JIDE Docking Framework Developer Guide

Contents

Purpose of This Document

Welcome to the *JIDE Docking Framework*, the most advanced framework for developing dockable windows in Swing.

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

What is JIDE Docking Framework

Since AWT/Swing was introduced, many companies have embraced this new technology and made many excellent user-interfaces with it. However, one thing that is obviously missing in most Swing applications is the dockable window. If you have ever used the Visual Studio .NET IDE, you already appreciate the value of dockable windows. Users have come to expect them because they greatly increase the application's ability to display information neatly. Without the nice ability to group information into fixed areas, your application can look cluttered and confusing.

How to Use JIDE Docking Framework

This section is for developers who want to develop applications using *JIDE Docking Framework*.

We developed *JIDE Docking Framework* with the intention of making migration very easy, even if you have already created your application without it. We support all fours types of *RootPaneContainer* (*JFrame*, *JWindow*, *JDialog,* or *JApplet*) as your application's main window.

Understanding the DockingManager

DockingManager is an interface for managing *DockableFrame*s. *DefaultDockingManager*, which implements the *DockingManager* interface, maintains a list of all dockable frames in the application. It also arranges dockable frames in response to the user's mouse and keyboard actions.

The *DefaultDockingManager* constructor takes two parameters:

```
public DefaultDockingManager(RootPaneContainer rootContainer, Container
contentContainer);
```

The *rootContainer* parameter is the main window of your application. It could be *JFrame*, *JWindow*, *JDialog,* or *JApplet*. To make it easy to explain, we will use *JFrame* as an example when describing most of the docking framework features. The *contentContainer* parameter is the Container that you ask *DockingManager* to manage – part of the content pane of *JFrame* in this case. DockingManager will manage only part of *JFrame*'s content pane and will use that part as a placeholder for all your dockable frames. You still have control over the rest of your *JFrame* content pane so that you can add toolbars and a status bar to it, for example. Please note, the

If you are writing your application from scratch, it's probably easier to make your *JFrame* extend *DefaultDockableHolder*. *DefaultDockableHolder* applies a *BorderLayout* to *JFrame*'s content pane and uses the CENTER part as the *contentContainer* that is passed to *DefaultDockingManager*'s constructor.

```
public class MyFrame extends DefaultDockableHolder {
```

```
    ……
}
```

If you have a main Frame class which needs to extend *JFrame* directly, or you don't want to use a BorderLayout for your content pane then your *JFrame* can implement *DockableHolder*. In this case, you need to create your own instance of *DockingManager*.

```
public class MyFrame extends JFrame implements DockableHolder {
    private static DockingManager _dockingManager;
    ……
    MyFrame() {
        ……
        _dockingManager = new DefaultDockingManager(…)
        ……
    }
    ……
}
```

During initialization, *DockingManager* creates a container called *Workspace* that is your application document area (you can call DockingManager#getWorkspace() to access it). You can fill the area that *DockingManager* allocates for your Workspace with either a *JDesktopPane* or *DocumentPane*[1] to manage your application documents. Here is how we do this in our sample code:

```
_documentPane = createDocumentTabs();
_frame.getDockingManager().getWorkspace().setLayout(new BorderLayout
());
_frame.getDockingManager().getWorkspace().add(_documentPane,
BorderLayout.CENTER);
```

[1] DocumentPane is the Tabbed Document Interface (TDI) implementation. Please refer to JIDE Components Developer Guide for detail.

Integration with Existing Applications

Since many of our customers have already built their application before they decide to use our product, we considered integration to be very important when we designed the *JIDE Docking Framework*.

The typical use case of *JIDE Docking Framework* is an application which has some tool windows in addition to the document windows.

What is a tool window? A tool window is a modeless secondary window that has controls a user can use to show the window or hide it. If your application only has one or two tool windows, you may not want to use the Docking Framework, but the more tool windows you have, the more benefit you can get by using the Docking Framework.

