JIDE Action Framework Developer Guide

Contents

Purpose of This Document

Welcome to the *JIDE Action Framework*, the product that can give your application's toolbars and menu bar more features and better look.

This document is for developers who want to develop applications using *JIDE Action Framework*.

What is JIDE Action Framework

Toolbar and menu bar are two very important UI components. Menu bar usually contains all functionalities of the application. Toolbar provides easy access to some of the functionalities that are available on menu bar as well. Though they look like two different UI components, they have so many

commonalities. For example, both are collections of actions. This is also the reason why we call this product *JIDE Action Framework*. In *JIDE Action Framework*, we introduce a new component called *CommandBar*. It will be the replacement for both menu bar and toolbar[1].

An important part of *JIDE Action Framework* is to support several styles. Just like we introduced Visual Studio .NET style in the release of *JIDE Docking Framework*, this time we introduce Office 2003 style along with the release of *JIDE Action Framework*. Later, we also introduced several other styles such as Xerto style, Office2007 style etc. See below for a screenshot of Office2003 style in different themes as well as the Office 2007 styles with toolbar addition.

Figure 1 Office 2003 Style (in three themes) and Office 2007 Style

[1] Why do we create a new component instead of reusing JToolBar and JMenuBar? The main reason is we want to implement both components in the same way. If we kept them as separate components with a common class to extend from, it will be hard to do so. In fact, *CommandBar* still extends *JMenuBar*.

Packages

The table below lists the packages in the *JIDE Action Framework*.

Packages	Description
com.jidesoft.action	Action Framework related components and class, such as CommandBar, DockableBar, DockableBarManager etc.
com.jidesoft.plaf	L&F classes for components in Action Framework

Migrating from Existing Applications

Since most of you have already built toolbar and menu bar before the release of *JIDE Action Framework*, we treated easy-to-migrate as a high priority when we designed the *JIDE Action Framework*. This chapter will help you to understand the migration steps.

We assume you are using *JToolBar* and *JMenuBar*, the two Swing components. So the migration steps below are based on this assumption.

In Swing, *JToolBar* and *JMenuBar* are two different components. However in *JIDE Action Framework*, we used a single component to replace both of them. It is called *CommandBar*.

To help you understand what are in *JIDE Action Framework*, we prepared a table below describing the relationship between Swing component and corresponding JIDE components. If you used any component on the left side, replace them with the one on the right side when you migrate.

Swing	JIDE
JMenuBar	CommandMenuBar (extends CommandBar)
JToolBar	CommandBar
JMenu	JideMenu (JideMenu extends JMenu)
JMenuItem	No change
JCheckBoxMenuItem	No change
JRadioButtonMenuItem	No change

JButton (only when it is used in JToolBar)	JideButton (JideButton extends JButton)
JToggleButton (only when it is used in JToolBar)	JideToggleButton
N.A.	JideSplitButton (combination of button and menu)

Except *CommandBar* and *CommandMenuBar* which are under com.jidesoft.action, all other components listed in the table above such as *JideButton*, *JideToggleButton*, *JideSplitButton* are under com.jidesoft.swing.

To show how easy it is to migrate, we used SwingSet2 demo which is included in any Java JDK demo directory. To best demonstrate, please use Windows XP with XP theme on in order to see the new Office2003 style. To make it easy for you, we included a modified version of SwingSet2 at "examples/A3. SwingSet2".

1. Open SwingSet2.java
2. Set the L&F and the Office2003 style at the beginning of SwingSet2 constructor.

```
try {
   UIManager.setLookAndFeel(WindowsLookAndFeel.class.getName());
}
catch (ClassNotFoundException e) {
}
catch (InstantiationException e) {
}
catch (IllegalAccessException e) {
}
catch (UnsupportedLookAndFeelException e) {
}

LookAndFeelFactory.installJideExtension(LookAndFeelFactory.OFFICE2003_STYLE);
```

3. Replace all *JMenuBar* with *CommandMenuBar* (Whole Word Only, Case Sensitive)
4. Fix a compile error at *setJMenuBar*. Comment *setJMenuBar* out and add *menuBar* to *toolbarPanel* which is used to add the old toolbar.

5. After "CommandBar menuBar = new CommandMenuBar();", add "menuBar.setStretch(true);". This method is not required but it will make it looks more like a menu bar. You can try not to call this method and see what you get as an experiment.
6. Replace all *JToolBar* with *CommandBar* (Whole Word Only, Case Sensitive)
7. Replace all *JButton* which is used in toolbar with *JideButton* (Whole Word Only, Case Sensitive). Actually in SwingSets, there is no such JButton.
8. Replace all *JToggleButton* which is used in toolbar with *JideToggleButton* (Whole Word Only, Case Sensitive).
9. Now run SwingSet2 demo. See screenshot below.

Figure 2 SwingSet2 demo with CommandBar etc components

With just two minutes of work, you got a nice Office 2003 L&F toolbar and menu bar. Now only that, Try to make the demo window smaller and see what happen. See screenshot on the right side. Ah ha, the new toolbar can hide some buttons when there is not enough space. You can still access those hidden buttons by clicking the chevron at the end.

You might start to try to drag the grippers on the the new toolbar and menu bar since they look very attractive. Hmm! Nothing happened. Don't worry. Let's get to the next step.

