Container window, DataView dataView) {
              SplitApplicationUI splitUI = getApplicationUIManager().getSplitUI();
              splitUI.removeUserComponent(window, SplitApplicationUI.WEST);
       }
```

```
    public boolean isUseNewWindow(ApplicationWindowsUI windowsUI) {
            return true; // because this is initial DataView on startup
        }
    }

    public static class NavView extends DataViewPane {
        JTree tree = new JTree();
        protected void initializeComponents() {
            setPreferredSize(new Dimension(100, 100));
            add(new JScrollPane(tree));
        }
    }

    public static void main(String[] args) {
        new NavApp().run(args);
    }
}
```

In this situation *isUseNewWindow()* is true, because our only DataModelFactory will vend a SecondaryDataModel, which would otherwise be ignored by our UI because of its non-primary status. Generally this should be false to integrate the DataModel into the existing UI.

## Window Management

GUIApplication automatically manages all windowing details concerning the presentation of DataViews. This may seem odd at first, but consider web development. Historically, part of the webs proliferation can be attributed to the fact that the page developer was only concerned about content, not the web browser window/application. We hope you will find it as liberating to only have to develop views for your data, and not have to worry about windowing details. Furthermore, there is, interestingly enough, a sufficient amount of OS-specific behavior concerning windows that is managed for you via this mechanism.

### WindowCustomizer

It is generally unnecessary to work directly with windows. Use a *DataViewListener* to capture window behavior. This interface abstracts the multiple ways in which DataViews may be hosted in an application and is most guidelines compliant. However, there are always exceptions, such as making keystroke bindings or other behavioral changes to a normal window. For these cases we provide the *WindowCustomizer*.

> We do not recommend using a WindowCustomizer for setting a window size. Window size is handled by the framework as a result of DataView contents. If you need to set an explicit size for your windows, use the pre-run option *setPreferredWindowSize()* in the ApplicationWindowsUI object of the ApplicationUIManager. If for some reason you need to invalidate the window, use the ApplicationWindowsUI.sizeWindow() method for compliant behavior.

You may register a *WindowCustomizer* with the *ApplicationWindowUI* object of the ApplicationUIManager. WindowCustomizer has two methods; one for when the window is about to show and one for when it is about to be disposed.

- *customizeWindow(ApplicationWindowsIU, Container)*
- *disposingWindow(ApplicationWindowsIU, Container)*

JDAF exposes windows as a *java.awt.Container* due to the various possible implementations of the windowing architecture. ApplicationWindowsUI can be used to access and perform functions on the window object in the windowing system in a cross-platform compliant manner.

### ApplicationWindowsUI

The ApplicationUIManager uses an OS-specific ApplicationWindowsUI class to manage the display of your DataView component automatically, using an appropriate document-window style for the OS and Application Style. The subclass tree is first split by two abstract subclasses; *SDIApplicationWindowsUI* and *MDIApplicationWindowsIU*. These classes will contain methods specific to these windowing styles. In the event of a TDI style application, the SDIApplicationWindowsUI hosts a *TabbedApplicationUI* which facilitates hosting DataViews in tabs within each SDI window. In the event of a Split style application, the SDIApplicationWindowsUI hosts a *SplitApplicationUI* which facilitates installing components in a split pane on the any of the NORTH, SOUTH, EAST, and WEST sides of each SDI window. Please see the javadoc for more details.

Note that ApplicationWindowsUI treats all window classes as java.awt.Container classes, so casting may be involved if you desire to work with a single window instance directly. To access the window of a DataView outside the DataViewEvent, you would use the *getDataViewWindow(DataView)* method. You can use the ApplicationUIManager methods *isSDI()* or *isMDI()* to determine whether to cast to a JFrame or a JInternalFrame, respectively. However there are many window functions in the ApplicationWindowsUI which accept the Container as an argument anyway.

## Window Sizing

JDAF will manage sizing characteristics of the data windows automatically based on DataView content size. There are platform neuances, such as cascading and "always on screen" policies and multiple monitor policies that make this desireable. However, sometimes a DataViews content is not determinate, or you may simply wish to set the initial size of your windows yourself. In these cases use the ApplicationWindowsUI method *setPreferredWindowSize()*. If you are using the JIDE Docking Framework and you are allowing JDAF to manage the layout persistence, JDAF will only respect your window size setting if there is no layout data saved yet. From then on, the users sizing is observed. (The layout files are stored in the default data directory returned from DesktopApplication method getDataDirectory(), which is a platform dependant path.)

Alternately, you can instruct the ApplicationWindowsUI to initially maximize windows on startup using the *setMaximizeWindows(true)*. We recommend this be done selectively in an OSApplicationCustomizer. On some platforms a maximized window is acceptable, such as Windows XP, but on others, such as Mac OS X with a 32" Cinema Display, it may be disconcerting to cover the entire screen.

The ApplicationWindowsUI method *getPreferredMaximumSize()* method will return an appropriate screen size for windows. This can be used along with the *setPreferredWindowSize()* to set a large screen size.

## Window Titling Logic

The ApplicationWindowsUI class handles the way titles are presented in GUI to mimic the semantics used on different platforms. Normally this behavior is sufficient. Titling is modeled on the OS Guidelines or, in their silence on the subject, the observation of popular applications on the given platform. However, it may be necessary to modify the behavior of the display characteristics.

While it may seem that setting a window title is a trivial issue, it can get quite complex when you consider the compound affects of multiple windows, various application styles, various OS guidelines, and various DataModel types. For example, does the title display the same when in a windowed interface as it does when in a tabbed interface? Since a tabbed interface is contained *in* a window, how does the title appear there? How about in the Main window of an MDI interface? Or all these simultaneously! How does the title display when there is no data? Does the title include the application name? If so, does it come before or after the document name? Furthermore, on some platforms when the document is dirty, this is denoted in the title. As you can see this is no trivial problem.

To address window titling we use a *WindowTitleFortmatter* object. This can be accessed from the ApplicationWindowsUI object in the ApplicationUIManager. WindowTitleFormatter is an interface that supplies three methods:

- *getMainWindowTitle(GUIApplication, DataModel)*
- *getDataWindowTitle(GUIApplication, DataModel)*
- *getDataTabTitle(GUIApplication, DataModel)*

These methods are called from the UI when the title of a UI element needs to be updated. So at least we have the three primary situations covered (MDI, SDI, TDI).

To facilitate the myriad of titling challenges with multiple OS and for multiple data model types, we use the *DefaultWindowTitleFormatter.* This WindowTitleFormatter implementation uses *ObjectFormats* to manage titling semantics. *ObjectFormat* is similar to a *MessageFormat*, but interprets user defined tokens (See ObjectFormat). There is an *ObjectFormat* and title pattern for each of the three titling requirements in the WindowTitleFormatter interface. To customize titling behavior, while still respecting OS guidelines, one can simply modify one or more of the title patterns.

But if you introduce a new DataModel type, or have mixed types, you have the option of customizing the pattern tokens interpretation by DataModel class, because ObjectFormats are mapped based on the DataModel class. The class mapping is polymorphic.

By default the AbstractDataModel class is registered with ObjectFormats so that all subclasses pick it up the title automatically. To designate a specific ObjectFormat to a specific DataModel class, create and configure the ObjectFormat, and add it to the DefaultWindowTitleFormatter using one of the *addXFormat(Class, objectFormat)* methods. The ObjectFormat instance must support the respective title pattern, because the pattern is OS dependant and shared across all ObjectFormats.

Currently the tokens supported in the window titling pattern are:

- ${APPLICATION_NAME}
- ${DATA_MODEL_NAME}
- ${DATA_MODEL_INDEX} (KDE-only)

Token interpretation is facilitated by ObjectFormatToken objects. Each ObjectFormatToken is associated with a specific token string. These objects are supplied to an ObjectFormat in order to substitute text against the pattern. The *ApplicationNameToken* and *DataModelNameToken* classes are used to perform the default token interpretation of the above tokens. To create a custom token interpreter, subclass one of these or the ObjectFormatToken class itself. See the javadoc for more information.

Alternately you can simply set a new pattern into an existing ObjectFormats using *setDataWindowPattern(), setTabDataPattern(), and setMainWindowPattern().*

## Working with Actions

The DesktopApplication supports an ActionMap for providing a centralized place for application functionality.

*Actions should be added before the application runs.*

GUIApplication is pre-configured with default Actions that provide basic application functionality. These actions are integrated into the Managed UI at startup during *run()*. Hence, you may remove or replace any other these actions prior to a call to *run()*. These standard Actions can be retrieved using the constants in the *ActionKeys* interface. Actions can be executed programmatically from the application using *executeAction(actionKey),* but normally they are installed into menus and toolbars (see *MenuBarCustomizer* and *ToolBarCustomizer*).

ApplicationAction contains a reference to the DesktopApplication once installed into the applications ActionMap. JDAF provides an ApplicationAction implementation designed for GUI applications called *GUIApplicationAction*. GUIApplicationAction has some enhancements over ApplicationAction, such as access to the GUIApplication instance, threaded execution, and support for large icons with the LARGE_ICON property constant.

When subclassing GUIApplicationAction you should consider overriding *actionPerformedDetatched()* instead of *actionPerformed()*. This threaded version provides better perceived performance because it allows the UI to respond immediately. The code is still executed on the event thread, it is just re-queued to execute after the UI has been restored, and making menus and toolbar buttons snap back into place before executing the Action.

When implementing an Action, it may be better to extend off one of these GUIApplicationAction subclasses instead, if their behavior is useful:

- *FocusTrackingAction*: This GUIApplicationAction always keeps a reference to the currently focused Component. This can be used when the current focused Component is needed for the action behavior.
- *ActiveStateAction*: This GUIApplicationAction will only be enabled when there is an open and non-minimized DataView-window available. It is useful for actions that require a DataModel/DataView to operate on.
- *DirtyStateAction*: This Action is only enabled when the focused DataModel is dirty.

A few GUIApplicationActions are reusable:

- ComponentAction: This FocusTrackingAction subclass is used primarily for Edit menu items. It delegates to the Action in the focused Component's ActionMap. You provide the actionKey that is used to look-up the Action in the Component's ActionMap. That Action is executed instead. The enable states are also synchronized.

- *DelegatingAction*: This GUIApplicationAction allows you implement your *actionPerformed()* method elsewhere, such as in a GUIApplication subclass or a dedicated class, under a synonym. This can reduce the number of Action classes to define.

## Pre-Installed Actions

GUIApplication installs a collection of fully-functioning Actions in GUIApplication by default. These Actions are loaded into the GUIApplication's ActionMap, and integrated into the menus and toolbars of the user interface. All GUIApplicationActions have localized names, mnemonics, icons and accelerators so they appear and function in an OS-specific manner.

The following listing details what default GUIApplicationActions are automatically installed.

*OS:* C=Cross Platform, G=Gnome-Linux, K=KDE-Linux, M=Mac OS X, W=Windows XP

| Action | Purpose | OS |
|---|---|---|
| AboutAction | Opens the about dialog | C,G,K,M,W, |
| ActivateAllWindowsAction | Brings all windows to the front/cascades on option-toggle | M |
| ActivateWindowAction | Brings the specified window to the front | C,G,K,M,W |
| ArrangeWindowsAction | Performs OS-specific window arrangement | W |
| CascadeWindowsAction | Arranges windows so that their top-left corners are positioned next to one another | C,M*,W |
| CloseAction | Disposes the focused DataModel | C,G,K,M,W |
| CloseAllAction | Disposes all open DataModels | G |
| CopyAction | Executes the "copy" or "copy-to-clipboard" Action of the currently focused Component** | C,G,K,M,W |
| CutAction | Executes the "cut" or "cut-to-clipboard" Action of the currently focused Component** | C,G,K,M,W |
| DeleteAction | Executes the "delete" or "delete-next" Action of the currently focused Component** | C,G,K,M,W |
| ExitAction | Exits the DesktopApplication | C,G,K,M,W |
| HelpAction | Invokes the DesktopApplication HelpSource*** | C,G,K,M,W |
| OpenAction | Calls *DesktopApplication.openData()* with user defined criteria, on the focused | C,G,K,M,W |

| Action | Purpose | OS |
|---|---|---|
| | DataModel**** | |
| MinimizeWindowAction | Minimize the front window/minimize all windows on option-toggle | M |
| NewAction | Calls *DesktopApplication.newData()* with user defined criteria, on the focused DataModel**** | C,G,K,M,W |
| NextAction | Cycles to the next window or tab in the application | K |
| PageSetupAction | Invokes the page setup dialog for the Printer*** | C,G,K,M,W |
| PasteAction | Executes the “paste” or “paste-to-clipboard” Action of the currently focused Component** | C,G,K,M,W |
| PreferencesAction | Invokes a PreferencesDialogRequest, as defined in the ApplicationDialogUI*** | C,G,K,M,W |
| PreviousAction | Cycles to the previous window or tab in the application | K |
| PrintAction | Invokes printing of the focused DataModel/DataView**** | C,G,K,M,W |
| RedoAction | Invokes *redo()* on the UndoManager of the focused DataModel | C,G,K,M,W |
| ResetAction | Calls *DesktopApplication.resetData()* on the focused DataModel**** | C,G,K,M,W |
| SelectAllAction | Executes the “selectAll” or “select-all” Action of the currently focused Component** | C,G,K,M,W |
| SaveAction | Calls *DesktopApplication.saveData()* on the focused DataModel. | C,G,K,M,W |
| SaveAllAction | Calls *DesktopApplication.saveData()* on all the dirty DataModels in the application. | G |
| ToggleWindowSizeAction | Toggles window size between the initial size and last user defined size. (Mac OS X “Zoom” command) | M |
| UndoAction | Invokes *undo()* on the UndoManager of the | C,M,W |

| Action | Purpose | OS |
|---|---|---|
| | focused DataModel | |

* On OS X, cascade is a side-effect of guidelines-recommended option-toggling the "Window->Bring to Front" menu item (ActivateAllWindowsAction), which results in an "Arrange in Front" menu item, which calls the cascade functionality.

** See ComponentAction.

*** Some actions are conditionally installed based on the application configuration. Examples are printing and help Actions which will not be installed into the UI if Printer and HelpSource are not available when the application is run.

**** FileBasedApplication/FileHandlingFeature installs alternate NewFileAction and OpenFileAction actions, as well as additional actions and behavior, specifically designed for File-handling requirements.

Some GUIApplicationActions of note have configuration capability to further define application behavior.

- NewAction and OpenAction can have their criteria set via *setCriteria()* which defines what is passed to the application via *newData(criteria)* and *openData(criteria)*, and to the respective DataModelFactory. A dialog can be used to set the criteria of the Action by queuing the DialogRequest in the ApplicationDialogsUI, and then setting this dialog key with *setDialogRequestKey()*. The criteria is pulled from the resulting DialogResponse.
- HelpAction can have its topic property set via *setHelpTopic()* to control what is passed to the HelpService from a menu item. Note that this value is overridden if the primary DataView component has the client property *HelpSource.HELP_HINT_PROPERTY_NAME* set with a topic.

The managed UI ensures that the Actions are installed in the correct places in the File, Edit, Window, and Help menus, according the OS-guidelines. One caveat is that the File menu in GUIApplication is the same for all OS. It simply provides access to the primary data methods of the DesktopApplication. However, by installing the FileHandlingFeature, or using the FileBasedApplication, which installs the FileHandlingFeature, the File menu will be populated according to OS-guidelines, which tend to promote a file-centric application.

Finally, unless otherwise noted, some Actions may be available in the ActionMap that are not installed into the UI. These can be used at the discretion of the developer.

### User Actions

The default Actions are available immediately after GUIApplication construction. In general, developers should install application-specific actions before *run()* is called. When *run()* is called, the UI will ultimately startup, which includes building the menus and toolbars via the actions.

The default Actions can be replaced using their *ActionKey* constants. By using these keys, the replaced Action will assume the OS-specific properties of the Action. Other keys will cause your own resource bundles to be used if they are available, according to the ActionResourcesBinding rules.

The GUIApplicationActionMap automatically uses a ActionResourcesBinding to bind resources to added Actions via their property keys, as defined in the Action interface. The resource file should be in the same file location as the Action class. Properties are recognized in a resource file by the following pattern:

```
[actionKey].Action.[propertyName]
```

As an example; for an Action keyed under "recalcTotals", the properties file could have a value:

```
recalcTotals.Action.Name=Recalculate Totals
```

The "Name" portion of the entry is equivalent to the property Action.NAME, so it will be loaded into that property of the Action. By default all the properties defined in the Action interface will attempt to load. You need only supply the values desired for localization.

Note: in order for the Action.SMALL_ICON and Action.ACCELERATOR_KEY to be loaded, that ObjectConverters should be configured for the Resources object where the Actions resources will be loaded. See the "Working with Icons" section.

Finally, if Actions are OS-specific, you can add them using an OSApplicationCustomizer to ensure that they are in sync with the Application UI. Or if using resource files, you can use the OS variant an define Action properties in different files.

### Auto Installation of Actions into GUI

A powerful ApplicationFeature called the *AutoInstallActionsFeature* may be added to the GUIApplication to install Actions into the GUI automatically. This feature uses special properties in your Actions and leverages JDAF menu and toolbar infrastructures to facilitate the installation of Actions. This includes the automatic creation of menus and toolbars as well as integration in the standard ones, largely removing the need to use MenuBarCustomizers or ToolBarCustomizers to write manual menu or toolbar code.

To use, simply add an instance of AutoInstallActionsFeature to your GUIApplication.

```
....
application.addApplicationFeature(new AutoInstallActionsFeature());
```

Next, for each of your Actions set some special properties to tell the feature where you want your action to appear in the UI. The special property names are defined in the AutoInstallActionsFeature.

```
...
Action action = new GUIApplicationAction("Export") {
      public void actionPerformedDetached() {...}
};
action.putValue(AutoInstallActionsFeature.MENU_ID, MenuConstants.FILE_MENU);
action.putValue(AutoInstallActionsFeature.TOOLBAR_ID,
       app.getApplicationUIManager().getToolBarsUI().getStandardToolBarName());
app.getActionMap().put("export", action);
```

This "Export" Action will be installed into the "File" menu and into the "Standard" toolbar. Either of these properties are the minimum needed for the Action to be installed. If you where to define your own menu name or toolbar name values into the respective properties, the feature would create the respective menu or toolbar and add the Action to it.

You can specify the exact positioning of the Action relative to another using the BEFORE_ACTION and AFTER_ACTION properties. For example, we would add the line:

```
action.putValue(AutoInstallActionsFeature.AFTER_ACTION, ActionKeys.SAVE);
```

to cause this Action will be installed after to the "Save" command.

When there is more than one Action being installed, it is important to define the NATURAL_ORDER property and set an Integer that is relative to the other Actions being installed.