It is usually very straightforward to decide whether you need the *JIDE Docking Framework*. Whenever you found yourself using *JSplitPane* within *JSplitPane* or using *JDesktopPane* and *JInternalFrame* in order to arrange all the child windows you have, most likely you can replace them with *JIDE Docking Framework*. For example, we browsed all the screenshots in Swing Sightings and found that about one third of the applications there would be good candidates for using the Docking Framework, not to mention almost all Java IDEs such as NetBeans, JBuilder, and IntelliJ IDEA. Consider Figure 1, which is a screenshot we took from Swing Sightings. The areas that are marked in red can be tool windows, while each 'tab' can be considered a tool window. There are eight tool windows in it. The locations are fixed. User cannot customize it. To make it worse, user cannot even view two tool windows side by side if they belong to the same tabbed pane.

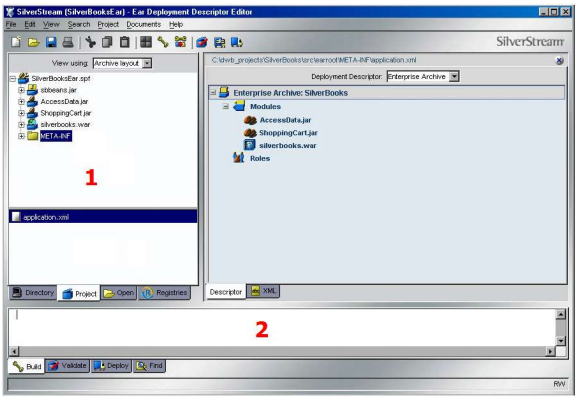


Figure 1 eXtend Workbench – SilverStream (this screenshot is copyrighted by SilverStream)

There are also cases in which some existing dialogs can be turned into tool windows. Most dialogs can be thought of as modal secondary windows. A modal secondary window requires the user to complete interaction with the secondary window and close it before doing anything outside the window. This prevents the user from breaking up actions that the programmer wants to happen together, like picking a filename and saving the file. In some cases this isn't needed, so you should consider converting these dialogs to modeless tool windows.

Once you identify all of the tool window candidates, you need to decide how to divide up your screen. Docking Manager can only manage one area for you, so your menu bars, toolbars,

and status bar should be excluded from that area. In addition, any window that you want visible at all the times should also be excluded from this area. The remaining area should be a *ContentContainer* that you pass as the second parameter of the *DefaultDockingManager*. In the screenshot above, the blue area represents the *ContentContainer*.

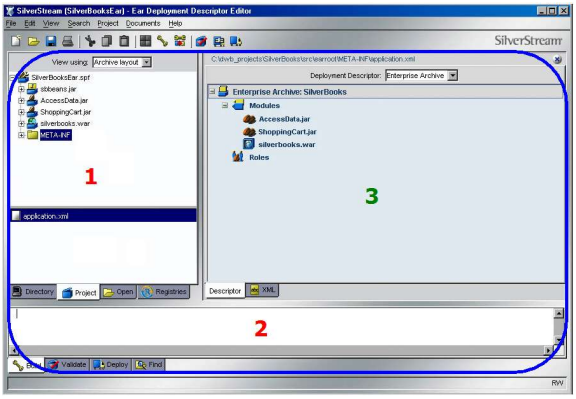


Figure 2 eXtend Workbench – SilverStream (this screenshot is copyrighted by SilverStream)

Once the *DockingManager* is constructed, it's time to add the tool windows. You will need to find the existing code that creates these tool window candidates (such as areas 1 and 2 in the above screenshot). Your Swing components should be contained within a Container or a JPanel, which you can insert into your *DockableFrame*'s content pane. You may also listen for a *DockableFrameEvent* so that you can customize what to do when a child frame is activated, deactivated or hidden etc. During integration, we suggest you only add one or two tool windows at first, and then continue to the next step (you can always add more tool windows later).

When *DockingManager* is constructed, it creates a *Workspace* area for you (area 3 in Figure 2), which is a placeholder for your document windows. You can add any sort of document to the Workspace area. If you prefer the traditional MDI style, for example, you can use *JDesktopPane*. If you like the TDI style, you can use *DocumentPane* in JIDE Components product. Furthermore, if your application already has an equivalent component to show your documents, then you can use it directly in *Workspace*.

Figure 3 provides an example in the form of a screenshot of our sample demo. The outermost frame is the main *Frame* (which could either be a JFrame, JDialog, or JWindow). The area within the red rectangle is the *ContentContainer*. This has a side bar on each of the four sides and a *MainContainer* in the center. The *MainContainer* is shown by the blue rectangle, while the green area is the *Workspace*. The *DockingManager* manages the area that lies between the perimeter of *ContentContainer* and the *Workspace*.