Now you are using command bars from *JIDE Action Framework*, but that's not enough. If you put all command bar into a manager, you will be able to drag and drop them. It is just like *JIDE Docking Framework*.

1. Good news is you don't need to touch the two command bars you've created. All you need to do is to give them a name. In the case of the old menu bar, you can specify it in the constructor.

```
CommandBar menuBar = new CommandMenuBar("Menu Bar").
```

In the case of the old tool bar, the easiest way is to change the old constructor and pass the name to super.

```
public ToggleButtonToolBar() {
   super("ToolBar");
   getContext().setInitIndex(1);
}
```

2. Changes SwingSets class, making it extends *ContentContainer* and implements *DockableBarHolder*.

```
public class SwingSet2 extends ContentContainer implements DockableBarHolder {
…
```

3. Implements *DockableBarHolder* by adding the following code at the end of SwingSet2 class. We will need this *DockableBarManager* to manage the two command bars we created.

```
DockableBarManager _dockableBarManager;

public DockableBarManager getDockableBarManager() {
   return _dockableBarManager;
}
```

4. Let's create *DefaultDockableBarManager* and add both command bars to it. All you need do is to create *DefaultDockableBarManager*, add command bars to the manager instead of adding to SwingSet2 panel directly. Instead of adding the content (the tabbed pane) directly to SwingSet2 panel, add it to the CENTER of *getMainContainer()* of the manager. It's just like you did in *JIDE Docking Framework* if you used it before.

```
public void initializeDemo() {
   menuBar = createMenus();
   popupMenu = createPopupMenu();
```

```
        toolbar = new ToggleButtonToolBar();

        // DockableBarManager
        _dockableBarManager = new DefaultDockableBarManager(frame, this);
        _dockableBarManager.setProfileKey("swingset2");
        _dockableBarManager.addDockableBar(menuBar);
        _dockableBarManager.addDockableBar(toolbar);
        _dockableBarManager.getMainContainer().setLayout(new BorderLayout());

        tabbedPane = new JTabbedPane();
        _dockableBarManager.getMainContainer().add(tabbedPane, BorderLayout.CENTER);
        tabbedPane.getModel().addChangeListener(new TabListener());

        statusField = new JTextField("");
        statusField.setEditable(false);
    // let's not add status field for now.
    //      add(statusField, BorderLayout.SOUTH);

        // after this, it's the same as before.
        demoPanel = new JPanel();
    ....
        tabbedPane.addTab(
                getString("TabbedPane.src_label"),
                null,
                scroller,
                getString("TabbedPane.src_tooltip")
        );
    // till the end of the method, added
        getDockableBarManager().loadLayoutData();
    }
```

5. Now let's run it. It looks the same as last time we run. But if you drag the gripper of both command bars, Yeah! It can be dragged now. You can make it floating, resizing the floating one, or drag it back to north side or dock to three other sides etc.

Figure 3 SwingSet2 demo using DockableBarManager

Isn't it cool? Just in a few minutes, you convert SwingSet2 demo to use *JIDE Action Framework*. And all of a sudden, SwingSet2 demo got the nice looking toolbar and menu bar. They match the style of Office 2003 application and they can be dragged and dropped, floated, and closed. In fact, it is just like a native window application. Now keep the SwingSet2 demo running, go to your Windows XP Display Property and change the theme to the other theme such as default blue theme, you will see SwingSets demo changes too. You are using the theme technology we implement. In fact, you have total control on look and feel and we will cover it in a section later.

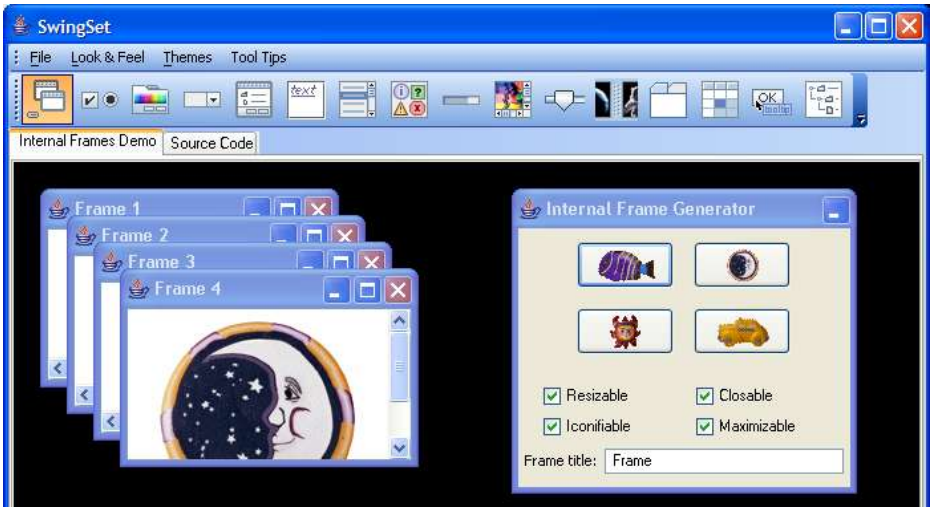


Figure 4 SwingSet2 demo with XP blue theme

Now, I hope you are ready to jump in and migrate your application to *JIDE Action Framework* and give your application a nice look.

DockableBarManager

If you worked with *JIDE Docking Framework* before, this chapter will sound familiar.