```
Action action = new GUIApplicationAction("Export") {
      public void actionPerformedDetached() {...}
};
action.putValue(AutoInstallActionsFeature.MENU_ID, MenuConstants.FILE_MENU);
action.putValue(AutoInstallActionsFeature.NATURAL_ORDER, new Integer(2)); // second
app.getActionMap().put("export", action);

action = new GUIApplicationAction("Import") {
      public void actionPerformedDetached() {...}
};
action.putValue(AutoInstallActionsFeature.MENU_ID, MenuConstants.FILE_MENU);
action.putValue(AutoInstallActionsFeature.NATURAL_ORDER, new Integer(1)); // first
app.getActionMap().put("import", action);
```

Here, the "Import" command will appear before the "Export" command due to the NATURAL_ORDER setting. Without this property we are at the mercy of the order returned by the underlying storage mechanism of the ActionMap.

When working with menus, the property MENU_GROUPING_ID allows an Action to be appended to a certain MenuGroup in the standard menus. This property will also create these groupings either in a standard menu or your own menu if they don't exist. This option can be used in leu of, or along with, the BEFORE_ACTION and AFTER_ACTION properties. For example, here we define the Action to be appended in the "Undo" group of the "Edit" menu.

```
Action action = new GUIApplicationAction("Undo History") {
      public void actionPerformedDetached() {...}
};
action.putValue(AutoInstallActionsFeature.MENU_ID, MenuConstants.EDIT_MENU);
action.putValue(AutoInstallActionsFeature.MENU_GROUPING_ID,
                                        MenuConstants.EDIT_UNDO_GROUP_ID);
app.getActionMap().put("undoHistory", action);
```

The section "MenuGroupings" describes more about JDAF menu infrastructure. The ApplicationMenuBarsUI can be used to programmatically discover the resulting menu structures.

Because the Action installation is property-based and because properties can be bound from resource files, it is then possible to completely configure menu and toolbars using resource files. This is done by defining properties in the same way other Action properties are defined. For example; our cumulative "Import/Export/Undo History" examples could be defined as:

```
export.Action.Name=Export
export.Action.MenuID=fileMenu
export.Action.NaturalOrder=2
import.Action.Name=Import
import.Action.MenuID=fileMenu
import.Action.NaturalOrder=1
undoHistory.Action.MenuID=editMenu
undoHistory.Action.MenuGroupingID=editMenu.undo
```

Be sure to check to javadoc for the correct constant values. Note that if a SELECTED_KEY boolean property is added to the Action, a "toggle" version of a menu item or toolbar button will be used.

One other important caveat for loading these special properties is that the ActionResourceBinding needs to be aware of them. By default it is only aware of the default Action properties and our own LARGE_ICON. Since GUIApplcationAction is a *Resourceful* object, it can return its own *ResourceBinding* implementation. This is where you should return an ActionResourceBinding with all of the properties added. However, for your convenience, the AutoInstallActionFeature provides a GUIApplicationAction called AutoInstallAction, which implements this detail:

```
Action action = new AutoInstallActionsFeature.AutoInstallAction() {
      public void actionPerformedDetached() {...}
};
app.getActionMap().put("import", action);
Action action = new AutoInstallActionsFeature.AutoInstallAction() {
```

```
        public void actionPerformedDetached() {...}
};
app.getActionMap().put("export", action);
Action action = new AutoInstallActionsFeature.AutoInstallAction() {
        public void actionPerformedDetached() {...}
};
app.getActionMap().put("undoHistory", action);
```

*AutoInstallActionsFeature currently does not support sub-menu definition or Toggle menu items or buttons.*

## Working with Icons

GUIApplication provides a default Application icon for each platform. To replace this icon, call the *setSmallIcon()* and *setLargeIcon()* methods of the ApplicationUIManager. These icons are shown in window titles and dialogs, and the default about dialog implementation[6].

Icons are also set in the default Actions when they are added to the GUIApplicationActionMap. To override these icons, simply set the Action properties as you normally would, or try using an *IconTheme* (discussed in the next section).

Icons can be loaded from resource bundles using the Resources API. To facilitate this, an *IconConverter* should be registered in the ObjectConverterManager. IconConverter allows for a root Class or resource path to be defined, which tells it where the icon resources can be found. Then icons can be loaded from resources like so:

```
ImageIcon icon = (ImageIcon)Resources.getObject(MyAction.class, "pencil", ImageIcon.class);
```

Icon loading is particularly important when binding resources to Actions. The *IconConverter* should be registered with the ConverterContext of the Resources object where the resource bundle is located, and the root of the IconConverter should be set to a location where the icon files reside, which may not necessarily be in same package. Note that when the IconConverter root is set to null, the system class paths are searched. Actions also require a converter to create KeyStrokes from Strings. JDAF provides a KeyStrokeConverter for this task. A static helper method is available that makes installing these resource converters a single step. Use the static ActionResourcesBinding method *installActionConverter()* once for each package where there are Actions.

---

[6] On Mac OS X, the only way to get the application name and icon to show in the dock, is to provide parameters in the JVM command line via the –Xdock:name="App Name" and -Xdock:icon="app.icns". The Jar Bundler will do this for you. Also, some installers take care of this as well. During development however, theses JVM properties can be set for convenience. To convert a .jpeg or .png file to an Apple .icns you can use 'System>Developer>Applications->Utilities->Icon Composer' application that comes with the Apple developer tools.

## IconThemes

JDAF provides a helper class called IconTheme that facilitates the configuring of icons in a single place for a GUIApplication. To use an IconTheme simply instantiate it and call the *install(GUIApplication)* method.

IconTheme requires four methods:

- getLargeApplicationIcon()
- getSmallApplicationIcon()
- getLargeActionIcon(actionKey)
- getSmallActionIcon(actionkey)

You can use any method to return the icons; Resources, IconsFactory, or traditional methods of loading ImageIcons. Each *geXIcon()* method is passed an *iconStyle*. Currently only one style NORMAL_ICON_STYLE is ever supplied. In the future, it may become desirable to access different versions of the icon, perhaps based on color depth or effect.

By default all icons are replaced when *install()* is called. If an icon method returns null, the icon is cleared in the destination. This can be changed by setting the *passive* property, in which case icons will not be replaced/cleared when an icon method returns a null value. This feature makes it straightforward to replace just a few icons. Passive is off by default. (Application icon setting is always passive.)

IconTheme supports a variation flag in the event that a given theme wishes to support variations. Default setting is the value NO_VARIATION.

Icon theme is abstract but we provide the following implementations:

- *MapIconTheme* – This IconTheme class is based on Maps. To use, simply load your icons into the theme declaratively before installing.
- *StandardIconTheme* – This theme is only useful if there is a need to restore the original OS-specific icons. It does however provide a special variation for Windows OS; the WINDOWS_MODERN_VARIATION. Normally, the Windows classic icons are shown, this variable allows a more modern set of icons to be installed that may better match on of the special Jide Look and Feels.
- *IconSetIconTheme* – This IconTheme interfaces with JIDE stock Icon libraries by using an internal JIDE Commons IconSetManager. It will automatically match the running OS to the installed icons library and otherwise ignore unavailable conditions.

## Working with Dialogs

In a normal desktop application, dialogs tend to be a scattered and disorganized portion of the application code. (Do you know where your dialogs are?). In large applications, this unstructured and largely non-object-oriented aspect of messaging can create maintenance problems.

GUIApplication provides a facility for centralizing and formalizing dialogs in a way that respects OS-guidelines and provides consistent, maintainable, and reusable presentations of dialogs. This is facilitated by the ApplicationDialogsUI object, which is a member of the currently installed ApplicationUIManager.

Instead of working with the JDialog directly, or clients of JDialog, such as JOptionPane or JFileChooser, you should use one of the DialogRequest classes. To show a dialog, the DialogRequest is presented to the ApplicationDialogsUI using *showDialog(request)*. The ApplicationDialogsUI presents the request to the user in an OS-specific manner and returns with a *DialogResponse*. The *DialogResponse* object provides a response code and an optional value, which is usually the button clicked, but could be any value, such as a File from a file dialog.

DialogRequests can also be queued with a key, for future display of pre-configured dialogs. Use *addQueuedDialog(application, key, dialogRequest)* to add a DialogRequest. Then to execute the dialog later, use *showQueuedDialog (application, key)*.

JDAF currently facilitates the following types of DialogRequests:

- MessageDialogRequest - Show messages and confirmations
- FileDialogRequest - Presents file negotiations
- StandardDialogRequest - Hosts user components
- ProgressDialogRequest – Presents a progress dialog. (used with DialogBlock in the Activity framework)
- PreferencesDialogRequest – Use to present a preferences/options dialog
- CompatibleDialogRequest - Allows legacy JDialogs to be presented as is

### Using DialogRequests

There are multiple patterns for sending a DialogRequest. But basically the most common way to send a dialog request is to use the static usability methods as this creates the most concise code. We will use a MessageDialogRequest as an example to illustrate the various use cases:

- Use the high-level static usability methods in a given DialogRequest class:

```
MessageDialogRequest.showConfirmDialog(application,
                        "The login was invalid. Try again?",
                        MessageDialogRequest.YES_NO_DIALOG);
```

- Create a DialogRequest object and submit it using a static usability method:

```
MessageDialogRequest request = new MessageDialogRequest(
                        "The login was invalid.",
                        "Try again?",
                        MessageDialogRequest.YES_NO_DIALOG);

DialogResponse response = MessageDialogRequest.showDialog(application, request);
```

- Create a DialogRequest object and submit to the ApplicationDialogsUI object:

```
MessageDialogRequest request = new MessageDialogRequest(
                        "The login was invalid.",
                        "Try again?",
                         MessageDialogRequest.YES_NO_DIALOG);

ApplicationDialogsUI dialogs = application.getApplicationUIManager().getDialogsUI();

DialogResponse response = dialogs.showDialog(request);
```

- Create a DialogRequest and queue it. Then execute it later:

```
MessageDialogRequest request = new MessageDialogRequest(
                        "The login was invalid.",
                        "Try again?",
                        MessageDialogRequest.YES_NO_DIALOG);

MessageDialogRequest.addQueuedDialog(application,"Invalid Login", request);
```

```
// Later
```

```
DialogResponse response = DialogRequest.showQueuedDialog(application, "Invalid Login");
```

DialogRequests have an id field that can be used optionally to identify DialogRequests at runtime. This is a simple String value. Use *setID()* and *getID()*.

### DialogListeners

A *DialogListener* can be registered to listen to all dialog activity via *DialogEvents*. There are two levels of listening. A DialogListener can be set in the DialogRequest itself, in which case it only receives events

during the lifecycle of the request. Or, the DialogListener can be added to the ApplicationDialogsUI, in which case it will be notified of all dialog activity.

### DialogRequestHandlers

Internally, DialogRequests are handled by *DialogRequestHandlers*. These are registered and mapped polymorphically by DialogRequest class. On occasion, it may be useful to access a DialogRequestHandler and set some options. Also, if you create a special kind of DialogRequest that requires special capabilities, creating and registering a DialogRequestHandler can be useful.

### Why Have Managed Dialogs?

At this point you may be wondering why dialogs should be managed like this. What's wrong with just using JDialog or JOptionPane? To be clear, using the ApplicationDialogsUI for dialog activity is elective. But we believe the benefits of using a managed system are manifold, so please allow us to elaborate on the gains of allowing JDAF to manage your dialogs:

- *Cross-Cutting Benefits* – Having dialog activity centralized and formalized, and being able to listen to all dialog interaction via the DialogListener mechanism, has significant potential. We use listeners to perform tweaks on various dialog presentations on certain OS, for example. Being able to queue requests allows for add-on dialogs or manipulations to dialogs in a modular fashion. Not only does this provide improvements in maintenance, but it allows cross-cutting functionality to be applied such as logging or other global dialog manipulations such as inserting custom components in messages or file dialogs, or providing custom validation or other trappings.
- *Extreme OS Fidelity* - The differences between dialog presentations from OS to OS are staggering. From message presentation (sub-titled on many platforms), to the ordering of buttons, to the way dialogs are titled and centered. To implement these differences for each platform, for each dialog, is impractical. Let JDAF do it.
- *Consistency* - All requests and responses are somewhat uniform in usage. This ultimately produces more consistent and standardized code.
- *Improved Design* - As a general design principle separating the data from the view is always desirable. While it may seem at first unorthodox, separating the request for the dialog from the dialog execution itself, yet this method has some significant benefits. For example, consider the variation of presentation on different platforms of messages, not just dialogs, but floating alerts, sliding sheets, or other animation that has become a popular trend in modern UI design. If your content is coupled to a JDialog, this becomes impractical to execute. Particularly if the display characteristics are presented differently on different platforms.

As a real world example, on OS X, Apple recommends that you use the native file dialog as opposed to the Swing JFileChooser. Having this separation of concerns allows this behavior to be transparent to your application code.

- *Modularity* - If we where to take a sampling of dialog examples from Swing, we actually would see a separation of concerns in the design. For example, JOptionPane and JFileChooser are both Components, not Dialogs. They each contain a createDialog() method for actual presentation. These methods simply wrap the component in a JDialog for presentation.

  Using a StandardDialogRequest is a similar model. Your dialogs will actually be more modular, and hence reusable, if you simply code the content and wrap it in a StandardDialogRequest, allowing the ApplicationDialogsUI to host the component in an OS dependant manner.

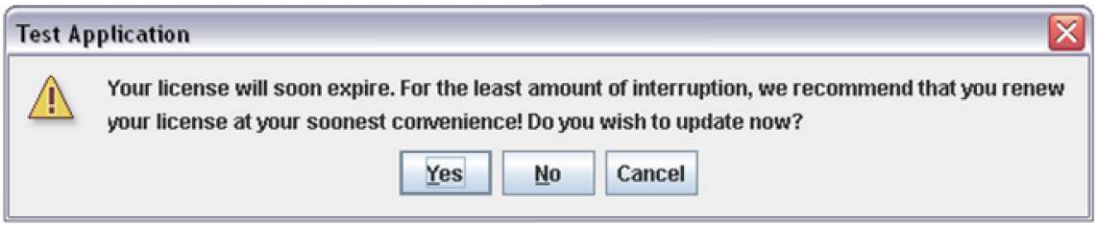
  Furthermore, being able to queue DialogRequests is a powerful way to hide implementation details of the request itself, and a scaleable approach to introducing snap-on standard dialogs via plug-in ApplicationFeatures or by other means, that can be accessed by others in a formal way. For example, Preferences, About, and an Application Exit warning dialogs are queued dialog requests and can be accessed using the constants in the ApplicationDialogsUI. The implementation details of their presentation and construction on not important.

## Presentation Example

Concerning the distinctions in the expression of dialogs, consider the following Yes/No/Cancel confirm dialog:

```
MessageDialogRequest.showConfirmDialog(
"Your license will soon expire. For the least amount of interruption, " +
 "we recommend that you renew \nyour license at your soonest convenience!"+
 "Do you wish to update now?", MessageDialogRequest.YES_NO_CANCEL_DIALOG,
                                MessageDialogRequest.WARNING_STYPE);
```

The following screenshots where produced by this code on the different OS/platforms. We made a similar dialog with JOptionPane. Notice the dramatic difference in presentation. Consider differences in both comparing an equivalent JOptionPane dialog, and JDAF dialogs to each other on the different OS:

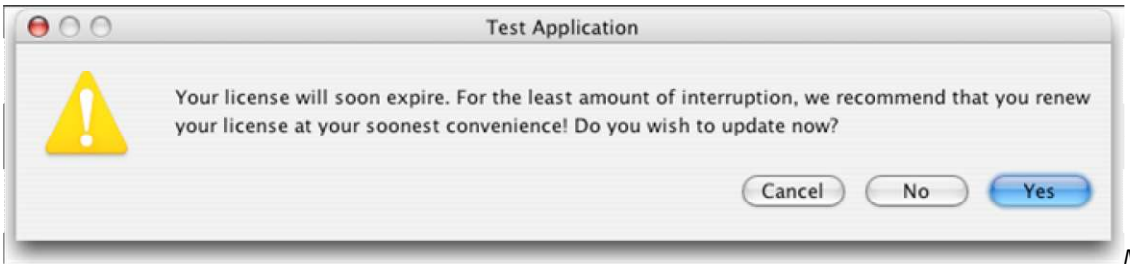



*CROSS PLATFORM – JOPTIONPANE*


*CROSS PLATFORM – JIDE DESKTOP APPLICATION FRAMEWORK*


*GNU/LINUX GNOME – JOPTIONPANE*

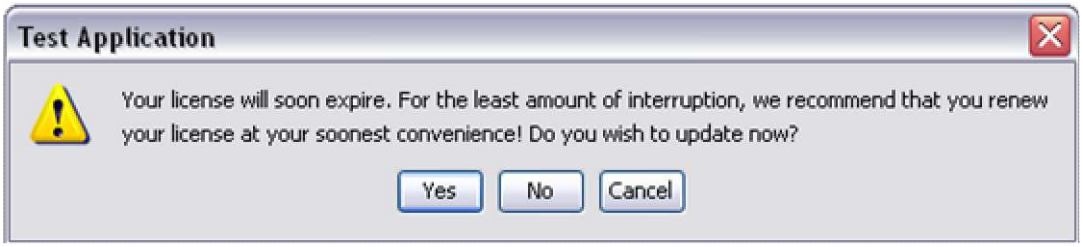

*GNU/LINUX GNOME – JIDE DESKTOP APPLICATION FRAMEWORK*




*MAC OS X - JOPTIONPANE*


*MAC OS X - JIDE DESKTOP APPLICATION FRAMEWORK*

---




*WINDOWS XP –*
*JIDE DESKTOP APPLICATION FRAMEWORK AND JOPTIONPANE ARE SIMILAR*

Notice the differences in dialog size, title formatting, information presentation, and button alignment. JDAF versions follow guideline recommendations. The JOptionPane versions have that homogenous look on all platforms (looks like Windows).

Not obvious in these screenshots is how ApplicationDialogsUI positions dialogs. Positioning is also mentioned in guidelines. Generally, a centered dialog is normal, but on Mac OS X for example, the dialog is

conventionally positioned ¼ down on the vertical. Also, Mac dialogs normally use the application icon in messages. (This is facilitated by using the PLAIN_DIALOG constant, hence, it is important to set your application icons in the ApplicationUIManager.)

### Message Dialogs

MessageDialogRequest provides message dialogs in the style of JOptionPane. In fact, it includes many static usability methods patterned specifically after the static methods in JOptionPane. Message and confirm dialogs are supported with MessageDialogRequest. Constants define general dialog message options. MessageDialogRequest supports the following default dialogs styles:

- OK_DIALOG (Similar to JOptionPane DEFAULT_OPTION)
- OK_CANCEL_DIALOG
- YES_NO_ DIALOG
- YES_NO_CANCEL_OPTION

The following icon styles are supported as well:

- PLAIN_ STYLE
- ERROR_STYLE
- INFORMATION_ STYLE
- QUESTION_ STYLE
- WARNING_ STYLE

When working with the static usability methods, response codes are returned, as opposed to the DialogResponse object (See DialogResponse.getCode()). The response codes will be one of:

- RESPONSE_YES
- RESPONSE_NO
- RESPONSE_CANCEL

Sun, Apple and GNU/Linux-Gnome guidelines recommend the use of subtitled message formatting. Notice the message formatting in the previous examples. However, subtitling messages is an extra burden on the cross-platform developer who attempts to respect OS-guidelines. There is a significant difference in implementation between single-line and subtitled dialog requirements.

To this end, we recommend that all messages be provided in a two-sentence format. The ApplicationDialogsUI will use a BreakIterator to split the sentences for use in subtitled dialogs, and simply use both sentences in non-subtitled dialogs. Alternatively, if one is migrating to JDAF from messages with only single line input, use the dialog signatures that provide for a dialog title. The subtitle logic will demote

the title argument to the primary message line, use the message as the secondary line, and repurpose the application name as the dialog title.

As a passing comment, it appears on most platforms there is a general tendency not to place special text into the title of message dialogs, but to use the application name. To this end we provide dialog method signatures that do not require a title, but use guidelines to provide the appropriate title and formatting.

Finally, all DialogRequest object maintain a properties Map. While this is for general and future use, MessageDialogRequest will use this Map as a source for an ObjectFormat and pass all messages through it. Hence messages can be parameterized using the UL-syntax supported by ObjectFormat. By default the "application" property will always be set with the GUIApplication instance, so any access method in your GUIApplication can be used in your messages.

### File Dialogs

The FileDialogRequest facilitates file chooser dialogs in both Swing and native flavors. Which dialog it uses depend on the guidelines of the OS. As a developer however, the implementation is transparent. Constants define general file dialog options. FileDialogRequest supports the following default dialogs styles:

- OPEN_DIALOG
- SAVE_DIALOG
- CHOOSE_DIRECTORY_ DIALOG

When working with the static usability methods, the selected File(s) are returned, as opposed to the DialogResponse. But there is significant flexibility when using the request itself such as using FileFilters, providing FileViews, etc.

Of special note; when using the FileBaseApplication or just installing the FileHandlingFeature, there is a significant improvement in quality when working with the underlying file system. The feature tailors file filters to those supported by your application. It also provides automatic detection and responses to platform-specific file validations and errors, using over 13+ OS-specific and guidelines-recommended file dialog alerts, warnings, and negotiations. File handling is discussed more in depth later in this document. (See "Making a Document-Based Application – File Mediation".)

### User Dialogs

To present your own dialog in a GUIApplication use the StandardDialogRequest. The basic idea is to simply provide the content for the dialog and allow the ApplicationDialogsUI to present it. The content for the dialog is simply a JComponent.

Dialog buttons are set in the DialogRequest using the *DialogOptions* class. DialogOptions is a special Map implementation that is particular concerned with keying button names. The significance of these keys is that they define the canonical roles of buttons in the dialog. The following keys are defined:

- YES_BUTTON
- NO_BUTTON
- CANCEL_BUTTON
- HELP_BUTTON

There are many static *createXButtons()* methods in DialogOptions that provide default configurations of common dialog button combinations.

- createStandardButtons(dialogType)
- createYesButtons(string)
- createYesNoButtons(string, string)
- createYesCancelButtons(string, string)
- createYesNoCancelButtons(string, string, string)

DialogOptions contains localized default values based on the create method. For example, calling DialogOptions.createYesNoCancelButtons(null, null, null) will populate the Map with the defaults for the keys YES_BUTTON, NO_BUTTON, CANCEL_BUTTON, which are the localized Strings “Yes”, “No”, “Cancel”. Because the keys describe the role of the buttons, these same create methods can be supplied with application-specific synonyms, for example:

```
DialogOptions.createYesNoCancelButtons(“Save”, “Don’t Save”, null);
```

This will yield the same result codes as the standard “yes, no, cancel” configuration. The Jide Commons Layer provides the interface ButtonNames which contains constants for common buttons. DialogOptions supports this interface and directly supports using these constants as values in the DialogOptions. The localized strings will be used in their stead.

To add individual buttons, use the *putButton()* method. You can use *put()*, but if you are using the ButtonNames constants, the putButton() version will do the string substitution for you. For example:

```
DialogOptions buttons =
        DialogOptions.createStandardButtons(OK_CANCEL_DIALOG);
// APPLY is the key, but we can use the key in the value position
// where it will be substituted with the localized button string
buttons.putButton(APPLY, APPLY);
```

You may add a help button to the DialogOptions that integrates with the application *HelpSource.*

```
DialogOptions buttons = …
```

```
buttons.putButton(DialogOptions.HELP_BUTTON, null));
buttons.put(HelpSource.HELP_HINT_PROPERTY_NAME, "mytopic");
```

Internally, the DialogButtonPanel will be used to render the values from the DialogOptions in an OS-guidelines recommended way.

In order to control the validation of your dialogs content, you can set a DialogListener into the StandardDialogRequest. The DialogResponse from the *DialogEvent* passed to your listener, will reflect the value of the pressed button. It is most concise to actually have your Component implement the DialogListener interface, so that your code is better encapsulated. As an example:

```
JComponent myDialogPane = new MyDialogPane();
StandardDialogRequest request = new StandardDialogRequest(
“My Dialog”, myDialogPane, (DialogListener)myDialogPane);
DialogResponse response =
    StandardDialogRequest.showDialog(application, request);
```

Furthermore, there are *showDialog()* signatures will automatically register your Component if it is a DialogListener. Of course, these requests can be queued, and there are static usability methods that make this code even more concise. See StandardDialogRequest in the javadoc.

### *DialogPane*

JDAF provides a convenience JPanel subclass that already implements *DialogListener.* It is unnecessary to implement the listener methods; they are delegated to lifecycle methods. Use these methods instead as they provide a clear interface to dialog behavior when using the managed dialog subsystem.

```
public class MyDiaogPane extends DialogPane {
   protected void initializeComponents(DialogRequest request) {
      // Setup your dialog Components.
   }
   protected void updateComponents(DialogRequest request) {
      // Set/reset the value state of components before
      // showing/resetting for the purposes of cached requests.
   }
   protected void commitComponents(GUIApplication application) {
      // Propagate dialog information to the application.
   }
   protected void validateComponents(DialogRequest request,
                                     DialogResponse response)
                                    throws ApplicationVetoException {
      // Perform validation logic.
   }
```

```
}
```

The GUIApplication can be accessed via *getApplication()*. The *DialogButtonsPanel* can be accessed easily using *getDialogButtons()*. This implementation also ensures that DialogEvents intended for secondary dialogs are ignored.

### *About Dialog*

The ApplicationDialogsUI provides a default About dialog for a GUIApplication as a queued StandardDialogRequest. The request is stored under the key *ApplicationDialogsUI.ABOUT_ DIALOG_REQUEST_KEY*. Default about content is provided by using the application name, version string, and application icon. However, this is minimal. We recommend that a "real" about content be supplied if the project is of significant effect.

To provide your own About dialog content, simply set the content component in the queued StandardDialogRequest, for example:

```
// create a simple panel with image
Image applicationImage = //
JComponent aboutPane = new JPanel(new BorderLayout());
aboutPane.add(new JLabel(new ImageIcon(applicationImage)));

// use a static usability method to set the contentComponent
// of the queued StandardDialogRequest
StandardDialogRequest.setQueuedDialogRequestComponent(application,
        ApplicationDialogsUI.ABOUT_DIALOG_REQUEST_KEY, aboutPane);
```

The AboutAction calls this queued request.

### *Preferences Dialog*

The ApplicationDialogsUI provides a means for defining a preferences dialog for a GUIApplication using the *PreferencesDialogRequest*. PreferencesDialogRequest is simply a specialized StandardDialogRequest. If your application will use a preferences dialog, it should be stored under the key ApplicationDialogsUI.PREFERENCES_DIALOG_ REQUEST_KEY.

Since this is an optional feature, the *PreferencesAction* will need to be added to the applications ActionMap as well.

Here is a long hand example:

```
// Add the action
application.getActionMap().put(ActionKeys.PREFERENCES, new PreferencesAction());

// set a preferences pane in the request
```

```
JComponent preferences = new MyPreferencesPane();
PreferencesDialogRequest request = new   PreferencesDialogRequest(preferences, (DialogListener)preferences);

// queue the request using the static usability method
PreferencesDialogRequest.addQueuedDialog (
        application, ApplicationDialogsUI.PREFERENCES_DIALOG_REQUEST_KEY, request);
```

PreferencesDialogRequest provides a static helper method called *installPreferences()*to do all this work in a single line, minimizing this work:

```
PreferencesDialogRequest.installPreferences(application, preferencesPane);
```

PreferencesDialogRequest provides some options concerning the button configuration, such as whether to show an "Apply" button. Otherwise it shows OK/Cancel buttons only. The ApplicationDialogsUI provides the appropriate title, position and presentation the dialog.

While the storage and retrieval of the preferences can be implemented using the java.util.Preferences node from DesktopApplication getPreferences(), the method of storage is certainly up to your application. You may wish to use the platform *data directory* and use a java.util.Properties object and the *PropertiesFileFormat to* manage the data. The data directory is available from the application *getDataDirectory().*

### *PreferencesPane*

*PreferencesPane* is a *DialogPane* subclass that can be used to provide a preferences or options dialog that fits well with OS guidelines. On Windows OS, there is an apply button and on Max OS X there are no buttons, but the apply ocurrs when the dialog closes. These details are managed.

```
PreferencesPane pane = new PreferencesPane() {
    protected void initializeComponents(DialogRequest request) {
    }
    protected void updateComponents(DialogRequest request) {
    }
    protected void updatePreferences(GUIApplication application) {
    }
    protected void validateComponents(DialogRequest request,
                                      DialogResponse response)
                                     throws ApplicationVetoException {
    }
};
```

### Legacy Dialogs

To interface pre-existing dialogs to the ApplicationDialogsUI, including ones from the Jide libraries, we provide the *CompatibleDialogRequest*. This is simply a wrapper request that places your dialog in the managed UI pipeline where it is eligible for *DialogEvents*. Here is an example using the static method:

```
DialogRequest response =
        CompatibleDialogRequest.showDialog(application, new MyDialog());
```

To gain tighter integration with the framework, you can set a DialogListener in addition and use the *dialogClosed* event to set up the DialogResponse values with values from your dialog.

## Working with Menus

GUIApplication manages menu bars in the framework. It provides standard File, Edit, Window, and Help menus. We call these "standard menus" because they are common menus recommended by all OS guidelines, though they may have very different layouts. To define additional application menus, or customize the standard menus, the developer simply installs a *MenuBarCustomizer* in the GUIApplication.

You may also use the AutoInstallActionsFeature to install Actions into menus without interfacing with the menus at all.

### MenuBarCustomizer

To add your own application specific menus or customize standard menus, simply add a MenuBarCustomizer to the GUIApplication using *addMenuBarCustomizer()*. A MenuBarCustomizer is called every time a new menu bar needs to be generated. A MenuBarCustomizer has two methods:

```
public void customizeStandardMenu(String, JMenu, ApplicationMenuBarsUI);
public JMenu[] createApplicationMenus(ApplicationMenuBarsUI);
```

The *customizeStandardMenu()* method will be called for each standard menu. The first argument is the menuID, which defines which menu is being passed. All menuIDs can be found in the MenuConstants interface. The following constants define the standard menus:

- FILE_MENU_ID
- EDIT_MENU_ID
- WINDOW_MENU_ID
- HELP_MENU_ID

The JMenu instance is passed in for each standard menu. Actions can be accessed from the ApplicationMenuBarsUI object via getAction(actionKey)as a convenience. Standard menus are arranged internally into MenuGroups. If you want to add Actions to a standard menu, we recommend that you look-

up the respective *MenuGroup* and add to it instead of adding to the menu directly. In this way actions can be added in a guidelines-compliant manner. MenuGroups are explained later in this section.

For example, most guidelines specify where "user" items should go in a particular standard menu. Since this location is dependent on the OS, using the respective MenuGrouping allows the item to be placed in a compliant manner. The following code adds an item from a registered Action called "export" to the user area of the File menu:

```
public void customizeStandardMenu(String id, JMenu menu,
                        ApplicationMenuBarsUI menusUI) {
    if(id.equals(FILE_MENU_ID)) {
      menusUI.addMenuItem(FILE_USER_GROUP_ID, menu, "export");
    }
}
```

The *createApplicationMenus()* method of the MenuBarCustomizer should return an array of application-specific menus. The ApplicationMenuBarsUI will make sure the menus are placed properly in the menu bar. For example:

```
public JMenu[] createApplicationMenus(ApplicationMenuBarsUI menusUI) {
   JMenu menu = menusUI.defaultMenu(MY_MENU_ID, "MyMenu");
   menu.add(menusUI.getAction("myAction1"));
   menu.add(menusUI.getAction("myAction2"));
   // ...
   return new JMenu[]{menu};
}
```

We recommend that you use the ApplicationMenuBarsUI factory methods *defaultMenu(), defaultMenuItem()*, *defaultRadioButtonMenuItem()*, and *defaultCheckBoxMenuItem(), etc*. where appropriate when constructing menus. Each of these methods accepts Actions or Action keys. While using these methods is elective, creating menus in this manner ensures that the application scales properly on various platforms, and that the ApplicationMenuBarsUI can provide the most appropriate implementations. The application code ends up more concise as a bonus.

### General Menu Options

While the OS-specific ApplicationMenuBarsUI subclass will likely make the most appropriate settings for your application concerning menus, here are a few options of interest.

- Use *setUsesMenuBars(false)* to not use menubars at all, or to manage them yourself. You can use a *WindowCustomizer* to install the JMenuBar. We do not recommend this, as it defeats the guidelines compatibility features of JDAF. We recommend using the MenubarCustomizer instead.

However, there are always exceptions. Note that on Mac OS X there will always me a minimal menubar.

- Use *setShowIcons(true)* to force icons to be shown in menus. Some ApplicationMenuBarsUIs will ignore this option if icons are specifically prohibited; others turn it on by default.
- Use *setUseHelpMenu(false)* to suppress the Help menu. This is not recommended as some platforms use this menu for the "About" command.
- Use *setUseWindowMenu(false)* to suppress the Window menu. If the application is set for a single window, this menu is suppressed by default.

### A Note about Edit Menus

The standard Edit menu provides Actions for undo, redo, cut, copy, paste, select all, and clear/delete. All Actions except undo/redo are *ComponentAction* instances; i.e. they operate by executing the respective Action from the ActionMap of the focused Component. Undo and Redo however interact with the focused DataModel's UndoManager, which requires UndoableEdits to exist to be effective. So, there is a degree of cooperative functionality required between the framework and the developer.

Three issues are involved in facilitating this Edit functionality in your application:

1) Conditionally enabling the respective Edit Actions in the Components ActionMap.
2) Providing the edit functionality via the Component TransferHandler.
3) Providing UndoableEdits for any edits.

Our current solution provides String-based editing for JTextComponent, JList, JTable, and JTree. If using the default Swing model implementations *and* using String values only, we will provide edit Actions and Undo/Redo behavior. By "default model" we mean DefaultListModel, DefaultTableModel, DefaultTreeModel, for JList, JTable, and JTree, respectively. Swing provides undo behavior for JTextComponent already, which we tap into automatically[7]. If the developer can live with these constraints, we can provide base edit functionality for these components. Otherwise, the tooling will simply enable the Components Actions if you provide them, based on the selection states.

To bind a Component with the Edit menu subsystem, simply call the static *EditUtils.installEditTooling(Component)* for a given Component. This can be done when setting up your DataView. There is a method signature for each of the four major component types. If you are using DataViewPane, it includes the method installEditables() that automatically installs this tooling on all

[7] To keep a JComponent from posting UndoableEdits, set the client property EditUtils.OMIT_UNDO_REDO to Boolean.TRUE

compatible Components in the view contents. You can cause any component to not be included in the edit system by supplying

We are currently in R&D for a solution that will minimize the work the developer needs to do to support these Edit menu features. In the meantime the developer will want to implement cut, copy, paste, select all, and delete/clear actions for each component as desired, and place the action in the Component's ActionMap. If these actions are missing, the Edit menu items will disable appropriately when the component is focused.

## Managed UI Menus

As comfortable as one may feel defining menus in Swing, challenges arise when approaching application menuing from a truly cross-platform and OS-guidelines sensitive perspective. The scheme by which menubars are implemented ends up being highly OS-dependant due to the differences in recommended menu item positions, windowing-styles, and Java implementation/OS issues.

The ApplicationUIManager uses an OS-specific ApplicationMenuBarsUI subclass to manage application menus. It provides standard menus depending on the OS guidelines. To understand how this adaptive system works, a few objects are worth noting: *MenuGroup* and *MenuGrouping*.

### *MenuGroups*

Recommended menus and their contents are different between OS'. A MenuGroup helps to manage this by facilitating a span of menu items in a menu that represent certain functionality, such as "save items" or "help items". Each standard menu is segmented into these MenuGroups. By adding items to a MenuGroup, instead of a menu directly, the application UI can handle where the items are laid-out in the underlying menus. By working with the MenuGroup the developer can define menus in a uniform way that otherwise would have required special platform-dependant coding.

MenuGroup can layout a group of items normally, one after the other "in-line", or it can layout the items in a submenu. One example of this feature is in the implementation of the "Recent Documents" feature (See "Making a Document-Centric Application"). On Windows XP, recent documents are displayed inline in the File menu. But on other platforms, they are defined in a submenu. To the developer it doesn't matter. Interfacing with the MenuGrouping is the same, because the underlying MenuGroup manages the details.

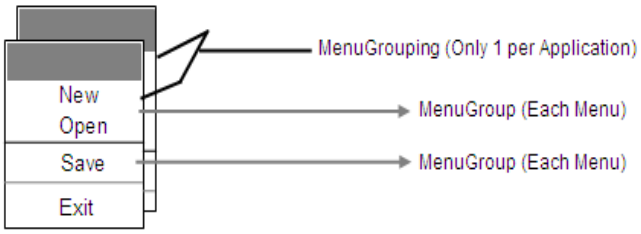
A MenuGroup will manage separators transparently so that items in each group are properly delineated. This can be turned off by calling *setUseSeparators(false)* on the MenuGroup.

MenuGroups come in three-flavors, distinguished by the types of menu items they produce. MenuGroup just uses standard menu items. CheckBoxMenuGroup will create checked menu items. RadioButtonmenuGroup will create radio button menu items and manage them with a ButtonGroup for exclusive selection.

A MenuGroup object requires a reference to a "root" menu and an "anchor item". The anchor item determines how and where the menu is laid out. The anchor item can be one of the following:

| Anchor Item | Positioning Logic |
|---|---|
| null | menu items will appear at the top of the root menu |
| JMenuItem | menu items will appear underneath the menu item |
| JMenu | menu items will appear in a sub menu of the anchor item |
| MenuGroup | menu items will appear after the last item in the anchor MenuGroup |

This last option is important. Complete virtual menu structures can be created simply by linking MenuGroups, even if the groups are empty (and the menu is empty). A MenuBarCustomizer may use the MenuGroup structure, via the MenuGroupings, to fill in each menu with file-specific items.




### *MenuGroupings*

In an MDI environment, there is a single menu bar. In an SDI environment, there are as many menu bars as there are document windows. Some SDI-implementations require special tricks to keep the menu bars maintained properly, such as Mac OS X. Hence, in most situations, there are many menus bars that need to be maintained and synchronized in an application.

JDAF provides standard File, Edit, Window, and Help menus and associated Actions. In most OS guidelines specifications, there are predefined, or at least conventionally understood, positions where menu items should go in a given menu. While it is true that menus from OS to OS will have items that perform similar functionality, the difference between the layouts of these functions can be dramatic.

To handle these challenges, each standard menu is partitioned into multiple MenuGroupings. A MenuGrouping facilitates the synchronization of menus across all menu bars. Mutator methods that add, insert, remove, and replace menus items in a MenuGrouping affect all menus in all menu bars in the application UI. For example, if an Action is added to a MenuGrouping, a menu item is added to all menu bars in the application. A menu can have more than one MenuGrouping to manage particular sets of menu items.

MenuGroupings internally do not access the JMenu directly; instead they manage a collection of MenuGroups which represent item groups in each menu of the application. A MenuGrouping manages a MenuGroup for each menu it represents.

MenuGroupings can be accessed from the ApplicationMenuBarsUI object using the method *getMenuGrouping(groupingID)*. A MenuGroup can be accessed from a MenuGrouping using a particular JMenu instance as a key via *getMenuGroup(JMenu)*. And you can discover what MenuGroups a menu has using *getMenuGroups(JMenu)*. The MenuGroup for a particular menu instance in a particular MenuGrouping can be accessed from the ApplicationMenuBarsUI using *getMenuGroup(groupingID, JMenu)*. See the javadoc for other methods that allow access to the menuing system.

MenuGrouping IDs are defined in the *MenuConstants* interface:

| | |
|---|---|
| FILE_NEW_GROUP_ID | EDIT_UNDO_GROUP_ID |
| FILE_OPEN_GROUP_ID | EDIT_EDIT_GROUP_ID |
| FILE_SAVE_GROUP_ID | EDIT_EDIT_SPECIAL_GROUP_ID |
| FILE_PRINT_GROUP_ID | EDIT_USER_GROUP |
| FILE_EXIT_GROUP_ID | MENU_PREFERENCES_GROUP_ID |
| FILE_USER_GROUP_ID | HELP_ABOUT_GROUP_ID |
| WINDOW_ARRANGE_GROUP_ID | HELP_HELP_GROUP_ID |
| WINDOW_ARRANGE_SPECIAL_GROUP_ID | WINDOW_ACTIVATE_GROUP_ID |

### *Using MenuGroupings with Custom Menus*

While it is important to use the ApplicationMenuBarsUI object when working with the standard menus, it is certainly up to the developer whether managing custom application menus using the MenuGroups/MenuGroupings sub-system is desirable. If you decide go this route here are some tips:

1. At pre-run time, create only one MenuGrouping for each functional “span” of items in your application. Add your MenuGrouping to the ApplicationMenuBarsUI via *putMenuGrouping()* with the key that can be used to identify it when adding menu items.
2. In your MenubarCustomizer add MenuGroups to the MenuGrouping for each menu created. Then add Actions or menu items to the MenuGroup.

Alternately you may desire to use the *AutoInstallActionsFeature*, which will install menu items and even create menus automatically based on Action properties.

### OpenWindows

*OpenWindows* is a special MenuGrouping used internally in the standard Window menu. Each OS supports this MenuGrouping under the WINDOW_ACTIVATE_GROUP_ID key. It uses CheckBoxMenuGroups to represent each window. As the ApplicationWindowsUI opens a new DataView window, an *ActivateWindowAction* will be added to the application's ActionMap. ActivateWindowActions focus the respective DataView, which brings its window to front.

The developer shouldn't interface with these Actions directly. An OpenWindowsFormatter can be set in thr ApplicationMenuBarsUI to provde custom formatting of the appearance of these DataModels in the application. By default, the ApplicationMenuBarsUI will install an appropriate *OpenWindowsFormatter* that best suits the OS.

## Working with ToolBars

GUIApplication automatically manages application toolbars. The framework provides one toolbar called the "standard toolbar", as recommended by the respective OS guidelines. To define additional application toolbars, or customize the standard toolbar, the developer simply installs a *ToolBarCustomizer* in the GUIApplication. But first, we will give a brief explanation of how toolbars are managed.

The ApplicationUIManager uses an OS-specific ApplicationToolBarsUI subclass to manage application toolbars. Whether toolbars appear is OS-dependant. For example, application toolbars are disabled by default on Mac OS X because they are application dependent and non-standard[8]. However on Windows XP they are common. The Cross-Platform Application UI also supports toolbars. This can be manually controlled with the pre-run option *setUsesToolbars()*, for example to remove toolbars from the application. Normally, the managed UI determines whether to provide toolbars or not.

You may also use the AutoInstallActionsFeature to install Actions into toolbars without interfacing with the toolbars at all.

### General ToolBar Options

The ApplicationToolBarsUI can implement different toolbar styling. The style is dependant on OS guideline recommendations, or deduced from popular appearance studies. While the ApplicationToolBarsUI decides on the appropriate default style for the OS, styles can be applied via the *setToolBarsStyle*() method as a pre-run option. This method takes one or more style constant flags. The following style constants are available:

---

[8] Use the pre-run option *setToolBars(true)* to turn on toolbars for OS X. On Leopard, JDAF will prepare the toolbar to work as a "unified toolbar", but to complete this effect you must call the *OSXProperties.setBackgroundEffect(OSXProperties.UNIFIED_TOOLBAR)* prior to instantiating the GUIApplication. You may then use *OSXProperites* when building your UI to create compliant toolbar buttons.

- FLOATABLE_TOOLBAR_STYLE – Toolbars will be floatable
- ROLLOVER_TOOLBAR_STYLE – Toolbars will use the rollover function in Swing
- LARGE_TOOLBAR_STYLE – Toolbars will use large icons
- TEXT_TOOLBAR_STYLE – Use only with LARGE style to show text underneath
- UNIFIED_TOOLBAR – Mac OS X-only option to set the Unified toolbar mode

If LARGE_TOOLBAR_STYLE is not set, then the default small icon toolbar is assumed. Setting the style to 0 will clear any special style settings, resulting in the default toolbar. If the style is to be changed manually, we recommend this be done in an OSApplicationCustomizer on an OS by OS basis. For example, the *WindowsXPApplicationToolBarsUI* simply uses scaled versions of the small icons for its large style, which isn't very appropriate. You can however replace the icons and use the large style.

A special feature of the ApplicationToolBarsUI is the integrated use of the JIDE Action Framework. The library can be used manually on any OS by setting the pre-run option *setUseJIDEActionFramework(true)* in the ApplicationUIManager. Note: This is ignored if the OS does not support application toolbars.

### ToolbarCustomizer

To define application-specific toolbars or customize the standard toolbars, install a *ToolBarCustomizer* in the GUIApplication using *addToolBarCustomizer()*. A ToolBarCustomizer is called every time a new toolbar bar needs to be generated. ToolBarCustomizer has three methods:

```
public void customizeStandardToolBar(String toolbarName,
                Container toolbar, ApplicationToolBarsUI toolbarsUI);

public String[] getToolbarNames();

public void createApplicationToolBar(String toolbarName,
                Container toolbar, ApplicationToolBarsUI toolbarsUI);
```

The *customizeStandardToolBar()* method will be called for each standard toolbar. Currently there is only one, the "*Standard*" toolbar, which is passed as the first argument. The next argument is the actual toolbar component, and finally, the ApplicationToolBarsUI object. ApplicationToolBarsUI has a convenience method *getAction(actionKey)* that accesses Actions from the application ActionMap. You can also use the *addToolBarButton(Container, String)* which will lookup the Action and add the Toolbar button. Or use *defaultToolBarButton().* We recommend when creating toolbars that the developer use these factory methods. These allow the ApplicationToolBarsUI to provide the most appropriate implementation.

The *getToolBarNames()* method should return the displayable names of any additional toolbars that need to be created. ApplicationToolBarsUI catalogs all toolbars for later retrieval using toolbar names. The names may also be used in the UI as well.

The *createApplicationToolBar()* method will be called for each name returned from *getToolBarNames()*. It passes the a name and toolbar component to add buttons to. If using the JIDE Action Framework, a CommandBar instance will be input, otherwise a JToolBar is passed. To determine if the JIDE Action Framework is being employed, use the ApplicationUIManager method *isJIDEActionFrameworkUsed().* But using the factory methods is the most resilient way to compose the toolbars in a type independent manner. For example:

```
public String[] getToolbarNames() {
   return new String[]{"Format"};
}

public void createApplicationToolBar(String toolbarName, Container toolbar,
                              ApplicationToolBarsUI toolbarsUI) {
      if(toolbarName.equals("Format")) {
            toolbarsUI.addToolBarButton(toolbar, "bold"); // uses Action name
            toolbarsUI.addToolBarButton(toolbar, "italic");
            toolbarsUI.addToolBarButton(toolbar, "underline");
      }
}
```

Some ApplicationToolBarUI implementations contain special methods for OS-specific behavior. This is true on the Mac. Mac Guidelines allow for multiple styles of toolbar buttons. You can use the methods in the MacOSXApplicationToolBarsUI to “format” your toolbar buttons. For example, to create a “segmented” style set of buttons you would do this in your ToolbarCustomizer:

```
...
public void createApplicationToolBar(String toolbarName,
            Container toolbar, ApplicationToolBarsUI toolBarsUI) {

            if(toolbarName.equals("Toolbar1")) {
              toolBarsUI.addToolBarButton(toolbar, "back");
              toolBarsUI.addToolBarButton(toolbar, "forward");

             ((MacOSXApplicationToolBarsUI)toolBarsUI).makeSegmentedToolbar(
                 toolbar, MacOSXApplicationToolBarsUI.BUTTON_STYLE_BEVEL, true);
}
...
```

It is normally unnecessary to access the toolbars outside of a ToolBarCustomizer. But, if you need to, toolbars can be accessed using the ApplicationToolBarsUI method *getToolBars(toolbarName)*. This returns an array of Containers of all versions of that Toolbar in each open window. The framework treats all toolbars as Containers so casting may be required between a CommandBar and a JToolBar if the JIDE Action Framework is being used.

## ApplicationFeature

JDAF provides a facility for imparting functionality to a GUIApplication modularly, without subclassing the GUIApplication class itself; the *ApplicationFeature*.

ApplicationFeature allows for cross-cutting application capabilities to be encapsulated and reused in different GUIApplications. It can be approached as a "sub-controller" that partitions your application's functionality into maintainable pieces, or it can be approached as a simple a way to add reusable functionality to any application.

> *While this sounds like a "plug-in" facility, it's not necessarily because it does not facilitate class loading or dependency checking. However for the sake of functionality, it is similar in that it adds functionality in a modular fashion, and it is most definitely the best strategy for developing a GUIApplication in modular, reusable way.*

To install an ApplicationFeature simply add one to a GUIApplication via *addApplicationFeature()*. To remove one, call *removeApplicationFeature()*. You need only implement *install()*. ApplicationFeatures may be accessed by class via *getApplicationFeature()*. (Only one instance of a given class may be added to a given application.)

The developing of an ApplicationFeature is only as trivial as the functionality it seeks to provide. ApplicationFeature is essentially a super-adapter of all the listeners and customizers available for a GUIApplication, rolled into one object. An ApplicationFeature implements:

- ApplicationGUILifecycleListener
- DataModelListener
- DataViewListener
- DialogListener
- WindowCustomizer
- MenuBarCustomizer
- ToolBarCustomizer

Hence, the object can perform significant configuration on all aspects of the desktop application. The feature is automatically registered to listen to all these events. You may override *installListeners()* to optimize this aspect.

An ApplicationFeature can also be used to provide API reflecting the functionality they deliver, thereby providing a clean interface for other developers.

Besides the adapter listener and customizer methods, ApplicationFeature provides two lifecycle methods; *install()* and *uninstall()*. You're only required to implement *install()*. These lifecycle methods are called when the object is added and removed from the GUIApplication, respectively. The *install()* method hence occurs during the time that one would normally set pre-run options when configuring a GUIApplication. For example, you could install resources such as DataModelFactory, DataViewFactory, and GUIApplicationAction objects, or any other resources and modifications, as appropriate to the feature.

If the install method fails for any reason, the normal behavior is to throw a RntimeException. If your ApplicationFeature need not install successfully, set *setPassiveExceptionHandling(true)* and it will be logged and bypassed instead of stopping the application.

*We recommended adding ApplicationFeatures immediately after instantiating the GUIApplication, as they are intended to extend GUIApplication functionality.*

The *uninstall()* method should attempt to remove/restore any resources that where installed via *install()*. The *uninstall()* method is only called if the ApplicationFeature is removed programmatically, which may not be likely during the life of the application. So this behavior may be optional depending on your usage. Be sure to document whether the feature can be uninstalled at runtime. The default implementation throws a FeatureException that prevents uninstallation. However, if you require clean up for your feature when the application exists call *setUninstallOnExit(true)* and the feature will be uninstalled after the UI shuts down successfully.

JDAF provides the following ApplicationFeatures:

- *AutoInstallActionsFeature* – Designed to automatically populate menubars and toolbars using Action properties.
- *FileHandlingFeature* – Designed to provide robust file handling functionality
- *FramedApplicationFeature* – Designed to facilitate application styles having auxiliary side components, using JDAF MVC architecture.
- *DockingApplicationFeature* – Designed to integrate the JIDE Docking Framework into JDAF MVC architecture.
- *GlobalSelectionFeature* – Designed to provide a global selection mechanism so that GUIs can be aware of a globally selected object and when that selection changes.
- *DialogLoggingFeature* – Redirects Java Logging API Loggers to a dialog. This dialog has a save button for storing the error to a file.

When creating an ApplicationFeature be sure to be sensitive to the possible effects of other ApplicationFeatures or individual listeners and customizers. Currently there are no constraints defined to prevent features from "rolling over" one another due to the open nature of the listening and customizing mechanisms. It is advised that the class comments of an ApplicationFeature make clear any behavior that may be considered "broad in stroke", and provide a way to circumvent extreme behavior.

## Making a Document-Centric Application

A file-based application, a.k.a. the "document-centric" application, is the most common and popular style of desktop application. This type of application is primarily concerned with providing an environment that works with data stored and retrieved from files.

JDAF provides a GUIApplication subclass specifically for this type of application, called the FileBasedApplication. This should be used if your application is file-centric. FileBasedApplication installs the FileHandlingFeature to facilitate its functionality and adds some usability methods for good measure. However, an equivalent application can be configured by using a GUIApplication instance and simply adding the FileHandlingFeature. This allows file handling capabilities to be "mix-in" if you have other `DataModel` needs. With this understanding, we will focus on the `FileHandlingFeature` and the capabilities it adds to a `GUIApplication`.

With the `FileHandlingFeature`, the File menu is configured with all the functionality necessary for working with files; New, Open, Save, Save As, and a Recent Documents facility, among other features. The actual "File" menu configuration is based on OS-guideline recommendations.

A *FileDataModelFactory* and *FileDataViewFactory* are installed to handle the creation of each virtual "document" (FileDataModel + DataView). FileHandlingFeature uses *FileFormat* objects to define what data formats the application should support. One simply uses the FileHandlingFeature instance when providing the FileFormat and a DataView that can display and manipulate data in that format. The following FileHandlingFeature methods provide this functionality:

- public void addFileMapping(FileFormat fileFormat, Class dataViewClass)
- public void removeFileMapping(FileFormat fileFormat)

The FileFormat and DataView class are associated so that files encountered with the given FileFormat are displayed properly. Internally, the FileFormat is registered with both the FileDataModelFactory and the FileDataViewFactory.

## A Plain Text Editor Application

JDAF provides some basic FileFormats. One is the TextFileFormat, which reads and writes plain text. As an example of a file-based application, consider this very simple example of a DataView implementation, "TextView", which manages plain text. We will use it to facilitate a plain text editor.