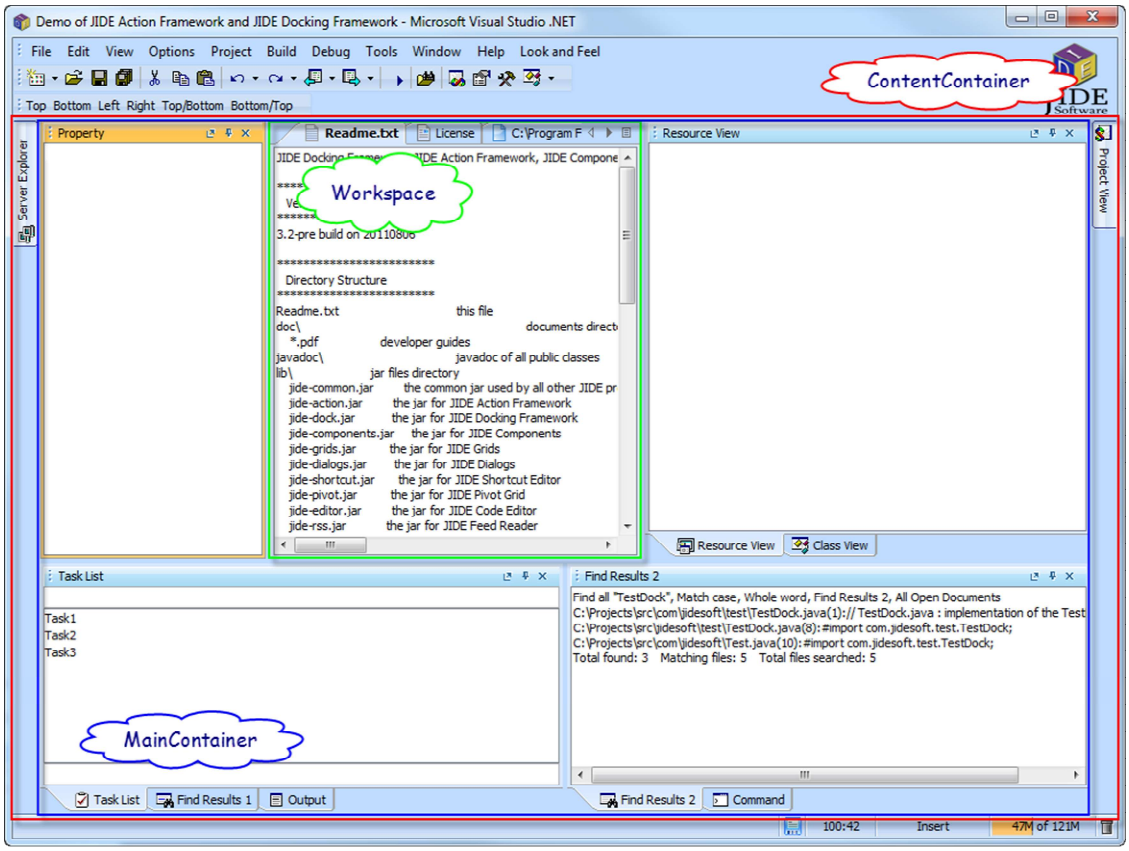


Figure 3 Relations of several panels

Adding DockableFrame

Once the *DockingManager* is set up, the only thing you need to do is to add all your *DockableFrame*s, using code of the form shown below:

```
DockableFrame dockableFrame = new DockableFrame("frameKey",
JideIconsFactory.getImageIcon("Icon for the frame"));

frame.getContext().setInitMode(DockContext.STATE_FRAMEDOCKED);
frame.getContext().setInitSide(DockContext.DOCK_SIDE_SOUTH);

.....

// Initialize DockableFrame such as setting init state and init dock side

// Add components to ContentPane of DockableFrame

....

_frame.getDockingManager().addFrame(dockableFrame);
```