You may wonder why the manager is called *DockableBarManager* not *CommandBarManager*. It's a good question. *CommandBar* is not the base class. *DockableBar* is. *CommandBar* extends *DockableBar*. Right now *CommandBar* is the only one that extends *DockableBar*. However we will have more in the future. The base class that works with *DockableBarManager* is the *DockableBar*, that's why we called it *DockableBarManager*.

The easiest way to get start with *DockableBarManager* is to make your *JFrame* extending *DefaultDockableBarHolder*. You can refer to A1.SampleWord as an example. What if you are already using *JIDE Docking Framework* and your *JFrame* already extends *DefaultDockableHolder*? If so, please use *DefaultDockableBarDockableHolder* which allows you to use both *Docking Framework* and *Action Framework*. A2. SampleVsnet is such an example which uses both frameworks.

You will create your *CommandBar* somewhere else. Once you have them ready, you just need to add them to *DockableBarManager* by calling *_frame.getDockableBarManager.addDockableBar()*.

```
// add command bar
_frame.getDockableBarManager().addDockableBar(…);
_frame.getDockableBarManager().addDockableBar(…);
_frame.getDockableBarManager().addDockableBar(…);
_frame.getDockableBarManager().addDockableBar(…);
```

The ContentPane of *JFrame* is set to *BorderLayout* automatically. *DockableBarManager* will take over the CENTER of the ContentPane. You can still use NORTH or SOUTH or WEST or EAST of the ContentPane. For example, for status bar, you can add it to SOUTH.

```
// add status bar
_statusBar = createStatusBar();
_frame.getContentPane().add(_statusBar, BorderLayout.AFTER_LAST_LINE);
```

For those components you used to add to the CENTER of *JFrame*'s ContentPane, now you need to add them to the *MainContainer* of *DockableBarManager* which you can get by calling *_frame.getDockableBarManager.getMainContainer()*.

```
_documentPane = createDocumentTabs(); // the components in the center
_frame.getDockableBarManager().getMainContainer().setLayout(new BorderLayout());
```

```
_frame.getDockableBarManager().getMainContainer().add((Component) _documentPane,
BorderLayout.CENTER);
```

At the end, call *loadLayoutData()* to load layout data from either javax preference or . If there is no layout saved previously, it will place the *DockableBars* at their default location.

```
_frame.getDockableBarManager().loadLayoutData();
```

Dockable Bar

Similar to *DockableFrame*, *DockableBar* also has a context which you can set the initial position and state.

There are several methods you can use to set the initial default setting: setInitMode(), setInitSide() and setInitIndex() or setInitSubindex. For example, if you want to put the dockable bar to the south side, you just need to set the init side to DOCK_SIDE_SOUTH. Here are the possible combinations of those values:

initMode	InitSide	initIndex	initSubindex	Comments
STATE_HORT_DOCKED	DOCK_SIDE_NORTH DOCK_SIDE_SOUTH	The row index. Any integer greater than 0	The start x	Dockable bars with the same initIndex will be on the same row.
STATE_VERT_DOCKED	DOCK_SIDE_EAST DOCK_SIDE_WEST	The column index. Any integer greater than 0	The start y	Dockable bars with the same initIndex will be on the same column.
STATE_FLOATING	N/A	N/A	N/A	The location is specified undockedBounds. Leave width and height as 0 to use the preferred size.
STATE_HIDDEN	N/A	N/A	N/A	

Here is a typical code to initialize a DockableBar. The code below will create a dockable bar on south side and at row index 0 (the first row) and start x is 100.

```
public static CommandBar createDrawingCommandBar() {
   CommandBar commandBar = new CommandBar("Drawing");
   commandBar.getContext().setInitSide(DockableBarContext.DOCK_SIDE_SOUTH);
   commandBar.getContext().setInitIndex(0);
   commandBar.getContext().setInitSideSubindex(100);
   ……
}
```

Manipulate Dockable Bar

Once the dockable bars have been added to *DockableBarManager*, the *DockableBarManager* will manage them based on the user's keyboard and mouse action. They can either be shown or hidden and they may also be docked, or floating. All these operations are done through the *DockableBarManager*. Here are some commonly used methods on *DockableBarManager*:

showDockableBar(): Show a dockable bar

hideDockableBar(): Hide a dockable bar

toggleState(): Toggle between floating state and docked state.

dockDockableBar(final DockableBar f, final int side, final int row, final boolean createNewRow, final int start): dock dockable bar at the specified side, row and start location. If the side is north or south, start is the x coordinate. If the side is east or west, start is the y coordinate.

floatDockableBar(final DockableBar f, final Rectangle bounds): float dockable bar at the specified bounds.

Please refer to the *DockableBarManager* javadoc for more details.

Dockable Bar Event

We support twelve events that are specific to dockable bar.

- DOCKABLE_BAR_ADDED: when *DockableBar* is added to *DockableBarManager*.
- DOCKABLE_BAR_REMOVED: when *DockableBar* is removed from *DockableBarManager*.
- DOCKABLE_BAR_SHOWN: when *showDockableBar* is called on the *DockableBar*.
- DOCKABLE_BAR_HIDDEN: when *hideDockableBar* is called on the *DockableBar*.