```
Public static void main(String[] args) {
    FileBasedApplication application = new FileBasedApplication("Text Editor");
    application.addFileMapping(new TextFileFormat(), TextView.class);
    application.run(args);
}
```

Following is the code for a TextView:

```
public class TextView extends DataViewPane {
   private JTextArea textArea;

   private DocumentListener docListener;

       protected void initializeComponents() {

           // sets the window size
           setPreferredSize(new Dimension(550, 400));
           setBackground(Color.WHITE);
           setBorder(null);

           // init text area
           textArea = new JTextArea();
           textArea.setFont(textArea.getFont().deriveFont(12f));
           textArea.setWrapStyleWord(true);
           textArea.setLineWrap(true);
           docListener = new DocumentListener() {
                 public void insertUpdate(DocumentEvent e) {
                       makeDirty(true);
                 }
                 public void removeUpdate(DocumentEvent e) {
                       makeDirty(true);
                 }
                 public void changedUpdate(DocumentEvent e) {
                       // unnecessary for plain text
                 }
           };
```

```
            textArea.getDocument().addDocumentListener(docListener);
            textArea.setBorder(BorderFactory.createEmptyBorder(4,10,4,4));
            JScrollPane scrollPane = new JScrollPane(textArea);
            scrollPane.setBorder(null);
            scrollPane.setOpaque(false);
            add(scrollPane);

            // install editing features
            EditUtils.installEditTooling(getApplication(), textArea);
    }

     public void updateView(DataModel dataModel) {
        FileDataModel fileDataModel = (FileDataModel)dataModel;
        textArea.getDocument().removeDocumentListener(docListener);
            textArea.setText(fileDataModel.getData() != null ?
               fileDataModel.getData().toString() : "");
            textArea.setCaretPosition(0);
            textArea.getDocument().addDocumentListener(docListener);
      }

    // set the data to save
    public void updateModel(DataModel dataModel) {
        FileDataModel fileDataModel = (FileDataModel)dataModel;
        fileDataModel.setData(textArea.getText());
    }

    //This implementation just prints the TextArea component.
    public Pageable getPageable() {
          return new ComponentPageable(textArea);
    }
    }
```

TextView uses a JTextEditor to edit the text data for the application. The DataModel, which is technically a FileDataModel, maintains the text data via its TextFileFormat member. FileHandlingFeature makes sure the binding between the DataModel and the DataView is valid by associating the TextFileFormat in both factories. Hence, the DataView can be confident that the data will be text.

## Working with FileFormats

FileHandlingFeature provides data support for files by use of the *FileFormat* class. FileFormat is an abstract class whose function is to bind data to and from the file system. A FileFormat can have a file type (ex. 'txt'), file type description (ex. 'Text'), an icon, and a semantic name.

FileFormat treats data generically as a Java Object for the broadest support of data. New data type support is trivial to introduce as long as the data format can be approached in Java by one who understands the format, or with libraries that do so. To allow an application to support a data format, subclass the FileFormat and implement these three methods:

- *readData(File)* – A FileFormat must be able to read the supported data from a file
- *writeData(File, Object)* – A FileFormat must be able to write data to the file in the supported format
- *makeDefaultData()* – A FileFormat must be able to provide a default version of the data it supports.

If the format is read only override the *isReadOnly()* method and return true.

### Provided FileFormats

JDAF provides five ready-to-use FileFormat implementations:

| FileFormat | Comment | Data Type |
|---|---|---|
| ObjectFileFormat | Reads and writes a plain Java Object (.data[9]) | Serializable |
| PropertiesFileFormat | Reads and writes in the Java Properties format (.properties) | Properties |
| TextFileFormat | Reads and writes plain text (.txt) | String |
| XMLFileFormat | Reads and writes XML (.xml) | org.w3c.dom.Document \| String[10] |
| XMLObjectFileFormat | Reads and writes a Java Object as XML (.xml) | Object/Java Bean (Uses XMLEncoder and XMLDecoder) |

[9] When using the ObjectFileFormat the .data extension is a somewhat nebulous choice. This format is not guaranteed to read any .data file. We encourage the developer to brand this kind of file so that it is appropriate and unique to the application.

[10] XMLFileFormat native data is a Document. But the format class provides methods for working with the XML as a String also.

### FileFormat Semantics

A FileFormat implementation has somewhat of a virtual or implied functionality because file extensions have a virtual nature to them. While '.txt' is the conventional extension for TextFileFormat, as is '.xml' for the XMLFileFormat, these are conventional. There is nothing stopping one from defining a file as .dat, and storing text in it. It is a matter of sensitivity to other programs and the branding needs of your own files.

To the user, the data they are working with may be perceived as a "Project", but internally the ObjectFileFormat is being used, or perhaps a "Document", when the TextFileFormat is being used. "Drawing" or "Image" are other possible impressions.

This notion of the perceived format and the actual format we call "file format semantics". The FileFormat classes attempt to adopt the most appropriate extensions and descriptions by default. But this extension can be changed in the constructor to suit application semantics. Also, constructors allow for a semantic name to be defined as well. This information is propagated the FileDataModel's semantic name, and hence to the UI. And the extension is used in file filtering in open and save dialogs.

The semantic name is inherited from the associated FileFormat. You can use the FileHandlingFeature method *setUseSemanticNameForNewMenus(true)* to specify whether a FileDataModel semantic name shows in the "New" menu items when there are more than one FileFormat loaded. Generally this is option is false, but may be desired otherwise the plain file description will be shown.

Some formats you may wish to view/open but would never originate in the application itself using the NewFileAction. For these formats, call *setCanCreate(false)*. This keeps the format from generating a NewFileAction, but it can still be opened.

### FileFormat Conversions

If FileFormat conversions are required by the application, the FileFormat to be converted to must be subclassed and the following two methods must be implemented:

- *isConvertableTo(FileFormat)* – Return whether the input FileFormat can be converted to *this* format.
- *convertData(Object, FileFormat, File)* – Called to perform the conversion of the Object data in the input FileFormat to *this* format, and written to the input File.

Additionally, call *setAllowsFormatConversions(true)* in the FileHandlingFeature if convertible formats are used. This causes the various UI elements to facilitate data format conversions by exposing convertible formats in menus and dialogs, as appropriate.

When using a format with *isReadOnly()* returning true, in order to save the data you must implement at least one convertible FileFormat that can convert the read only format.

### Other FileFormat Uses

FileFormats are used by the FileHandlingFeature by virtue of adding a "file mapping". This mapping involves passing the FileFormat to the FileDataModelFactory and FileDataViewFactory and ultimately to a FileDataModel when such data is loaded or saved. Once the mapping association is made, the actual read and write operations are managed by the application automatically.

However, there is no reason FileFormats can't be used individually. For example, to read or write properties, xml data, or some other auxiliary file tasks. Simply instantiate one and use the *readData()* and *writeData()* methods as needed.

Utility save and open dialog tasks, outside of the managed UI framework can also be facilitated using FileFormats. Simply use the FileDialogRequest class. These methods take FileFilters as one of their arguments. A *FileFormatFileFilter* can be used to wrap a FileFormat object for use in these dialogs. This provides a higher quality of file dialog negotiation than using regular FileFilters, because the FileFormat provides a greater degree of information that the FileFilter alone does not portray.

## CommandLine File Loading

The FileHandlingFeature facilitates opening of files from the CommandLine automatically if the program arguments have been captured via the *CommandLine* class. The FileDataModelFactory method *getCommandLineFiles()* returns Files extracted from the CommandLine filtered by what FileFormats have been set in the factory. Internally, a *File.class* ObjectConverter mapped to a *FileFormatFileConverter* is used in the GUIApplication ConverterContext. This is used to detect files on the command line. Hence, only files that have a resolvable FileFormat and exist in the file system will be opened.

## Working with Document-Centric Actions

FileHandlingFeature installs, and in some cases replaces, some GUIApplicationActions specifically designed for file-handling, as defined below:

| Action | Comment |
|---|---|
| NewFileAction | Creates a new FileDataModel using the provided FileFormat. Replaces the Action under ActionKeys.NEW. |
| OpenFileAction | Opens a file using the FileFormat as a filter. Replaces the Action under ActionKeys.OPEN. |
| RevertAction | Reverts the DataView to the data in the File. Provides a revert alert |

| | |
|---|---|
| | dialog. |
| SaveAsAction | Performs direct and indirect "Save As" operation. Allows format conversion if *isAllowsFormatConversions()* is true, and the file FileFormats allow for conversion. |
| SaveAllAction | Saves all open DataModels |

NewFileAction allows for a FileFormat to be defined as the criteria. FileHandlingFeature will create a NewFileAction for every FileFormat, and present them in a submenu, controlled by a MenuGrouping under MenuConstants.FILE_NEW_GROUP_ID.

OpenFileAction presents an "Open File" dialog. One or more *FileFormatFileFilter* objects are used to wrap the FileFormats and present them in the open dialog as available file types. If the file validates properly (see next sub-section), the selected File will be used in a call to the application's *openData(File)* method. The FileDataModelFactory takes over from there. It uses the File to select the proper FileFormat, binds the File and FileFormat to a new FileDataModel, and uses the FileFormat to read the file. Note, if *isAllowsMultipleOpens()* is true, multiple files can be opened from the Open dialog. This simply repeats this process for each file.

SaveAsFileAction presents a "Save" dialog. It makes use of DataModelListener events to detect when a new File needs to be saved, and hence is able to invoke the Save dialog under indirect conditions, as well as an invocation from the File menu. The *setAllowsFormatConversions()* method in FileHandlingFeature actually delegates to this Action. If *isAllowsFormatConversions()* is true, all FileFormats that are convertible to the saving FileFormat will be available in the dialog. If one of these other formats is selected by the user, the framework takes care of performing the calls to the *convertData()* method, and updates the DataModel with the new FileFormat. Otherwise, only the saving DataModel's FileFormat is made available, via a FileFormatFileFilter.

## File System Mediation

One of the hidden risks of casual file system interaction in Java is the lack of file validation and application integration in Swing's JFileChooser. For example, JFileChooser has no logic to check for potentially harmful read/write conditions such as locked files, externally modified files, illegal characters on the OS, illegal filenames on the OS, or whether the user names a file after one that already exists (file replace). The situation is only marginally better when using the native FileDialog. Only *some* validations are performed, but not many, and not the same validations across all platforms.

Additionally there is the application interaction with the file system, such as save alerts and enforcing that a file extension is actually attached to the file name when writing. These are all things that the

application developer must handle and can be a very time consuming, error-prone, and distracting activity.

The problem compounds when considering multiple OS platforms and their individual semantics. Characters that are illegal on Windows XP are not necessarily illegal on Unix.

Then there is the myriad of responses to these issues. Everything from guidelines recommended response dialogs, to automatic OS-handled behavior. Again, these are not handled automatically by the Java APIs. OS vendors expect their own native toolkits to be used, which provide some of these protections; but cross-platform developer must be mindful of these conditions on all platforms.

To resolve this deficiency, JDAF provides safe, OS-sensitive, cross-platform conscious application interaction with the underlying file system. The classes that facilitate this are *FileSystemMediator* and the *FileSystemResponseHandler*.

### FileSystemMediator

A FileHandlingFeature maintains a FileSystemMediator subclass called FileDialogValidator. FileDialogValidator is used during File open and save scenarios by virtue of acting as a DialogListener. It provides the following checks:

- Save Alerts
- File Read/Write Scenarios
- Illegal File Name
- Illegal File-System Characters
- File Type Integrity
- File Exists
- File Previously Opened
- Externally Modified

FileSystemMediator facilitates optional policies that can be set by the developer to define response behavior which could be application specific.

The method *setOpenPolicy()* accepts constants that define responses to situations where the file has been previously opened:

- OPEN_POLICY_BRING_TO_FRONT - FileDataModel will be focused (Default)
- OPEN_POLICY_IGNORE_IF_OPENED – Open request is ignored
- OPEN_POLICY_ASK_TO_REVERT – User will be asked to revert to the opening File

The method *setSavePolicy()* accepts constants that define responses for when the file has been previously saved under the given name:

- SAVE_POLICY_ASK_TO_REPLACE – Allows user to decide to replace file (Default)
- SAVE_POLICY_REPLACE – Replaces file without user consent

Normally the default settings for both of these are sufficient.

### FileSystemResponseHandler

For every condition detected by FileSystemMediator, there needs to be a response by the application. FileSystemResponseHandler is an interface which provides opportunity to respond to file exceptions and issues.

The FileHandlingFeature's FileDialogValidator implements this interface as well and so responds to file issues using guidelines recommended response dialogs. However, responses are also implemented by a FileSystemMediator in a ConsoleApplication, which provides the very same responses, but via console prompt interactions instead of dialogs.

One thing to note is that this behavior is automatic when using the FileHandlingFeature.

## Recent Documents

FileHandlingFeature provides an integrated "Recent Documents" facility for the File menu. This facility is implemented by the *RecentDocuments* object, which is a MenuGrouping subclass. It can be accessed from the ApplicationMenuBarsUI using the *RECENT_DOCUMENTS_MENU_ID* key. While this is a managed UI feature, there are some customizations that can be performed.

RecentDocuments can have an *item limit*, which can be set with *setRecentItemLimit()*. This is initially set to a default value based on the OS. To clear recent items, you would call *clearRecentItems().*

RecentDocuments menu items are represented by OpenFileActions in the application's ActionMap. The ActionMap will have up to *item limit*-number of OpenFileActions, even if the files themselves are not defined in all the Actions. Please do not add or remove these items directly in the ActionMap; use the item limit control in the RecentDocuments object to control the number of recent items shown in the menu.

The appearance of recent document menu items is OS-specific. A *RecentDocumentsFormatter* is used to format these items as they are introduced. The FileHandlingFeature provides a default RecentDocumentsFormatter implementation based on the OS requirements. This object is set via the *setRecentFormatter()* method.

# Integration with Other JIDE Products

Since the Application UI feature of GUIApplication manages the application UI automatically, we should briefly discuss how to use other JIDE products in your application.

Using *JIDE Components* and *JIDE Grids* should be straightforward, as they are used to build the DataView implementation for your application. We recommend starting with DataViewPane, which is based on JPanel and adds useful features.

## JIDE Docking Framework

*JIDE Docking Framework* will integrate automatically by simply calling the ApplicationUIManager method *setUseJideDockingFramework(true)*. If the docking jar is in the classpath, JDAF will install the Docking Framework in such a way that DataViews are installed into the Workspace portion of the DockingManager.

### Normal Docking

To use the Docking Framework traditionally as prescribed in its documentation, meaning without DataModels and DataViews, use a WindowCustomizer. The window is guaranteed to be a *DockableHolder.*

```
application.getApplicationUIManager().getWindowsUI().
  addWindowCustomizer(new WindowCustomizer() {
   public void customizeWindow(ApplicationWindowsUI ui, Container window) {
     DockableHolder dockableHolder = (DockableHolder) window;
     DockingManager dockingManager = dockableHolder.getDockingManager();
     if(holder.getDockingManager().getFrame("title") == null) {
       // configure dockable frames
     }
   }
   public void disposingWindow(ApplicationWindowsUI ui, Container window){}
});
```

By default, JDAF maintains a layout called "default" under the data directory of the application (See *getDataDirectory()* ). JDAF will load and save this layout automatically.

To manage your own layout, call the ApplicationUIManager pre-run option *setManageLayoutPersistance(false).* Then use a WindowCustomizer to load and save the layout as prescribed in the JIDE Docking Framework documentation. You may call *getLayoutPersistance()* in the ApplicationUIManager to return the appropriate LayoutPersistance object if you are, or plan to, mix the Docking Framework and Action Framework, as this returns the appropriate instance.

There is another alternative however to the traditional usage of the Docking Framework; MVC docking.

### MVC Docking

As an alternative to traditional Docking Framework usage, JDAF provides a means of tightly integrating the JIDE Docking Framework into JDAF MVC architecture via the *DockingApplicationFeature*. This feature requires the use of secondary DataModels and allows a DataView to be docked. The idea is to let you simply work with DataModels and DataViews and let the feature manage the docking.

For a DataModel's view to be routed to a DockableFrame, a DockableConfiguration must be defined and added using *addDockableMapping().* The DockableConfiguration defines how the docking framework will be utilized when a given DataModel and DataView are introduced. The following example creates two docks called “Frame1” and “Frame2”.

```
public class MyApplication extends GUIApplication {
  public MyApplication() {

   // feature
   DockingApplicationFeature feature = new DockingApplicationFeature();

   // a dockable mapping configuration
   DockableConfiguration config = new DockableConfiguration();
   config.setFrameName("frame1");
   config.setInitState(DockContext.STATE_FRAMEDOCKED);
   config.setInitSide(DockContext.DOCK_SIDE_WEST);
   config.setInitIndex(0);
   config.setDataModelClass(SecondaryBasicDataModel.class);
   config.setDataViewClass(DockedView.class);
   docking.addDockableMapping(config);

    // another configuration
    config = new DockableConfiguration();
    config.setFrameName("frame2");
    config.setInitState(DockContext.STATE_FRAMEDOCKED);
    config.setInitSide(DockContext.DOCK_SIDE_SOUTH);
    config.setInitIndex(0);
    config.setDataModelClass(SecondaryBasicDataModel.class);
    config.setDataViewClass(DockedView.class);
    docking.addDockableMapping(config);   ...
 }
```

```
    //…DataView implementations
  }
```

When a window opens, the default behavior of the DockingApplicationFeature is to install all DockableConfigurations by originating each configured DataModel and docking their DataViews. You may change whether openData() or newData() is used by calling *setOriginationMode()* to OPEN_DATA_ORIGINATION_MODE (default) or NEW_DATA_ORIGINATION_MODE.

If you desire to not initially originate DataModels when the window opens, you may call the DockableConfiguration method *setAutoInstall(false)*. Then call *toggleDockable(frameName)* with the name of the DockableFrame, or openData() or newData() manually. We provide the *ToggleFrameAction* to do such a thing for you, and it maintains a checked state as well. Alternately, you can leave *autoInstall* true and set the *initState* to DockContext.STATE_HIDDEN. Then when the DataModel is originated, the view is hidden. The toggleDockable() will then simply hide and show the DockableFrame without effecting the data lifecycle.

*Be mindful during testing that JDAF by default stores the docking layout when the window closes, so the initial states you define may be overwritten the next time you startup. That is, if the DockableFrame was visible at first, then you close the DockableFrame using the close box, then exit the application, the next run the DockableFrame will be hidden! Don't panic, you simply need to complete your code that manages the visibility of the DockableFrame and show it again. or delete the default.layout file in the data directory returned from getDataDirectory().*

Should you need to define criteria for your mapped DataModel, call *setCriteria()* on the DockableConfiguration. This criteria will be used to originate the DataModel. Be sure to pass this same criteria in, if you call openData(criteria) or newData(criteria) manually on the GUIApplication.

The DockingApplicationFeature will automatically call saveData() on any of its mapped DataModels on the dataModelClosing() event. By default this occurs when the window closes or with an explicit call to closeData(DataModel). Now, when a user clicks on a DockableFrame, it's normal behavior is to hide. But, you may wish for the behavior instead to close the DataView and DataModel, thereby forcing a save, and in effect cleanly tighting the data lifecycle to the DockableFrames accessibility. (Opening a DataModel opens the DockableFrame. Closing the DockableFrame closes the DataModel). To do this, call the *setCloseDataOnCloseFrame(true)* on the DockableConfiguration. Note that the ToggleFrameAction is knowledgeable of all this various behavior and will “re-originate and close” or simply “show and hide” the DockableFrame as appropriate to the individual DockableConfiguration.

In this final example, we use a DockableConfiguration so that the feature will not originate the DataModel on startup. The execution of the ToggleFrameAction will cause it to show and hide.