The key of the dockable frame is passed in as a parameter to the constructor. Since we use the key as a key for a hash map internally, it must uniquely identify a dockable frame in this docking manager. If you attempt to add another frame with the same key, the framework will print out an error message and do nothing. Note that since the key is not displayed anywhere on screen, you don't need to localize it. There are three more strings which will be displayed on screen – the title on the title pane, the title on the tab when the frame is tabbed with other dockable frames and the side pane title when the frame is auto-hidden on the side. The title is displayed on the title pane of the dockable frame. The tab title appears in the tab area along the bottom of the panel. The side pane title, as the name indicates, appears on the side pane when the dockable frame is autohidden. You can call setTitle(String) to set the title, setTabTitle(String) to set the tab title, setSideTitle(String) to set the side pane title. If you never call those methods, all titles will be default to the dockable frame key which is not good because titles should be localized and key is not localized. So you should always call at least setTitle(string) to set to a localized string. The tabTitle and sidePaneTitle will default to the title if setTitle(string) is called.

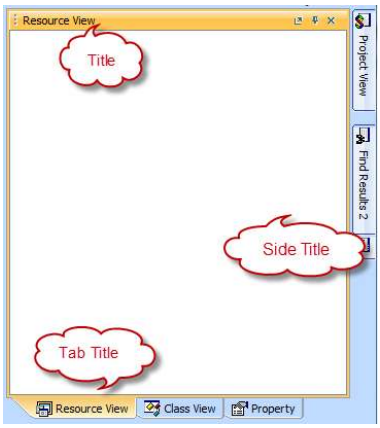


Figure 4 tab title v.s. side title v.s. title of DockableFrame

There are several methods you can use to set the initial default setting: setInitMode(), setInitSide() and setInitIndex(). Dockable windows are placed related to the *Workspace*. For example, if you want to put the dockable window to the south of workspace, you just need to set the init side to DOCK_SIDE_SOUTH. Here are the possible combinations of those values:

initMode	InitSide	initIndex	Comments
STATE_FRAMEDOCKED	DOCK_SIDE_EAST DOCK_SIDE_WEST DOCK_SIDE_NORTH DOCK_SIDE_SOUTH	Any integer greater than 0	Frames with same mode, same side, and same index will form a single tabbed pane.
STATE_AUTOHIDE STATE_AUTOHIDE_SHOWING	DOCK_SIDE_EAST DOCK_SIDE_WEST DOCK_SIDE_NORTH DOCK_SIDE_SOUTH	Any integer greater than 0	Frames with same mode, same side, and same index will form a group on the side bar. AUTOHIDE_SHOWING is treated the same as

			AUTOHIDE mode.
STATE_FLOATING	N/A	Any integer greater than 0	Frames with same mode and same index will form a tabbed pane and lie in the same floating window.
STATE_HIDDEN	N/A	N/A	

The Docking Framework only provides the title bar of your dockable frame, leaving you to manage the ContentPane. You can call dockableFrame.getContentPane() to get the ContentPane and add whatever components to it. To make it easy for you, you can also call dockableFrame.add method directly without calling dockableFrame.getContentPane().add(). Internally we will delegate and add the component to the content pane.

Once you have added all your components to the ContentPane, you can DockingManager.addFrame(dockableFrame) method to add the dockable frames to Docking Manager. At this point, you just tell the Docking Manager to manage this dockable frame for you. Nothing is displayed yet. Once all dockable frames are added to the docking manager and you are ready to display all the dockable frames, you call dockingManager.loadLayoutData() to layout the frames. If there is no previous saved layout data, the Docking Framework calls to resetToDefault() internally to layout the frames based on its initial default settings.

resetToDefault() will layout them according to the initial settings in the table above. By default, it will split the content pane horizontally into three piece, then split the middle pane of the first split pane vertically into three pieces. All dockable frames will then be added to those split panes based on the initial settings, leading to a layout similar to that shown in Figure 3, above. This behaviour can be changed by calling the method *setInitSplitPriority*(), passing in a new split priority. The default value is defined as SPLIT_SOUTH_NORTH_EAST_WEST. What this value means is it will split the south area first, then north area, then east then west. Then the rest is left for workspace area. There are several other values you can use and will get result as the screenshots below.

resetToDefault() will reset every single DockableFrame to its initial state. Below are the methods which will impact the initial state. You could invoke those methods after dockingManager.loadLayoutData() to configure those initial state.