- DOCKABLE_BAR_HORI_DOCKED: when *DockableBar* changes from other states to HORI_DOCKED state.
- DOCKABLE_BAR_VERT_DOCKED: when *DockableBar* changes from other states to VERT_DOCKED state.
- DOCKABLE_FRAME_FLOATING: when *DockableBar* changes from other states to FLOATING state.

Styles and Look And Feels

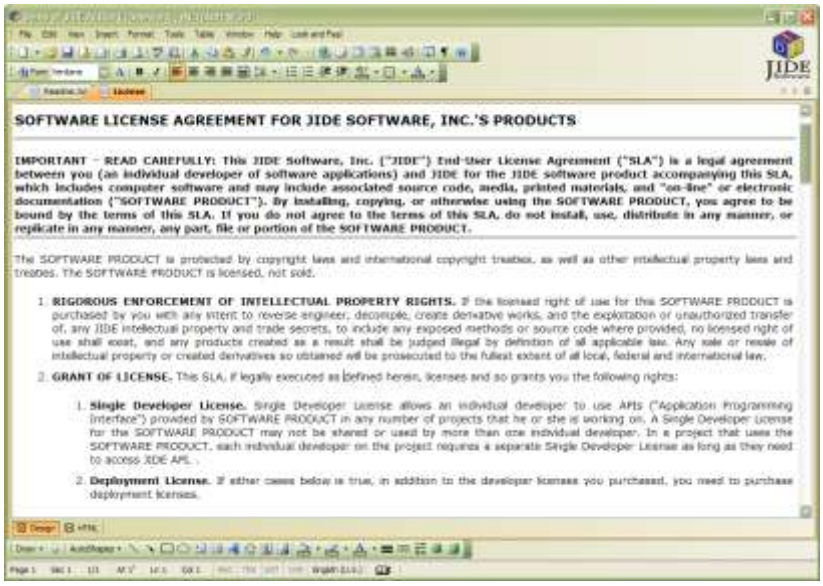
Our products always come with modern styles to match with existing look and feels. Our first product *JIDE Docking Framework*'s interface concept was based on Visual Studio .NET. Since then, the user interface changed a lot. First, Windows XP introduced a new theme which overhauled the look and feel of Windows. Then Microsoft Office 2003 is released with fully support of XP style and its Office 2003 style addition. Since then Office 2003 style because a new style that everyone wants to follow. Unfortunately before the release of *JIDE Action Framework*, there is no public library available to have such an L&F style using Swing. To fill in this obvious gap, along with the introduction of *JIDE Action Framework*, we introduce the Office 2003 LookAndFeel.

In this release, we include all four themes for Office 2003 style. They are the same as Windows XP - Blue, HomeStead[2], Silver and Gray. It's very easy to add your own color theme. We will expose it later after it gets stabilized in a few releases. In addition to the four standard themes, there is another theme we called it Default theme. Different from the four standard themes which defined their own color palette, this Default theme derived all colors from existing UIDefaults such as "control", "controlShadow". So if you are using Windows classic style L&F (not XP) and a color theme other than "Windows Standard" on Windows display property, Default theme will be the preferred one to use.

We also added the *CommandBar* related UI classes for the original Vsnet L&F and Eclipse L&F. There are also UI classes for Metal L&F and Aqua L&F. For other third party L&Fs, you can always use one of the existing UI classes and tweak them to fit in the style of that L&F. Please refer to LookAndFeelFactory.java for more details. In that class, we add code to tweak two famous L&Fs - JGoodies Plastic3D L&F and Alloy L&F. Later, we also introduced Office 2007 style to match with the theme of Microsoft Office 2007.

[2] Also know as Oliver Green. HomeStead is the name used internally

If you choose Windows L&F + Office 2003 style, you will get seamless integration on Windows XP. When user changed the system theme from Blue to Silver for example on Windows XP, your application will change theme with it[3].



On the other hand, we also enhanced *JIDE Docking Framework* to have Office 2003 L&F *JideTabbedPane*, *DockableFrame* and *SidePane*. See below for an example of a sample application that mimics Microsoft Word and another sample application that mimics Visual Studio .NET. Both samples, of course, used *JIDE Action Framework.*

Figure 5 Sample application mimicking MS Word

[3] We leave the native integration as an option. You can turn it on or off using Office2003Painter.setNative(true/false). A save way to do this is to call Office2003Painter.setNative(SystemInfo.isWindowsXP()) so that it only uses native on XP.