```
public class MyApplication extends GUIApplication {
    ...
```

```
    DockingApplicationFeature feature = new DockingApplicationFeature();
    DockableConfiguration config = new DockableConfiguration();
     config.setFrameName("Welcome");
     config.setAutoInstall(false);
     config.setInitState(DockContext.STATE_FRAMEDOCKED);
     config.setInitSide(DockContext.DOCK_SIDE_WEST);
     config.setInitIndex(0);
     config.setCriteria("Hello World!"); // criteria ref
     config.setDataModelClass(SecondaryBasicDataModel.class);
     config.setDataViewClass(MyDataView.class);

     feature.addDockableMapping(config);
       ...
     getActionMap().put("frame1", new ToggleFrameAction("Welcome"));

     // install in menu and toolbar
   }
```

Note in this example, if you needed to call openData(criteria) or newData(criteria) methods directly, you would use the criteria set in the DockableConfiguration ("Hello World!" in this case). If the criteria in the DockableConfiguration is null, you may use the frameName in the methods call instead.

The DataModel and DataView mappings can be managed outside the feature as well, solely relying on the criteria match to route the DataView to a DockableFrame. Internally, a *HashedDataModelFactory* and *SemanticNameDataViewFactory* are used by the feature.

DockableFrames can be customized by adding a *DockableFrameCustomizer*:

```
    feature.addDockableFrameCustomizer(new DockableFrameCustomizer() {
      public void customizeDockable(DockingApplicationFeature feature,
                                    DockingManager manager,
                                    DockableFrame dockable,
                                    DataView dataView) {
      }
    });
```

The *DockingManager* can be customized and the layout managed manually by adding a *DockingManagerCustomizer*:

```
    feature.addDockingManagerCustomizer(new DockingManagerCustomizer() {
        public void customizeDockingManager(DockingApplicationFeature feature,
```

```
                        DockingManager manager) {}
    public void loadLayoutData(DockingApplicationFeature feature,
                        DockableHolder window,
                        LayoutPersistence layoutPersistance) {}
            public void saveLayoutData(DockingApplicationFeature feature,
                        DockableHolder window,
                        LayoutPersistence layoutPersistance) {}
});
```

If no *DockingManagerCustomizer* is added, then JDAF will manage the a "default" layout in the data directory (using *getDataDirectory()*). Otherwise, if this feature detects a DockingManagerCustomizer, it turns off the default layout management so that your customizer can manage it. JDAF will provide the appropriate *LayoutPersistance* object for both the Docking Framework and/or the Action Framework as appropriate.

## JIDE Action Framework

*JIDE Action Framework* will integrate into your application automatically by simply calling the ApplicationUIManager method *setUseJideActionFramework(true)*. If the jar is in the classpath, JDAF will install and use a *DockableBarManager*.

The *DockableBarManager* can be accessed through the host window of a given DataView, which is guaranteed to be a *DockableBarHolder*.

We recommend setting up toolbars using a ToolBarCustomizer. The ToolBarCustomizer treats the toolbar itself as a Container. But code for adding buttons and other Components is identical whether you are using the Action Framework or not, as long as you use the ApplicationToolBarsUI factory methods *defaultToolBar()*, *defaultButton()*, *addToolBarButton()*and *addToolBarSeparator(),* the correct classes will be used (CommandBar or JToolBar, JButton or JideButton).

This is important particularly if the Action Framework is used conditionally for different OS/platforms as well as for OS Guidelines fidelity.

## DocumentPane

If you are using an application style that results in a Tabbed UI (TDI_APPLICATION_STYLE or MDI_APPLICATION_STYLE on Linux), the default Component for the tabs is a *JideTabbedPane*. However, if you have JIDE Components available, the *DocumentPane* can be used simply by setting the *setUseDocumentPane(true)* option on the ApplicationUIManager. You can access either tabbed component from application like so:

```
ApplicationUIManager appUI = application.getApplicationUIManager();
```

```
Container frontWindow = appUI.getWindowsUI().getFrontWindow();
JideTabbedPane tabbedPane = (JideTabbedPane)appUI.getTabbedUI().
                            getTabbedComponent(frontWindow);
```

If you need to use a custom DocumentPane implementation, you can install a *DocumentPaneFactory* in the ApplicationTabbedUI like so:

```
application.getApplicationUIManager().getTabbedUI().setDocumentPaneFactory(
    new DocumentPaneFactory() {
      public Object createDocumentComponent(JComponent content,
                    String name, String title, Icon icon) {
        ...
      }
      public Component createDocumentPane() {
        ...
      }
});
```

## JIDE Stock Icons

While JDAF provides standard icons for those OS-platforms (XP, Vista/ Windows 7, Mac OS X) that support icons by default, JIDE Stock Icon products contain beautiful sets of icons to provide for the rest of your application functionality. Also, if you wish to have toolbar icons on Mac OS X, the Mac OS X icon set is an easy choice.

JDAF provides a no hastle way to integrate these icons using the *IconSetIconTheme*. This IconTheme will detect the OS-platform you are on and any stock icon library you have in the classpath, and will install those icons based on the running OS-platform. Using is simple:

```
GUIApplication application = new GUIApplication("My App");
IconSetIconTheme iconTheme = new IconSetIconTheme();
iconTheme.install(application);
application.run();
```

IconSetIconTheme interacts with an internal JIDE Commons IconSetManager by mapping JDAF ActionKeys to IconSet icon keys. Therefore, all the standard GUIApplicationActions provided by JDAF are matched to the IconSet automatically. If you wish to use the other icons in the IconSet with a custom action you have added to the GUIApplication ActionMap, you simply add an actonKey-to-IconSet mapping like so:

```
iconTheme.addActonIconMapping("export", IconSet.File.EXPORT);
```

Where "export" is the actionKey. You can also access icons using the IconTheme methods *getSmallIcon()* and *getLargeIcon()* or by accessing the IconSetManager directly using *getIconSetManager()*, to gain access to the other icons in the IconSet for other purposes.

# Making a Console Application

Console-based applications remain a popular development choice in the Linux and UNIX worlds. In fact, many GUI programs are actually front-ends to command-line programs. Their continued popularity remains because they are still the quickest way to build a utility or tool, test a model, or make a prototype when time or other motivators will not permit a full GUI.

So what happens when you apply MVC architecture to console application development? Order and Ease! Gone is the giant switch or if-else statement blocks that control input and program flow, replaced by encapsulated and reusable ConsoleCommands. Gone are all the string parsing and data type conversion chores, replaced by a ConsoleView that provides type-blocked reads and formatted writes! Additionally, by virtue of the framework, you also get full data support, file handling, a help system, and printing capability!

The Console Application API is not a trivial library. It is every bit as powerful as GUIApplication, just without the GUI. So whether you're making a custom shell emulator, a console game, utility, tool, test harness, prototype, or a full-blown application, the Console Application API will greatly increase productivity and help deliver a stable, maintainable, and robust application.

## ConsoleApplication

ConsoleApplication is a subclass of DesktopApplication. Create a ConsoleApplication like so:

```
ConsoleApplication application = new ConsoleApplication ("My Application", "menu");
// configure application (built-in commands are available)
application.run(args);
```

The first argument in this constructor is the application name, which shows in an introduction prompt when the application runs. The second argument is the "start command". A ConsoleApplication is "command-driven". The user types a command into the Java console and the program responds by locating and executing a ConsoleCommand. This repeats until the program is terminated.

ConsoleApplication is preconfigured with a dozen+ commands that manage data and file handling, help, printing, and application termination. It also has an "idle" command. The idle command is a default behind-the-scenes command that is used when an explicit command is undefined. It also serves as a fail-safe command to ensure a ConsoleApplication can still run in the event a command is not recognized.

A ConsoleApplication supports only one type of view; a *ConsoleView*. ConsoleView is a virtual object behind the actual command prompt. ConsoleView is managed automatically; there is no responsibility on the developer for the view portion of the application. The primary focus of console application development is deciding how data will be managed, and developing the commands necessary to perform the functionality of the application.

Because the view is always present, a respective DataModel must also be present. Therefore, ConsoleApplication is preconfigured with a FileDataModelFactory and a BasicDataModelFactory. In this way the ConsoleApplication can create data by default, either from files off the command line, or by default with a BasicDataModel.

## ConsoleApplication Constructors

ConsoleApplication has many constructors. The constructor you choose largely has to do with your application requirements and how to achieve a suitable data infrastructure. Below is a guide we hope will aid in making this initial decision:

| Constructor Arguments | Comments |
|---|---|
| (String name) | This is a minimal constructor. The application name is always required.<br>ConsoleCommands and other customizations should be added before *run()* is called. The idle command will be the start command. |
| (String name, String startCommand) | This is the most common constructor. It allows the definition of the start command.<br>The start ConsoleCommand and other commands should be added before *run()* is called. |
| (String name, String startCommand , DataModelFactory dataModelFactory) | This constructor is useful to explicitly define a DataModelFactory for the ConsoleApplication. In this case, the input factory is placed before the BasicDataModelFactory in precedence.<br>It is similar to the previous, in that the start command can be specified.<br>The primary benefit is a custom DataModelFactory provides more control over the initiation of the DataModel. Hence, no DataModelListener is needed. |
| (String name, String startCommand , DataModelFactory dataModelFactory, ConsoleViewFactory consoleFactory) | This constructor is similar to the previous, but it allows for a subclass of the ConsoleView-Factory to be provided, so that a ConsoleView subclass could be introduced, if needed. |

| Constructor Arguments | Comments |
|---|---|
| (String name, String startCommand , Class dataModelClass) | This constructor is useful in order to define the type of DataModel to create automatically without the definition of DataModelFactory or DataModelListener. |
| (String name, Class dataModelClass) | This constructor is similar to the previous; you just don't need to define the start command. |

### Managing Data

A ConsoleApplication is preconfigured with both a FileDataModelFactory and a BasicDataModelFactory. A TextFileFormat is already registered with the FileDataModelFactory so text can be read by default. Add or remove FileFormats as necessary using the FileDataModelFactory directly. It is up to the developer as to whether this factory setup is sufficient to handle the data requirements.

Whatever the factory setup, a ConsoleApplication must create a DataModel on *run()*, no matter what the setting of *isOpenDataOnNew()* is. This is because a ConsoleApplication always has a view; the ConsoleView, and there must be a corresponding DataModel.

The developer, then, may wish to control what the initial data should be. Without modification the following scenario takes place:

1) If files are found on the CommandLine, they will be loaded, and the first FileDataModel will be set as the focused DataModel. Text files are supported by default. The developer may want to add FileFormats as appropriate to the FileDataModelFactory.

2) If no files are found on the CommandLine, or the files are unsuccessful in loading, a BasicDataModel will be created, unless a DataModelFactory or DataModel class where provided in a constructor.

DataModelFactory objects can be added and removed as necessary, but when *run ()* is called, there must be at least one DataModelFactory that can successfully produce a DataModel, or an Exception will be thrown.

If there are no data requirements, then nothing needs to be done. The DataModel is passive unless a command is directed to use it. Of coarse you can define your own DataModel and prepare it as appropriate in the *newData()* method.

## CommandMap

A ConsoleApplication is "command-driven". The Console is always in the state of executing a command while it is running. Commands are facilitated by a special purpose ApplicationAction called ConsoleCommand.

ConsoleCommands are used to drive user interaction in the console and perform functionality. ConsoleCommands are maintained in the application ActionMap. However, ConsoleApplication is only compatible with ConsoleCommands. It ignores other Action types.

A ConsoleApplication has a CommandMap. CommandMap manages keyword to Action relationships. Command line keywords are mapped to keys in the application's ActionMap. When a user submits a command from the console, ConsoleView will use it to lookup the ConsoleCommand Action to execute. Hence, ConsoleCommands are only "exposed" to the console if they have a mapping in the CommandMap.

To add a ConsoleCommand to the ConsoleApplication you use one of these signatures:

```
public void putCommand(String, String, ConsoleCommand);
public void putCommand(String, ConsoleCommand);
```

Both methods will add the ConsoleCommand to the ActionMap. The first method allows the command keyword, action key, and console command to be provided in one fell swoop. The second method will use the same value for the CommandMap and action map when adding the ConsoleCommand.

## ConsoleCommands

A ConsoleApplication is command-driven. It must always have a command to execute. This keeps the application running. The developer can set the "start" command from the ConsoleApplication constructor. ConsoleApplication always maintains the notion of a "next" command. When a ConsoleCommand is finished executing, the application will use the next command to locate the next ConsoleCommand to execute. ConsoleApplication tracks the "previous" command as well.

### Built-In Commands

ConsoleApplication is pre-configured with built-in commands. The commands below are available upon construction:

| Command | Class | Comments |
|---|---|---|
| exit | ExitCommand | Exits the ConsoleApplication |
| help | HelpCommand | Executes ConsoleHelpService |
| dir, ls | DirectoryCommand | Lists the files an directories in the current |

| Command | Class | Comments |
|---|---|---|
| | | directory |
| cd | ChangeDirectoryCommand | Views and sets the current directory for file handling |
| open | OpenCommand | Loads a file defined by the argument |
| close | CloseCommand | Disposes and removes the named DataModel from the application |
| new | NewCommand | Creates new data |
| save | SaveCommand | Stores the data to a file. |
| print | PrintCommand | Prints the focused DataModel |
| display | DisplayCommand | Lists data models resident in the application and their data |
| select | SelectCommand | Sets the named DataModel as the focused model |

These commands can be easily removed with a call to *uninstallDefaultCommands()*. The intent of these built-in commands is to facilitate basic operations such as data management, help, printing and termination. Try running a ConsoleApplication and experimenting with these commands. Type “help” to view information on command usage.

Any built-in commands can be removed if the functionality is not desired. Or they can be remapped to a command more appropriate to the application. Use *removeCommandMapping()* to remove just the command mapping. The actual ConsoleCommand remains in the ActionMap and can be remapped using add*CommandMapping()*. The *removeCommand()* method will remove both the command mapping and the corresponding ConsoleCommand. We don’t recommend removing the exit command.

Please, do not remove the idle command. It is used internally and not mapped. If it is removed from the ActionMap, unpredictable behavior may result.

It should be noted that the IdleCommand can be replaced with an alternative ConsoleCommand. This is a useful technique; set a “menu” command as the idle command. That way, unless a command sets the *next* command, the application always returns to idle command, which is your menu command. Easier still, you can set a prompt into the existing IdleCommand to serve as a default menu prompt. By default there is an empty prompt.

Finally, the *DelegatingComand* is a convenience class that delegates its *actionPerformed()* method to another class, such as the ConsoleApplication or some other target. This reduces the number of ConsoleCommand classes that need to be defined.

### Creating ConsoleCommands

As an ApplicationAction, ConsoleCommand is very straightforward to implement; simply subclass and implement the *actionPerformed()* method. Alternatively, you may implement a synonym for this method with the same arguments signature as *actionPerformed()* in another class and use the DelegatingCommand to call it. Either way, following is a discussion of how ConsoleCommands are used in a ConsoleApplication.

ConsoleView displays the text from the ConsoleCommand via getPrompt() when the command is executed. The value is actually stored in the *Action.NAME* property of the ConsoleCommand but is accessed with the *getPrompt()* and *setPrompt()* methods. Normally, the cursor is placed at the end of the prompt. To make a new line occur after the prompt, use *setUseNewLine(true)*.

When the user responds to the prompt by hitting enter in their console, the *actionPerformed()* method of that ConsoleCommand is called. In this way each ConsoleCommand represents a single "dialog" with the user.

A ConsoleCommand can have a null prompt, in which case no output is sent, and the *actionPerformed()* method is called immediately. This is useful for "function-only" commands.

Hence, the *actionPerformed()* method contains the application logic of the command. It should generally do the following:

*1)* Use one of the *read()* methods from ConsoleView to get the user's input. The ConsoleView can be accessed directly in the ConsoleCommand via *getConsole().*

*2)* Perform some action, such as operations on the focused DataModel or application, or write some output to the ConsoleView. The focused DataModel is available using *getFocusedDataModel().* The ConsoleApplication object is available via *getApplication().*

3) Finally, set the next command to execute. This is done by calling the *setNextCommand()* method. The command is a String matching a command registered in the CommandMap.

Setting the next command is important! If the next command is not set ConsoleView will revert to the IdleCommand. This could be convenient if the user's input was not acceptable, but in general the developer should set the next command to control the application flow. If you desire for the ConsoleCommand to execute repeatedly, then set the *setRepeatable(true)*. This is not recommended as it is easy to get into an infinite loop. To break the loop set the next command other than the current one.

To set the next command to the previous command, use the ConsoleCommand method *resetCommand ().* This resets the command to the command before the current command was executed. This is a useful reusability technique. It allows ConsoleCommands to be created or used which perform a task, and then return to the caller, anonymously. Keep in mind; if the next command is not recognizable, the IdleCommand is used.

### Command Arguments

A ConsoleCommand can accept arguments. These are simply captured from the ConsoleView input when the ConsoleCommand is called. The ActionEvent passed to the *actionPerformed()* method will contain the entire console command that was used, in its ACTION_COMMAND_KEY property. The *getArguments*(*ActionEvent*) method can be used to extract these arguments as a CommandString. CommandString provides a robust means by which to extract argument values.

## File Handling

ConsoleApplication provides means to open, close, and save file data. ConsoleApplication remembers the last directory used by the user when managing files with the file commands. The ChangeDirectoryCommand and DirectoryCommand simulate popular shell syntax for browsing directories on the file system, so that file data can be saved and loaded conveniently.

ConsoleApplication has an integrated FileSystemMediator object which provides the robust file validation capabilities. Each ConsoleView is a FileSystemResponseHandler, so the ConsoleView can automatically respond to file system validation issues and negotiate textual dialogs with the user as required.

See the sub-section on *File System Mediation* in the “Making a Document-Centric Application” section for more information on this feature.

## ConsoleViewFactory

A ConsoleApplication is preconfigured with a ConsoleViewFactory, which vends a ConsoleView for each DataModel, no matter what the type. Only a ConsoleViewFactory should be used in the ConsoleApplication, but the developer may subclass the ConsoleViewFactory in order to deliver a subclass of the ConsoleView. The constructors of ConsoleApplication make this straightforward by accepting a ConsoleViewFactory as input.

While providing a ConsoleView alternative is not likely to be necessary, one possible scenario might be to redirect the ConsoleView I/O. A ConsoleView constructor allows for these streams to be defined. By default, the ConsoleView I/O are *System.in* and *System.out*.

## Using ConsoleView

ConsoleView is a virtual interface to the OS native console (Java console). Beyond being an input and output device for the application, it is also the command dispatcher. Command dispatching is automatic, however. Normally, the developer need only be concerned with reading and writing with the ConsoleView.

### Reading from the Console

ConsoleView makes getting input from the user very simple. It features type-blocked read methods, uses a powerful string parser called CommandString, and uses the JIDE *ObjectConverter* API for pluggable control of string to object conversion.

The following read methods are available:

- readString()
- readInt()
- readChar()
- readLetter()
- readLetterUpper()
- readLetterLower()
- readLong()
- readDouble()
- readObject(Class)
- readInput()
- readResponse()

These methods block until the appropriate data is input by the user. Meaning, the read method will not return, and the last prompt will repeat until the requested type of data is input properly.

The *int*, *char*, *long*, *double, and letter* read methods are self-explanatory. They return the respective primitive types.

The *readObject(Class)* method blocks until an Object of the given Class type can be retuned. The *readObject(Class)* method relies on ObjectConverters to convert a String to an Object. If the *ObjectConverterManager.initDefaultConverter()* has been called, all of the default JIDE ObjectConverters will be available. Otherwise, the developer should register the desired ObjectConverter with the ObjectConverterManager. ConsoleView is a *ConverterContextSupport* object, so if necessary, set the ConsoleView with your ConverterContext. The ConsoleView's ConverterContext is null by default.

Both *readString()* and *readInput()* will return if there is at least one character on the input stream. The difference between them is that one returns the raw String value, the other parses the String and returns it as a CommandString object. CommandString provides a robust means of validating and parsing the input, which could contain user arguments.

### Console Dialogs

The *readResponse()* method provides for controlling program flow via user input. This method implements the console analogy to a GUI option dialog. It provides a question with a defined set of response codes. This method is very convenient for branching in a ConsoleCommand. For example, during a ConsoleCommand's actionPerformed method, this method can be called when the next command is based on a user's confirmation.

There are two signatures for these methods:

The *readResponse(String, int)* method provides for a question and option type. Option type is a constant defined by JDAFConstants class; OK_CANCEL_DIALOG, YES_NO_DIALOG, or YES_NO_CANCEL_DIALOG. The return is an int corresponding to one of JDAFConstants response codes RESPONSE_YES, RESPONSE_NO, or RESPONSE_CANCEL.

As an example, consider the following code:

```
public void actionPerformed(ActionEvent e) {
  // evaluate some answer
int response = getConsole().readResponse("Do you want to continue?",
                              DialogConstants.YES_NO_OPTION);
  if(response == DialogConstants.RESPONSE_YES) {
       setNextCommand("question");
  }
  else {
       setNextCommand("menu");
  }
}
```

In this example, the output will read:

```
Do you want to continue? (Yes/No)
```

The *readResponse()* method blocks until something interpretable[11] as a "Yes" or "No" is input. The response is translated to a response code which can be used to make branching decisions.

[11] The Console method *translatePromptResponse()* is used to interpret the String response to a question. The values that are accepted are lenient and case insensitive. For example if (Yes/No) is the hint, RESPONSE_YES will be returned if the value is y, Y, Yes, YES, yes, OK, ok.. These values are localized and can be replaced and internationalized by changing the resource files.

The *readResponse(String, Object, int)* method is similar to the previous method, but provides for a parameterized question using an internal Format object. If an Array is input, MessageFormat is used, otherwise ObjectFormat is used.

### Writing to the Console

All output to the Java console must be directed through the ConsoleView object for the application to work properly. While writing to the Java console directly is trivial, a ConsoleView provides many features that make this requirement not only convenient, but decidedly more beneficial.

Much output is automated, because ConsoleView, by design, writes the NAME property value of the executing ConsoleCommand as the command prompt. Output, however, may be written at any time, usually during a ConsoleCommand's actionPerformed method. ConsoleView keeps track of all output with close attention to line breaks and such, so that that display integrity is maintained with all I/O traffic. For example, the *isNewLine()* method can be called to determine if the next output character will be at the beginning of a new line.

The following write methods are available:

- write(Object)
- writeLine(Object)
- write(String, Object)
- writeLine(String, Object)
- writeTable(TableModel, Map);

The ConsoleView *write* methods come in two flavors, *write()*, which appends to the current line, and *writeLine()*, which appends to the current line and adds a new line afterwards.

The *write(Object)* and *writeLine(Object)* methods take a String or any other Object. If the argument is A String.class, it is written verbatim. Other types use ObjectConverters to perform Object-to-String conversion. If an ObjectConverter cannot be found for the Object's Class, the Object's *toString()* method is called.

The *write(String, Object)* and *writeLine(String, Object)* methods provide for parameterized output using a Format class. If the Object is an array, it is passed through the ObjectConverters. This behavior can be switched off using *setUseConvertersOnOutput(false)*. MessageFormat is then used to construct the final output. Otherwise, ObjectFormat is used.

The *writeTable(TableModel, Map)* method will write a TableModel's data to the console. Data values will pass through the ObjectConverts as well. This behavior can be switched off using

*setUseConvertersOnOutput(false)*. Strings are output verbatim in any case. Table layout can be controlled by passing in a Map of properties. See javadoc for details.

### ConsoleFilter

A ConsoleView can have one or more *ConsoleFilter* objects. A ConsoleFilter is used to filter input and output to and from a ConsoleView. If more than one ConsoleFilter is added, the respective input and output will be passed through each filter in the order the filters where added. ConsoleFilter is an interface with the following methods:

```
public String filterInput(String input, String command);
public String filterOutput(String output, String command, boolean newLine);
```

A ConsoleFilter can have many uses, for example, it could be used to extract useful information from the output generated by an ExternalCommand process. Or, all output could be formatted with a special prefix, such as a terminal emulator “>” character. The uses are up to the developer, but this facility provides an easy way to work at a low level with the ConsoleView I/O without subclassing ConsoleView.

## ConsoleViewListener

For every DataModel created, a ConsoleView is also created. While physically there is a single Java console, internally there is a virtual ConsoleView for each unique DataModel. When a DataModel is focused, its respective ConsoleView will be activated and all I/O will go through that ConsoleView instance. While normally this is an uninteresting implementation detail, there are times when knowing the whereabouts of a particular ConsoleView instance is desirable; for example to set properties on a ConsoleView, or some other task.

To this end the developer can register a ConsoleViewListener with the ConsoleApplication using *addConsoleViewListener()*. A ConsoleViewListener can receive the following events:

- consoleActivated()
- consoleDeactivated()

## CommandString

CommandString is a powerful utility class used by the Console Application API. It provides pattern validation and advanced type parsing capabilities, providing a greatly simplified means of working with command line Strings.

Creating a CommandString is simple and inexpensive:

```
CommandString commandString = new CommandString(commands);
```

CommandString separates a String into command elements using white space. To include white space in a command element, double-quotes are required to define the element bounds. CommandString is mutable. It can be reused by calling *parseCommands(String).*

CommandString can reconstitute the original string or a portion of the command string using *toString()* or *substring()*. It can also return the parsed command array in full or by range, using *getCommands()* and *getCommands(int, int)*. The later is convenient for making another sub CommandString.

### Command Validation

Once a CommandString is created, the number of command elements can be counted using *length().* CommandString provides a very simple way to validate the type potentiality of elements by using Class patterns. The *validate()* methods accept Classes and positional information and one uses RegEx. There are four signatures:

- validate(Class, int)
- validate(Object[])
- validate(Object[], int, int)
- validate(String, int)

The first method returns whether the nth command element can be parsed to the given type. The second method compares each element in the input list to each element in the internal command element list. If the input element is a Class, the conversion potential of the corresponding element is checked. If the input is a String, *equals()* and *match()* are used to compare the corresponding command element. The third method provides the same functionality but by a range of command elements. The last method uses a regular expression to explicitly test the nth command element.

Single and mixed switches can also be detected using *isSwitch()*. A single command can be validated position ally using *indexOf(command)*.

### Type Conversion

CommandString uses the ObjectConverter API to perform conversion and validation. The developer should provide ObjectConverters for any types needing conversion. CommandString is a ConverterContextSupport object as well, and can be set with a ConverterContext to target a particular set of ObjectConverters.

CommandString provides many primitive typed access methods such as *getInt(int)*, *getLong(int)*, *getDouble(int)*, and a special *getList(int, String)* which provides for a separator to extract a single element

as an array. The *getObject(int, Class)* method will attempt to parse the nth element to the given type using an ObjectConverter.

## Console Help

ConsoleApplication already comes with a system that makes integrating new help almost automatic. ConsoleApplication contains a HelpSource implementation called *ConsoleHelp*. ConsoleHelp runs through every command in the command map and prints a neatly justified table of each command and the description from the ConsoleCommand. Hence, implementing a common help system found in shell and other console applications.

For each custom ConsoleCommand the developer can provide a help String using *setHelp()*. This String should be short and to the point, including any information concerning arguments to the command. The help String is stored in the Action.SHORT_DESCRIPTION property of the Action. The ConsoleCommand methods *setHelp()* and *getHelp()* use this property internally.

HelpCommand is built-in to ConsoleApplication. It is mapped to the command “help”. Because there are quite a few built-in commands, seeing help in action is simple; create and run a ConsoleApplication, then type “help” to see a listing of all the loaded commands.

Of course there is nothing stopping the developer from providing a custom HelpSource. The HelpCommand can still be used to invoke it.

## Console Printing

Printing in a ConsoleApplication is identical to printing in a GUIApplication. But, instead of invoking the Printer from the File Menu’s Actions, printing is invoked using the PrintCommand, which is mapped under the command “print”.

ConsoleView already implements PageableSource. In order to provide printing, set a Pageable object in the ConsoleView.

## ConsoleApplication - About

Oddly, the last thing we mention is the first thing that happens when you run a ConsoleApplication. When a ConsoleApplication runs, it internally fires an AboutCommand. The AboutCommand is in the ActionMap but not mapped as a command. It provides a simple prompt that introduces the program. The prompt is composed by using the application name and the version String.

To change this about introduction prompt, simply access the AboutCommand from the ActionMap using AboutCommand.KEY, and set the prompt to whatever the desired introduction should be.

## ObjectFormat

You may encounter in various places in the framework an object called *ObjectFormat*. While ObjectFormat is not specifically desktop application related, like ObjectConverters have general utility, we rely on ObjectFormat quite often, especially when formatting resource Strings. Once you understand ObjectFormat, you may wish to use it as well.

ObjectFormat is like a mini-template engine. It provides a means of producing a parsed information from an Object in language-neutral way using a String pattern. This object can be used to construct Strings to display to end users based on source information. ObjectFormat takes a *source object* and a String pattern, extracts and formats information from the source object, then inserts the information into the pattern at the appropriate places.

ObjectFormat is not a *java.text.Format* subclass, as it does not provide the contract to turn a String back into an Object. It is also not concerned with locale-sensitive issues like *NumberFormat*. However, it is used in a similar manner to MessageFormat. In fact it can replace MessageFormat, as it leverages much more readable pattern strings, which is more conducive to translation activities. It also makes formatting in your application more readable and maintainable.

ObjectFormat can be used as an instance:

```
String pattern = // String pattern
Object sourceObject = // object
String result = new ObjectFormat(pattern).format(sourceObject);
```

Or statically:

```
String result = ObjectFormat.format(pattern, sourceObject);
```

To extract information from a source object, ObjectFormat uses a very simple expression language using UL syntax (Universal Expression Language) in the form:

```
${property}
```

The property is interpreted as a bean method against the source object. For example, if the source object is *Car*, having a method *getColor()*, a pattern with the token `${color}` would cause the value of *Car.getColor()* to replace occurrences of the token. Nested method execution can be accomplished by dot-separating property names. For example, the token `${color.blue}` would substitute the token with the value resulting from *Car.getColor().getBlue()*. If a source object is a java.util.Map implementation, the property is used as the key into the *Map.get()* call, otherwise, the nested properties are used to make method calls to each object in the call chain. In fact, any accessible zero-argument method can be used in the token call chain that produces a value.

## FORMATTING MESSAGES

The EL-syntax can be extended to include a format pattern that is compatible with MessageFormat. If the property resolves to an Array or Collection, the values are used as input to the MessageFormat pattern. Otherwise, the value is presented singularly. The MessageFormat pattern is placed after the property, separated with a comma. For example:

```
${color, The car is {0}}
```

At first this looks trivial. After all, this could be accomplished more readably using ObjectFormat syntax like so:

```
The car is ${color}
```

However, keep in mind that all the value formatting power of MessageFormat is available, including the use of ChoiceFormat, since ChoiceFormat features are interpretable in MessageFormat patterns. Consider that the Car class has a *getMileage()* method. Now consider the following ObjectFormat pattern:

```
The mileage is ${mileage, {0,choice,0#low|90000.0#high}}
```

This would yield a String "The mileage is low" or "The mileage is high" depending on the value of *getMileage()*

If the property is a boolean and a MessageFormat pattern is provided using a ChoiceFormat sub-pattern with the ranges 0 to 1, the boolean result will be converted to false=0 or true=1. This allows for simple in-line if/else logic to be embedded in the resulting String. Let our Car have a boolean method *isAvailable()*.

```
The car ${available, {0,choice,0#is|1#is not}} available.
```

This pattern could yield the String; "The car is available" or "the car is not available".

Finally, the values inside the MessageFormat pattern can also contain UL-syntax to further parameterize the MessageFormat pattern String.

**Note:** If a token in the pattern expression cannot be interpreted, due to syntax error or any other exceptions, the token will remain in the String pattern.

### Implementing Custom Pattern Syntax

EL-syntax is not the only way ObjectFormat can parse information. Let us describe how ObjectFormat resolves a pattern. To resolve a pattern, ObjectFormat uses one or more *ObjectFormatToken* objects. (In fact, the UL-syntax is facilitated by instances of the inner *ELFormatToken* classes.)

Each ObjectFormatToken is responsible for extracting information from a source Object, based on the presence of its representative token in the pattern. Thus, user-defined tokens in a given pattern use a corresponding ObjectFormatToken class to interpret their value.

To provide a custom token interpretation, the developer subclasses ObjectFormatToken and instantiates it with the representative token String to be replaced in a pattern. Then the *getSubstitution(Object source)* method should be implemented to return the replacement value. This expandability allows both EL-interpretation overrides and a mix of token interpretation styles, even the implementation of a completely custom token syntax.

To facilitate custom pattern syntax, ObjectFormat is used like so:

```
String pattern = // String pattern
Object sourceObject = // object
ObjectFormatToken[] tokens = // define token interpreters
String result = new ObjectFormat(pattern, tokens).format(sourceObject);
```

Or statically:

```
String result = ObjectFormat.format(pattern, tokens, sourceObject);
```

## Migrating To JIDE Desktop Application Framework

Whether one has a finished desktop application, is planning to start a new application, or is in the middle of development, there are many reasons that may justify migrating to JDAF:

1. To obtain the benefits of the data and file handling.
2. To tame an “at risk” architecture and produce a more maintainable application
3. To gain the benefits of the managed UI harness and its OS integration features
4. To offset existing developer liability by placing a large portion of the application architecture onto a supported product

There are other possible motivations, but whatever the reason, we want to make your migration as easy as we can.

Desktop Application migration is a wieldy topic because the possible architectures to be migrated are numerous[12]. We will attempt to outline what would need to be done to migrate by assuming some common designs that many applications end up implementing. In any event, please read this document thoroughly before attempting a migration.

[12] We are not speaking of console applications here, but primarily GUI applications. Likely migrating a console application is not a predominate concern. If we’re wrong, let us know!

## AWT and SWT Applications

During the design of this framework, we had to make certain decisions about GUI support; supporting AWT, Swing and SWT where concerns. While we made some architectural decisions with all toolkits in mind, ultimately it seemed that the framework would be too abstract to be useable. While possible to support such frameworks in the future, we decided to use the Swing toolkit due to the advances in Java 5 and 6 and its default availability in J2SE.

Therefore, if your application is in straight AWT or SWT, there may be considerable rework of your GUI. You will have to decide whether a migration is desirable or feasible for your application. If you intentionally wish to move to Swing, this is a wonderful way to do it, because the framework minimizes the Swing code needed to create your application.

The migration procedure from this perspective would be to follow the steps in "Making a GUI Application" and/or "Making a Document-Centric Application". The following tips may be useful as a general roadmap:

- Cull out a data model from your application and find a suitable DataModel to host this data in the application. If your application is file-based, you should identify or create a FileFormat as opposed to a DataModel, and use the FileBasedApplication class.
- Re-implement the content of your application windows into DataViewPane subclasses. May we suggest the use of our other JIDE products to aid in this task, such as JIDE Components, JIDE Grids, and JIDE Dialogs? These can greatly improve development productivity and presentation.
- Re-implement any menus and toolbar logic by creating and adding Actions to the GUIApplication ActionMap. You may use GUIApplicationActions if their functionality will benefit you.
- Add MenuBarCustomizers and ToolBarCustomizers and use these Actions to populate menus and toolbars.

## Swing Applications

Swing-based applications tend to lead developers towards GUI-centric design. Often the application *main()* method is in a JFrame subclass, which seems to be the center of the application. In this scenario most application methods sit in the JFrame subclass, perhaps called by Actions, or implemented in Actions. The application may or may not be using the JFrame as a main window or using JInternalFrames for content.

A more robust architecture may identify the "application" as the central class, thereby managing the application methods and GUI separately. An even more robust application may use a "loader" class as the main entry point; so that the application class-loading is sandboxed and corporate policies can be

enforced. In either of these designs, Swing class-loading is preempted to provide a snappier splash screen or to control other class loading issues such as secure jar loading, updating, license control, etc.

In all cases we can be sure menus and toolbars are constructed using JMenuBar and JToolBar, normally populated with Actions. The application may or may not support a Splash Screen, Printing, Help, About Windows, Preferences with Preference Window, data persistance, security, and other various desktop application needs.

We will attempt to address application migration covering each area.

## Application Entry Point

GUIApplication is intended to facilitate the entire desktop application lifecycle. The GUIApplication will represent your application in some manner. While it is certainly acceptable or necessary to create your own DataModels, if you are working with File data, try using a FileFormat and adding the FileHandlingFeature to the GUIApplication. This facilitates OS-sensitive and robust file support. Note that this feature has a dramatic effect on the standard File menu. If your application is purely file-based, use the FileBasedApplication class as it is preloaded with a FileHandlingFeature and has some usability methods.

Whether you use your own class as your *main()* class, and use a GUIApplication instance internally, or you wish to subclass a GUIApplication and put *main()* there, is your preference.

If your application is designed with an “application class”, subclassing one of our GUIApplication classes may be a conceptually easier transition. Move all application methods and fields from your class to the new GUIApplication subclass. Ultimately you will expire GUI methods for the functionality provided in GUIApplication.

If your application extends off JFrame, you can do either; make a GUIApplication subclass or make a general class and use an instance of GUIApplication. Either way, move all non-*GUI* application methods from your JFrame class to the new class. Which ever way it is done, our GUIApplication class will be the focus of the application.

It is important that the GUIApplication instance be instantiated as immediately inside the main method as possible. Particularly before any Swing resources are engaged.

GUIApplication ensures that the UI is started on the Swing event thread.

## Migrating Data

If your application acknowledges data as a single Java object but does not require persistence, then wrapping your data in a BasicDataModel implementation is straightforward. Use a BasicDataModelFactory and use a DataModelListener to manually introduce your data using *setData()* during the DATAMODEL_INITIALIZED event. GUIApplication should be used in this case.

If your application uses files, then we recommend using FileBasedApplication. If you need to mix models (view both files and data from other sources), you may use a GUIApplication and install the FileHandlingFeature object. In either case, provide the appropriate FileFormat objects for your data. If we do not supply an appropriate FileFormat class, subclass FileFormat and implement the *readData()* and *writeData()* methods, and provide the appropriate file extension.

If your data is managed from a database[13], we currently recommend subclassing AbstractDataModel or BasicDataModel and setting up a data access layer in the init() method. The data lifecycle methods can be implemented such as *newData()*, *openData()*, *saveData()*, resetData(), *closeData()*, etc, to interface with the database. The NewAction criteria or another GUIApplicationAction could be set with a query string or a connection string so that a ResultSet could be loaded, creating data for the application. A DataModelFactory should be used to vend the new DataModel in response to a query string or some other data source, such as a Connection object. The data property could house a ResultSet, data from a ResultSet, or a Connection object that is queried each time the DataView requests it. There is plenty of flexibility and it can depend on how your UI is structured.

Just keep in mind, your UI will have access to the DataModel, which has a data property. But you may use DataModel however it is most convenient.

## Setting Look and Feel

JDAF Managed UI sets the Swing Look and Feel automatically. Unless there is a strong motivation, such as corporate branding, we recommend respecting the native look and feel of the platform for the best integration with the OS. If we don't currently have a specific Application UI for a particular OS, the Java Cross-Platform Application UI is used, which defaults to the "Metal Look and Feel". If this is not acceptable, by all means replace the look and feel.

You will need to turn off the setting in the ApplicationUIManager by calling *setSetsLookAndFeel(false)*, or it will override your setting when *run()* is called.

[13] We plan to provide a legitimate JDBC-based DataModel implementation in the future.

## Migrating Window Code

GUIApplication manages all document-windows automatically. Hence, your window's content will need to be migrated to a DataView, likely a subclass of DataViewPane. Your code is probably in the content pane of a JFrame or JInternalFrame. Basically, you should be able to copy the code that adds Components to the content pane, and paste it in the *initializeComponents()* method of a DataViewPane subclass.

When the DataModel is introduced, your DataView class will be created and presented. You can use the *updateDataView()* method to set the state of your Components to reflect the DataModel, and use *updateDataModel()* to update the DataModel to reflect the values of the GUI. The only challenge here is if the code used to add your components is the same code used to populate the components. If you separate this behavior you may find that it generally makes the application more maintainable. By the same token, there may be circumstances where this is not possible. Please feel free to put all the code in the *updateDataView()* method, but just know that this method will be called each time the view needs to be refreshed.

If there is any code that manipulated the window consider these possible solutions:

- If you have an application icon set in the frame, please supply this icon to the ApplicationUIManager object using its *setSmallIcon()* or *setLargeIcon()* methods. The ApplicationUIManager is available from the GUIApplication.
- If your application is a single-window application, call the GUIApplication method *setAllowsMultipleOpens(false)*. You can also remove various menu items if certain behavior such as Close and New are not appropriate. You may use *showNewDataOnRun(true)* to open the window at startup. This is normal behavior by default.
- If you are setting the window to a specific size, move that Dimension to the DataView *initializeComponents()* method in a call to *setPreferredSize()*. The hosting window will use *pack()*. Alternately, you may set the size for all windows in the ApplicationWindowsUI using *setPreferredWindowSize()*.
- If your application maximizes windows on startup, get the ApplicationUIManager from the GUIApplication, then get the ApplicationWindowsUI via *getWindowsUI()*, and call *setMaximizeWindows(true)*. We recommend that this be done on an OS by OS basis using an OSApplicationCustomizer. Particularly for platforms such as Mac OS X where users may have very large screens. A maximized window on a 32" Cinema display can be unnerving, unless you are convinced this is appropriate for your application. If you set both the preferredSize in the DataView and selectively set the maximize setting in the ApplicationWindowsUI, your application should behave appropriately on all platforms. You may also choose not to maximize, but to use

the ApplicationWindowsUI method *getPreferredMaximumSize()* which returns a size that is safe on each platform.

- If you are using the JIDE Docking Framework, JDAF will handle the low level details of creating the DockableHolder and DockingManager automatically, per window. Before *run()* simply access the ApplicationUIManager from the GUIApplication and call *setUseJIDEDockingFramework(true)*. If you manage your own persistence canll *setManageLayoutPersiatance(false)*. Then setup your DockableFrames in the *customizeWindow()* method of a WindowCustomizer. You can persist your layout in the *disposingWindow()* method. You can cast the window to a DockableHolder, from which you can obtain the DockingManager. An alternative is to use the DockingApplicationFeature.
- Finally, if there are any other important windows settings, you can use a *WindowCustomizer*. Just keep in mind that different platforms will use different window classes; either JFrame or JInternalFrame. So type-safe your code appropriately.

## Migrating Actions

GUIApplication has an ActionMap. We recommend that you add all of your Actions to this ActionMap under standardized keys, prior to a call to *run()*. This likely will be an extra step if your Action/ActionListener code is coupled to your menu building code via inner classes. If so, just add the inner-class Actions to the application's ActionMap instead of the JMenus or JToolbars. We strongly recommend this step as menus and toolbars will rely on this organization.

Secondarily, consider changing your Actions to extend from GUIApplicationAction, as opposed to AbstractAction where you think you could benefit from the *actionPerformedDetatched()*, and thereby gain a much snappier user interface.

To ensure there is no functional overlap; please see the Actions listed previously in this document. You can expire any of your Actions if we have covered that functionality. This is especially true of file handling.

Alternately you can replace any of the standard Actions. If you use the same ActionMap key (found in the ActionKeys class) your Action will assume the platform properties for the Action of that kind, including name, accelerators, and Icons.

For your Action icons, you can replace our icons. The default Actions are available after the GUIApplication is constructed. This can either be done using the Resource bundles or using an IconTheme (See "Working with Icons"), or just manually as needed.

The next two sections are about menus and toolbars. Be sure to check out the AutoInstallActionsFeature as this lets your define Action properties in such as way that the Acton will self-install into menus and toolbars without writing any menu or toolbar code.

## Migrating Menu Code

GUIApplication manages menubars automatically. The benefit is the provision of OS- recommended menu configurations, which make for an intuitive and integrated application on each given platform.

You should build the menu items using Actions from the ActionMap, as this provides a way of customizing the menu item remotely without having to access the menuing system, which may include multiple menubars, and whose menu items may be configured in different ways.

The framework provides the four most common application menus; File, Edit, Window, and Help. If you have implemented these menus in anyway, please audit against what we provide, and retire any duplications.

You configure menus by adding a MenuBarCustomizer. Keep in mind that the framework manages the Menu Bars. If you are using XML or some other menu configuration method, you may be able to migrate this code to a MenuBarCustomizer.

The framework-provided standard menus will be passed into the *customizeStandardMenus()* method. Use MenuGroups to add Actions to the standard menus that are not already there. MenuGroup represents a span in a JMenu where user items should go according to OS-guidelines.

To access a MenuGroup use the ApplicationMenuBarsUI method *getMenuGroup(string, menu)* method with the desired MenuGrouping constant and input menu.

To add other application menus, implement the *createApplicationMenus(ApplicationMenuBarsUI)*. Create your menus here and return them. They will be placed in the menu bar in the guidelines recommended manner. When creating menus, the ApplicationMenuBarsUI object should be used. You can add items to the standard menus using *addMenuItem()*. Actions can be accessed using *getAction(actionKey)*. Use the factory methods *defaultMenu(String)* and *defaultMenuItem(Action)* to create the menus and items. This makes sure that the ApplicationMenuBarsUI can vend the most appropriate Component.

If you show icons in your application, be sure to set *setShowIcons(true)* in the Application-MenuBarsUI object. You can repress icons in the menus by setting this property to false.

Alternatively, you may use the AutoInstallActionsFeature, which allows Actions to be installed into menus automatically just by setting a few properties.

## Migrating ToolBar Code

GUIApplication manages all toolbars automatically. The benefit is the provision of OS- recommended toolbar configurations, which make for an intuitive and integrated application on each given platform. The framework provides one toolbar called the “Standard” toolbar. If you have implemented toolbars please audit against what we provide, and retire any duplications.

You configure toolbars by adding a ToolBarCustomizer. The ToolBarCustomizer wants to know the names of all your toolbars via *getTooBarNames()*. If they don't have a name, you will need to supply them. The "standard" toolbar will be passed into the *customizeStandardToolbars()* method. To add Actions that are not already in the toolbar, use the input Container. In the *createApplicationToolBar()* method you will be asked to provide a toolbar for each name you returned from *getTooBarNames().*

When creating toolbars, the ApplicationToolBarsUI object should be used. Actions can be accessed using *getAction(actionKey).* Use the factory methods *defaultToolBar(String)* and *defaultButton(Action)* or *addToolBarButton()* to create the components. This makes sure that the ApplicationToolBarsUI can vend the most appropriate Component.

If you are using the JIDE Action Framework, JDAF will do the low level integration automatically. Just set the ApplicationUIManager method *setUseJIDEActionFramework(true).* If you use the factory methods as prescribed above to create toolbar items, there is no further work needed. If you need to access the DockableBarManager, you just access the window using the ApplicationWindowsUI object *getWindow(DataView)*, and cast it to a DockableBarHolder, then access the DockableBarManager. This is generally done inside a DataViewListener, where the Window can be accessed directly from the DataViewEvent.

Alternatively, you may use the AutoInstallActionsFeature, which allows Actions to be installed into toolbars automatically just by setting a few properties.

## Migrating Printing Code

Printing in JDAF is facilitated by the Printer. Printer provides the printing infrastructure for a DesktopApplication. A Java Printing API *Pageable* provides the printing layout/rendering. Therefore it should be straightforward to move your printing code to the Printer.

What may take more thought is how the Pageable is delivered. The DataViewPane, where your GUI was migrated, to is a *PrintSource*. A PrintSource is chiefly concerned with delivering a Pageable instance. Therefore when the PrintAction is executed, the GUIApplication checks the front-most DataView for a Pageable. Hence, to get your Pageable to the Printer, override/implement the DataViewPane *getPageable()* method. If null is returned, the Print actions are disabled. Alternately you can set the Printer disabled using the Printer method *setEnabled(false)*.

If the PrintSource system will not work, simply create your own Print action and set it under the ActionKeys.PRINT. It will assume the platform-specific formatting of a Print menu item.

## Migrating the Help System

If you have a help system, likely it has a menu item that invokes it. GUIApplication provides help via setting a HelpSource implementation into the application via *setHelp(HelpSource).* You can use this interface to wrap and invoke your help system.

If you have a single help menu item, you are done. The UI handles it. If your help menu has more than one help command, we recommend you add a HelpAction for each additional help command into the application ActionMap. For each HelpAction provide a hint via *setHelpTopic(String)*. This will be passed to your HelpSource implementation. Then in the MenuBarCustomizer, in the *customizeStandardMenus()* method, when the MenuConstants.HELP_MENU_ID is passed in, add the HelpActions to the HELP_HELP_GROUP_ID MenuGroup.

If you have help topics attached windows, set the topic in a client property of the DataView components with the key *HelpSource.HELP_HINT_PROPERTY_NAME.*

## Migrating Preferences

Migrating preferences is a bit tricky. DesktopApplication uses the Java Preferences API. You can simply use Preferences node returned from *getPreferences().* However, there is no heavy enforcement of a Preferences mechanism.

If you are already using the Preferences API, we recommend that you continue using it in the way you are using it now, and ignore our Preferences node, unless you want to go through the trouble of migrating the values to the application Preferences node.

If you are managing preferences via a Properties object, you should continue to do so, unless you want to go through the trouble of migrating the values to our Preferences node. One benefit would be to use the PropertiesFileFormat to load and save your data, and expire your code. DesktopApplication provides a *getDataDirectory()* that will return the platform-specific location where such files should be stored.

GUIApplication does however suggest that if you have a Preferences dialog, you let the application know by adding is as a queued DialogRequest in the ApplicationDialogsUI using the key:

ApplicationDialogsUI.PREFERENCES_DIALOG_REQUEST_KEY

To use your existing dialog, add a queued *CompatibleDialogRequest*. If the dialog setup code is in the calling method, as opposed to the dialog itself, you may need to migrate this code into a dialog subclass.

Additionally, you need to add the PreferencesAction to the GUIApplication's ActionMap.

```
application.getActionMap().put(PreferencesAction.KEY, new PreferencesAction());
```

If you want to fully migrate your preference dialog, consider using the *PreferencesDialogRequest* object.

If you want better guidelines capability, you may consider migrating your content to subclass our PreferencesPane.

### Migrating About Window

If you have an About dialog, simply let the GUIApplication know by queuing it in the ApplicationDialogsUI using a CompatibleDialogRequest under the key:

```
ApplicationDialogsUI.ABOUT_DIALOG_REQUEST_KEY
```

This will allow the AboutAction to invoke it.

If the About dialog setup code is in the calling method, as opposed to the dialog itself, you may need to migrate this code into a dialog subclass.

### Migrating Startup and Shutdown Functionality

If you have startup functionality such as splash screens, security checks, license validations, etc., we recommend moving that code into the *applicationOpening()* and *applicationOpened()* events of one or more ApplicationLifecycleListeners.

If you have shutdown functionality such as saving states, preferences, or general cleanup tasks, use the *applicationClosing()* and *applicationClosed()* events.

Note that it is unnecessary to invoke the *GUIApplication.run()* on the event thread. GUIApplication starts up the UI on the event thread internally.

## JSR-296 Compatibility

JSR-296 (appFramework) is a java.net community project created to provide an API that addresses the nominal needs of Swing-based application. JDAF provides a JSR-296 compliancy API for those who are either migrating from it, or those who wish to use the features provided by it.

This section assumes an understanding of JSR-296.

*AppFrameworkApplication* facilitates using JDAF as a JSR-296 compliant Application. AppFrameworkApplication is a JSR-296 Application subclass that wraps a JDAF DesktopApplication instance. The instance can be accessed via the *getJDAFApplication()* method.

To use AppFrameworkApplication, subclass it and implement the method *createJDAFApplication(String name)* to return a JDAF DesktopApplication as discussed otherwise in this document. Launch your application in the normal JSR-296 way, using the *Application.launch()*.

AppFrameworkApplication provides the following integration with JSR-296:

- AppFrameworkApplication delegates the JSR-296 startUp() and exit() methods to JDAF methods run() and exit(), respectively. So it is unnecessary to implement these methods.
- In the spirit of the JSR-296 ExitListeners, AppFrameworkApplication implements the ApplicationLifecycleListener interface. Simply override the desired methods. Note that JSR-296 ExitListeners will fire before JDAF *applicationClosing()* method.

- AppFrameworkApplication configures a “tuned” Resources object that facilitates the recommended resource bundle configuration of JSR-296. This instance can be accessed via the *getResources()* method. Set up your resource bundles in the JSR-296 manner by creating a directory called "resources" off your AppFrameworkApplication subclass root package, and naming the bundle file the same name as your class.

  AppFrameworkApplication will read the "Application.name", “Application.vendor” and "Application.version" properties automatically, and is configured to load icons from that resources location. In this way you can use both JDAF and/or the JSR-296 methods of resource access.
- The command line is automatically captured. You may use JDAF CommandLine class to access program arguments.
- Similar to the *ApplicationContext.getInstance().getApplication()* we provide *AppFrameworkApplication.getApplication()* to provide a static handle to JDAF DesktopApplication.

Consider using JDAF DataViewListeners to use JSR-296 features like resource injection. You could also use the ApplicationLifecycleListener methods to utilize the JSR-296 SessionStorage mechanism.

Note: A JSR-296 implementation is *not* included in JDAF package, but will be needed to use this compliancy API. This API was designed to an early 0.42 version of the JSR-296. See Swing Application Framework for details and downloads of a default JSR-296 implementation.

## Internationalization Support

All Strings used in JDAF are contained in properties files throughout the packaging structure. In you deploying to a non-english speaking locale, please include the '*jide-properties.jar*' in your classpath to take advantage of any existing localization.

Note that we haven't done any localization for JDAF in particular. If you want to support languages other than English, just extract the properties files, translate them to the language you want, use the correct locale postfix in the file name, and then jar it back into JDAF jars.

Each properties file is suffixed by an OS variant. This is an *extended* variant, meaning you can still use your own variant postfix, just be sure to append the OS extended variant at the end of the file name. In fact, this extended variant can be added at any locale reduction level, making a resource file OS sensitive by default, language, language+country, or language+country+variant.

You are welcome to send the translated properties files back to us if you want to share them!