initMode	Methods impact the initial state
STATE_FRAMEDOCKED	dockableFrame.setInitSide(int side) dockableFrame.setInitIndex(int index) dockableFrame.setDockedWidth(int dockedWidth) dockableFrame.setDockedHeight(int dockedHeight)
STATE_AUTOHIDE STATE_AUTOHIDE_SHOWING	dockableFrame.setInitSide(int side)

	dockableFrame.setInitIndex(int index) dockableFrame.setAutohideWidth(int autohideWidth) dockableFrame.setAutohideHeight(int autohideHeight)
STATE_FLOATING	dockableFrame.setUndockedBounds(Rectangle undockedBounds)
STATE_HIDDEN	N/A

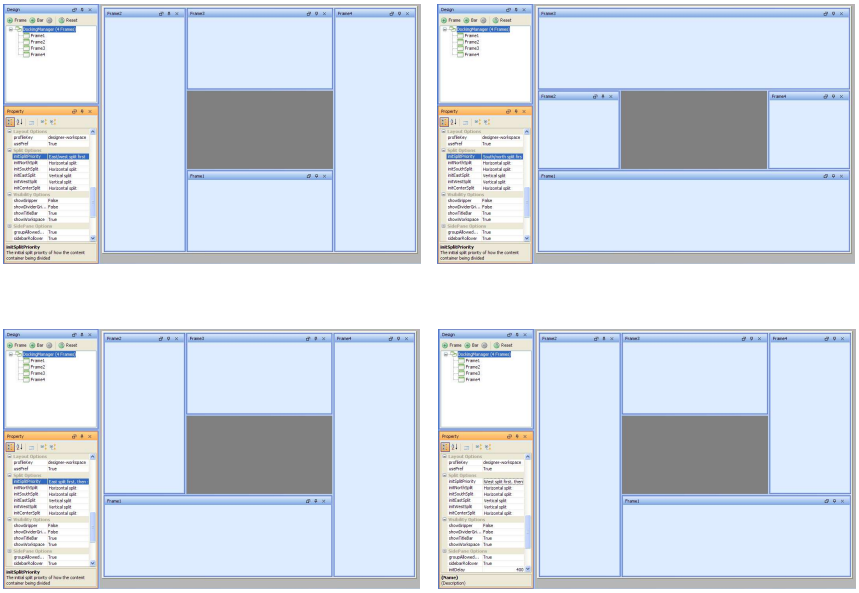


Figure 5 With different initSplitPriority

You can compare the four layouts above and see the difference.

Define Initial Layout using Visual Designer

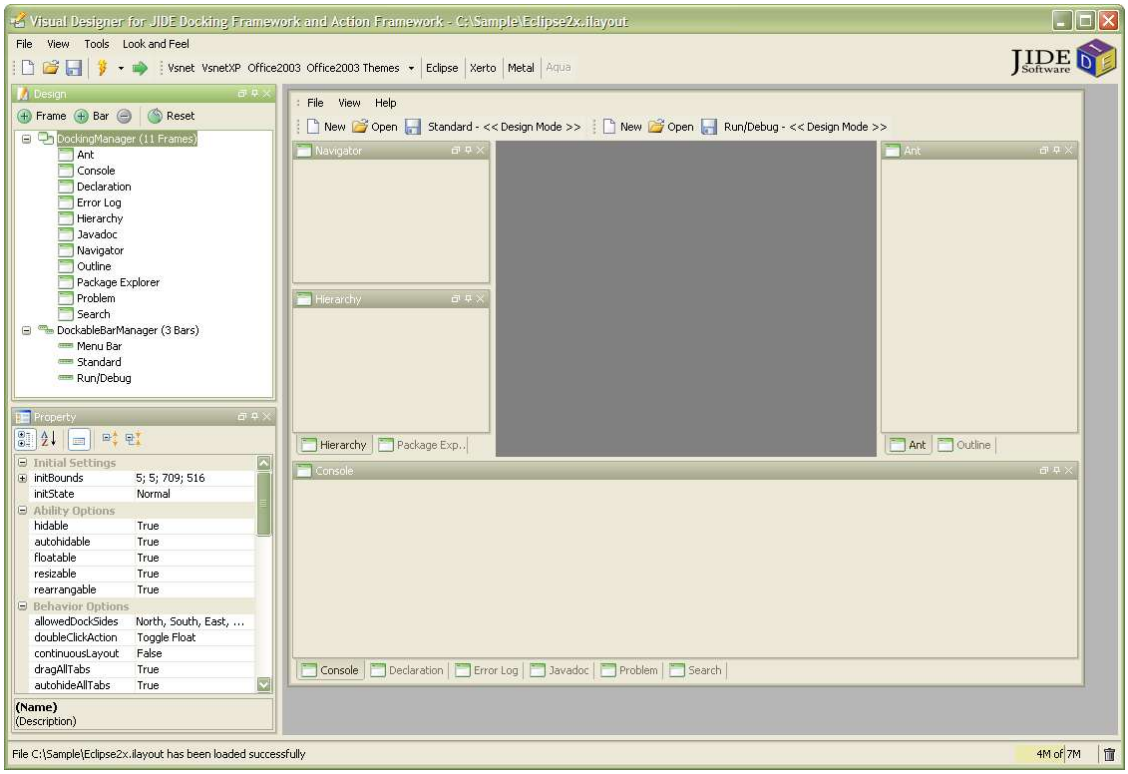
As you can see from the sample code to create a new DockableFrame, the initial layout is actually defined using Java code. See these two lines below.

```
frame.getContext().setInitMode(DockContext.STATE_FRAMEDOCKED);
frame.getContext().setInitSide(DockContext.DOCK_SIDE_SOUTH);
```