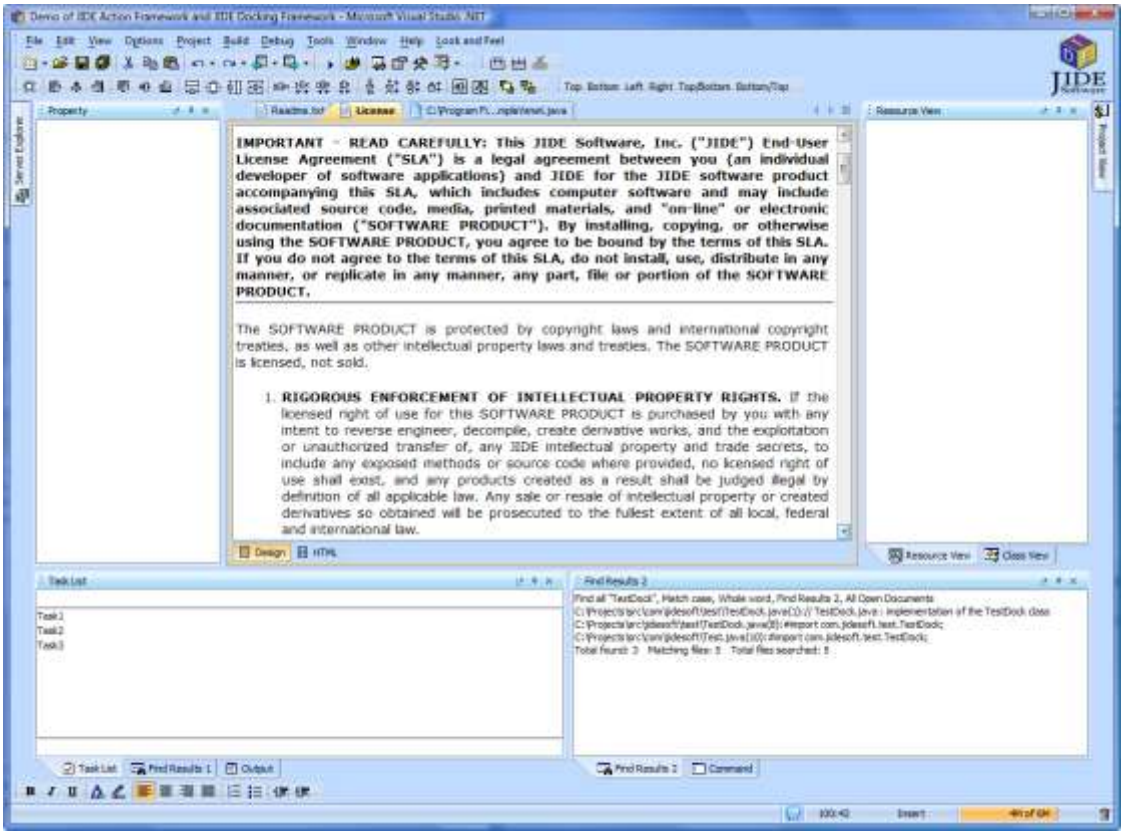


Figure 6 Sample application mimicking Visual Studio .NET under Office 2007 style

It is also very easy to use those look and feels. All you need to do is to set the base L&F such as Windows, or Metal or Aqua. After that, call *LookAndFeelFactory.installJideExtension()*. In the case of multiple styles to choose from, you also need to pass in the style as parameter. The following code will set the L&F to Windows and use Office 2003 style.

```
UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());

// to add additional UIDefault for JIDE components
LookAndFeelFactory.installJideExtension(LookAndFeelFactory.OFFICE2003_STYLE);
```

If the base L&F is Windows L&F, there are three styles you can choose from. They are VSNET_STYLE, OFFICE2003_STYLE and ECLIPSE_STYLE respectively.

For OFFICE2003_STYLE, there are four build-in themes. They are BLUE, HOMESTEAD, METALLIC and GRAY. The constants are all defined in XPUtils. The following code will set to BLUE theme.

```
((Office2003Painter) Office2003Painter.getInstance()).setColorName(XPUtils.BLUE);
// update all UIs
frame.getDockableBarManager().updateComponentTreeUI()
```

Aqua LookAndFeel and Mac OS X

We also support Mac OS X using Aqua LookAndFeel. See the screenshot below.

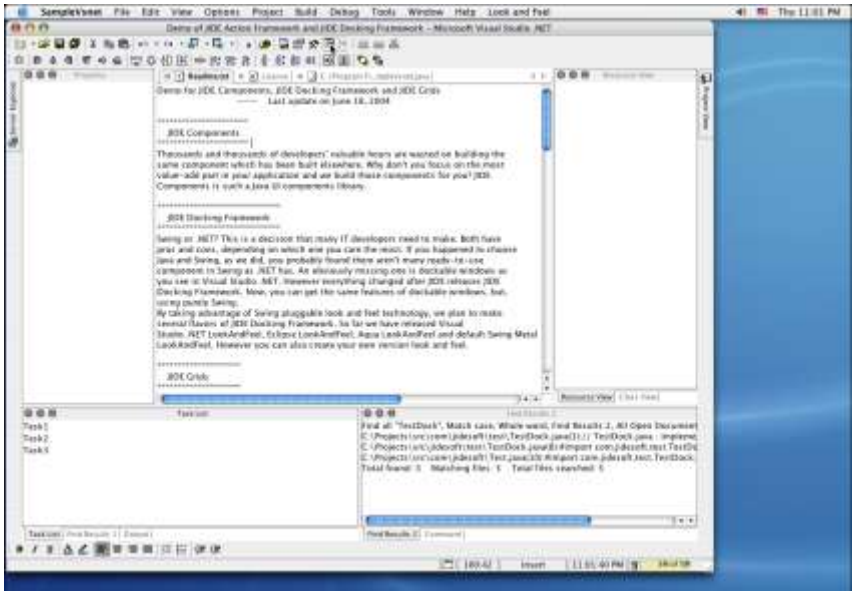


Figure 7 Aqua LookAndFeel on Mac OS X

Mac OS X has a special menu bar called screen menu bar. *JIDE Action Framework* supports it too. To get the screen menu bar, you need to make sure

1. You set the system property "apple.laf.useScreenMenuBar" to "true". You can do it either use –D in the command line or call System.setProperty() in your code.
2. You need to make sure you can use CommandMenuBar when you create the CommandBar which you want to use as menu bar. [4]

[4] The reason we emphasize this is because under other L&Fs, you can create a *CommandBar* instance first, then call setMenuBar(true). However under Aqua L&F, you must create *CommandMenuBar* instance directly or you will get a

```
CommandBar menuBar = new CommandMenuBar("Menu Bar");
```

Once you use the screen menu bar, *JIDE Action Framework* will lose the control over that menu bar (even though you did call *addDockableBar* to *DockableBarManager* just like other *CommandBar*s.) User can't drag the menu bar like before. This beats the purpose of *JIDE Action Framework*. However this is exactly what Mac OS X users will expect.

CommandBar/Chevron UIDefaults

Name	Type	Description
CommandBar.font	Font	The font for CommandBar
CommandBar.background	Color	The background of CommandBar
CommandBar.foreground	Color	The foreground of CommandBar
CommandBar.border	Border	The border for CommandBar when it is docked horizontally
CommandBar.borderVert	Border	The border for CommandBar when it is docked vertically
CommandBar.borderFloating	Border	The border for CommandBar when it is floating
CommandBar.titleBarSize	Integer	The title bar height
CommandBar.titleBarBackground	Color	The title bar background
CommandBar.titleBarForeground	Color	The title bar foreground
CommandBar.titleBarFont	Font	The font for CommandBar's title bar.
CommandBar.minimumSize	Dimension	The minimum size of a ComamndBar
CommandBar.separatorSize	Integer	The separator width (or height if vertically)
CommandBarSeparator.background	Color	The separator's background
CommandBarSeparator.foreground	Color	The separator's foreground
Chevron.size	Integer	The width (or height if vertically) of the Chevron
Chevron.alwaysVisible	Boolean	If there is no hidden components on CommandBar, should the Chevron be visible?

ClassCastException during application startup. To be safe, we always recommend you to use *CommandMenuBar* for *CommandBar* which you will use as menu bar. This way will work under all L&Fs.

Persisting Layout Information

JIDE Action Framework offers the ability to save the exact position of dockable bar information between sessions, using the javax.utils.pref package. This means that under Windows, the information will be stored in the registry, while under UNIX, it will be stored in a file in your home directory.

All layout data are organized under one key called the 'profile key'. This can be any string, but usually it's your company name (we use "jidesoft" in our sample application). You should call **setProfileKey(String key)** to set this key when your application starts up.

Under the profile key, there is a name for each layout configuration. The configuration supports multiple sets of window positions, and can also be used for storing other information, such as user preferences. Thus, when John runs your application, he doesn't have to use the same window layout that Jerry used. The default set of preferences lies under the key "default", and is used whenever **loadLayoutData()** and **saveLayoutData()** are called to persist the window state.

If you prefer to specify the configuration, then **loadLayoutDataFrom(String layoutName)** and **saveLayoutDataAs(String layoutName)** will persist the window state under the key profileName. This is what you would use for the user preferences example above, or for distinct projects or workspaces, etc**.**

getLayoutRawData() and **setLayoutRawData(String layoutData)** are methods allowing you get the layout data as a byte[], in case you want to load/save it without using javax.util.pref.

If you prefer that *JIDE Action Framework* use a file, rather than the registry, then simply use **loadLayoutDataFromFile(String filename)** and **saveLayoutDataToFile(String filename).** The filename param is, as you would expect, the destination of the configuration data.

Another option you have is to let *JIDE Action Framework* use its default file location. By default it uses javax.util.pref to store layout information. However if you prefer disk storage, but want JIDE to manage the location, you can call **setUsePref(false)** to disable using javax.util.pref. Your layout data will be stored at {user.home}/.{profileName}, where profileName is either "default" or your profile name as specified above. If you want to specify where to store the layout data, you can call **setLayoutDirectory(String dirName)**. Please note, the directory will be used only when setUsePref is false. You also need to make sure you call set those values (i.e. **setProfileKey()**, **setUsePref()**, **setLayoutDirectory()**) before you call any loadLayout or saveLayout methods.

Once you decide to use preference or save as file, you can use several methods to check if a layout is available or get a list of all layouts saved before. **isLayoutAvailable(String layoutName)** will tell you if a layout is available. **getAvailableLayouts()** will return you a list of layout names. **removeLayout(String layoutName)** will remove the saved layout.

Each stored layout has a version number assigned. If the returned version doesn't match the expected value then the layout information will be discarded. For example, if your application has changed a lot since it was last released to users, you may not want the user's old layout information to be used. You can just call **setVersion(short)** to set the framework to a new version. This means that when a user runs your application, the previously stored layout information will not be used.

You can switch between layouts at any time and each layout can have a different set of dockable frames. In order to function correctly, you need to call **beginLoadLayoutData()** first and then call **addDockableBar()** or **removeDockableBar()**. In the end, you should call one of the **loadLayoutData()** methods to load the layout. Please note that if you add a frame between calling beginLoadLayoutData() and loadLayoutData(), the frame will not be visible until loadLayoutData() is called. However if you add a frame before calling beginLoadLayoutData() or after loadLayoutData(), then the frame will be visible immediately.

Usually the user wants the main window's bounds and state (as in **JFrame.setExtendedState()** or **JFrame.setState()** for JDK1.3 and below) to be part of the layout information so that the information can be persistent across sessions. This means that when you switch layout, not only is the layout of dockable window reloaded but also the location and size of the main window. If you wish, you can disable this default behaviour of saving the main window's bounds and state by calling **setUseFrameBounds(boolean)** and **setUseFrameState(boolean)**.

Integration with JIDE Docking Framework

We've addressed some points in chapters above. However we will discuss further in this chapter.

Layout

Both *JIDE Action Framework* and *JIDE Docking Framework* need to persist layouts. However when you use both frameworks, you don't want to save two layout files.

All methods related to layout are exactly the same in both frameworks. This is enforced because both *DockingManager* and *DockableManager* extend a common class called *AbstractLayoutPersistence*. In order to use both frameworks as one unit, we introduced a new class called *LayoutPersistenceManager*. This guy can manage multiple instance of *LayoutPersistence*.

If you want to use both frameworks, we recommend you to use *DefaultDockableBarDockableHolder* as your *JFrame*. This class will create *LayoutPersistenceManager* automatically and expose it through *getLayoutPersistence()* method.

```
// add dockable bar to DockableBarManager
_frame.getDockableBarManager().addDockableBar(…);
_frame.getDockableBarManager().addDockableBar(…);
_frame.getDockableBarManager().addDockableBar(…);
…

// add dockable frame to DockingManager
_frame.getDockingManager ().addFrame(…)
_frame.getDockingManager ().addFrame(…)
```

```
_frame.getDockingManager ().addFrame(…)
…

// instead of using getDockingManager() or getDockableManager() to load layout
// use getLayoutPersistence() instead.
// _frame.getDockingManager().loadLayoutData();
// _frame.getDockableBarManager().loadLayoutData();
_frame.getLayoutPersistence().loadLayoutData();
```

In short, you should call all layout related methods from *getLayoutPersistence()* instead of using *DockingManager* or *DockableBarManager*. For the list of methods related to layout, you can look at the javadoc of *LayoutPersistence*.

Is there any case that you still should call to *DockingManger* or *DockableBarManager* for the layout related methods? Yes. Below is such an example.

When recovering from autohide all, we just need to restore the layout of *DockingManager*. So we need to store the layout of *DockingManager* by calling getDockingManger().getLayoutData(). See the two lines in red.

```
JMenuItem item = new JMenuItem("Toggle Auto Hide All");
item.setMnemonic('T');
item.addActionListener(new AbstractAction() {
   public void actionPerformed(ActionEvent e) {
      if (!_autohideAll) {
         _fullScreenLayout = frame.getDockingManager().getLayoutRawData();
         frame.getDockingManager().autohideAll();
         _autohideAll = true;
      }
      else {
         // call next two methods so that the farme bounds and state will not change.
         frame.getDockingManager().setUseFrameBounds(false);
         frame.getDockingManager().setUseFrameState(false);
         if (_fullScreenLayout != null) {
            frame.getDockingManager().setLayoutRawData(_fullScreenLayout);
         }
         _autohideAll = false;
      }
```

```
    }
});
menu.add(item);
```

Base JFrame class

In *JIDE Docking Framework*, we have a *JFrame* class called *DefaultDockableHolder*. You can use this class to replace your *JFrame* and it can automatically provide the function of dockable windows. In *JIDE Action Framework*, we also have a *JFrame* class call *DefaultDockableBarHolder*. It can provide function of command bars. What if you want both dockable window and command bar? Since there iss no multiple inheritances in Java, you can not extend both *DefaultDockableHolder* and *DefaultDockableBarHolder*. We have to introduce a new class called *DefaultDockableBarDockableHolder*. So if you want to use both *JIDE Action Framework* and *JIDE Docking Framework*, use *DefaultDockableBarDockableHolder* as your *JFrame*.

The screenshot below illustrates the relationship of each area when using both frameworks. The area in the red rectangle is managed by *JIDE Docking Framework*. The area between red rectangle and green rectangle are managed by *JIDE Action Framework*. If you use *DefaultDockableBarDockableHolder*, this is what you get.

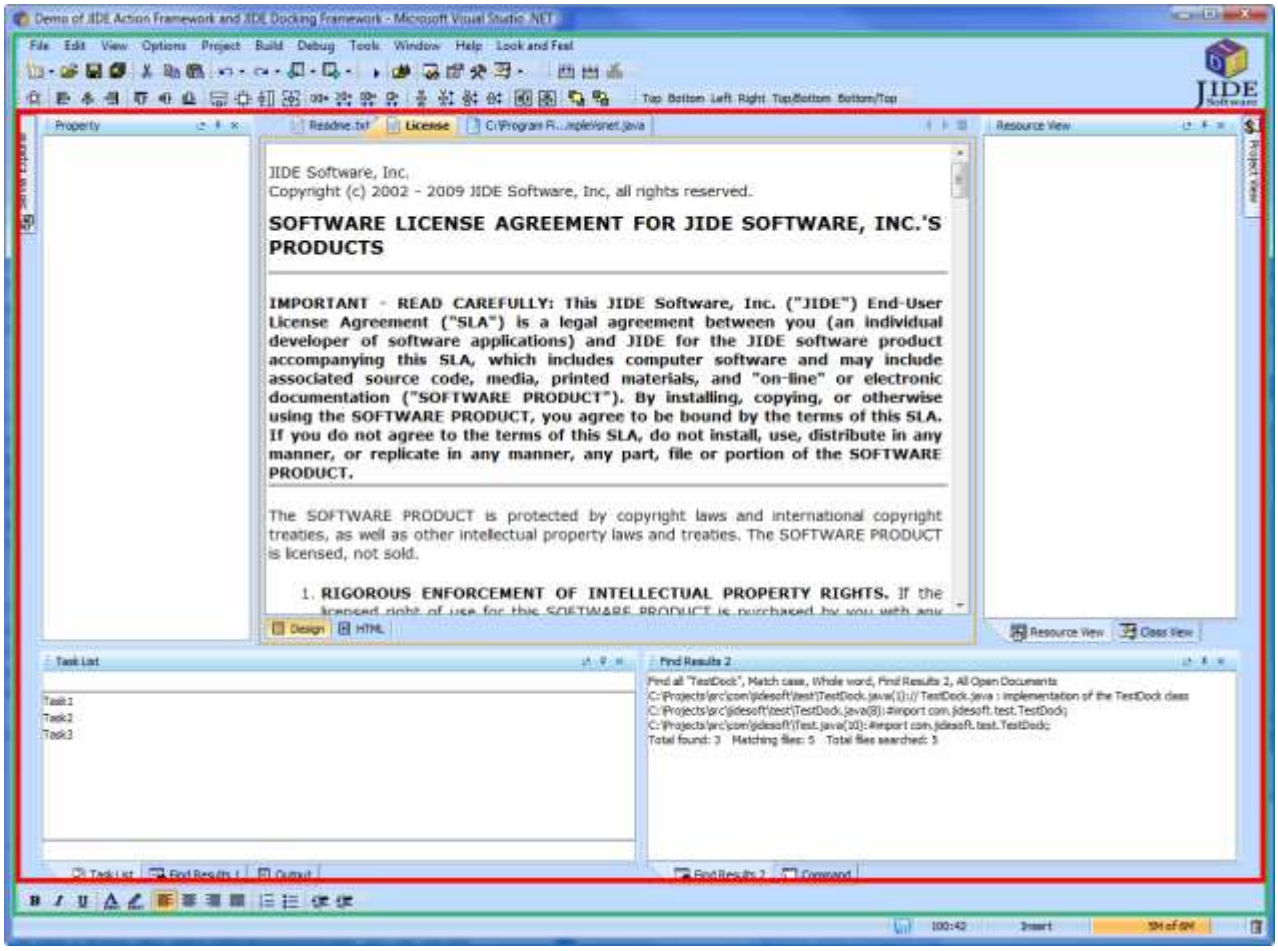


Figure 8 Use both JIDE Docking Framework and JDE Action Framework

If for some reasons, you can't use the three base JFrames we provide, you still need to make sure your *JFrame* has the necessary functions that the base *JFrame* has.

1. Allocate the *contentContainer* for *DockingManager* and *DockableBarManager*.
2. Make sure the *DockingManager* and *DockableBarManager* are initialized.

```
// Create ContentConatiner. You can use JPanel but ContentContainer will give you a nice
// gradient under Office2003 style
Container contentContainer = new ContentContainer();
getContentPane().setLayout(new BorderLayout());
getContentPane().add(contentContainer, BorderLayout.CENTER);
```

```
_dockableBarManager = new DefaultDockableBarManager(this, contentContainer);

_dockableBarManager.getMainContainer().setLayout(new BorderLayout());
Container dockingManagerContentContainer = new JPanel();
dockingManagerContentContainer.setOpaque(false);
_dockableBarManager.getMainContainer().add(dockingManagerContentContainer,
BorderLayout.CENTER);

_dockingManager = new DefaultDockingManager(this, _ dockingManagerContentContainer);
```

3. If you need to use both frameworks, create LayoutPersistenceManager and add both DockingManager and DockableManager to it.

```
_layoutPersistence = new LayoutPersistenceManager();
_layoutPersistence.addLayoutPersistence(getDockableBarManager());
_layoutPersistence.addLayoutPersistence(getDockingManager());
```

4. If you use *DockableBarManager*, make sure you override *getJMenuBar()* method.

```
/**
 * Override in DefaultDockableBarHolder to return the menu bar in DockableBarManager.
 *
 * @return the menubar for this frame
 */
public JMenuBar getJMenuBar() {
   if (getDockableBarManager() != null) {
      Collection col = getDockableBarManager().getAllDockableBars();
      for (Iterator iterator = col.iterator(); iterator.hasNext();) {
         DockableBar bar = (DockableBar) iterator.next();
         if (bar instanceof CommandBar && ((CommandBar) bar).isMenuBar()) {
            return bar;
         }
      }
   }
   return super.getJMenuBar();
```

```
    }
```

Internationalization Support

All Strings used in *JIDE Action Framework* are contained in one properties file called action.properties under com/jidesoft/action. Some users contributed localized version of this file and we put those files inside jide-properties.jar. If you want to support languages other than those we provided, just extract this properties file, translated to the language you want, add the correct postfix and then jar it back into jide-properties jar. You are welcome to send the translated properties file back to us if you want to share it.