There are quite a few other initial layout settings. Some of them are on each DockableFrame; others are on *DockingManager*. If you only have very few DockableFrames, it's not big deal. But if you got more than 10 DockableFrames, it will be a tedious process to define the initial layout and get it correct. Then if you need several initial layouts (i.e. one layout for design mode, one layout for debug mode), it will be even harder. That's why we introduced Visual Designer to make initial layout much easier.

Visual Designer is included in the release. After you run “ant Designer” in example folder to run Visual Designer. Or you can always use the web start version from our website at http://www.jidesoft.com/products/download.htm.

See below for a screenshot of Visual Designer under Windows XP. The power of Visual Designer is it allows you to define adding/removing DockableFrame on fly and use drag-n-drop to rearrange the layout.



Once you design the layout, then save it as .ilayout file. The .ilayout file is just a XML file. We can use what postfix you want. However we recommend you to use .ilayout meaning initial layout to tell the difference from .layout which is normal layout file.

If you will use initial layout, you don’t need to define initial layout of each *DockableFrame* when you create it. See the code below. It’s the same code as the example above but you don’t need to call those two lines anymore.

```
DockableFrame dockableFrame = new DockableFrame(“frameKey”,
JideIconsFactory.getImageIcon("Icon for the frame"));

frame.getContext().setInitMode(DockContext.STATE_FRAMEDOCKED);
frame.getContext().setInitSide(DockContext.DOCK_SIDE_SOUTH);

.....

// Initialize DockableFrame such as setting init state and init dock side

// Add components to ContentPane of DockableFrame
```

```
....
_frame.getDockingManager().addFrame(dockableFrame);
```

After you add all the *DockableFrame*, you call:

```
_frame.getDockingManager().loadInitialLayout(layoutFileName or an
InputStream);
_frame.getDockingManager().loadLayoutData();
```

The first line will load the initial layout from ilayout file created by Visual Designer. The second line will load the last saved layout file. User might change the layout after application started. When application exits, it will save current layout. So loadLayoutData() will load that layout back. If there is no saved layout file, it will use the exactly layout as initial layout.

Visual Designer is also a great place to learn all the customizable options provided by *JIDE Docking Framework*. All the options are displayed in the property table when you select *DockingManager* tree node or *DockableFrame* tree node.

Manipulate DockableFrames

Once the dockable frames have been added to DockingManager, the DockingManager 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, floating, or auto-hidden. In addition, DockingManager also provides support for those cases in which you want to control the frames directly. All these operations are done through the DockingManager. For example, you may want to have a certain window show up when editing a Java file or a MenuItem that collapses all dockable frames to the closest side. Here are some commonly used methods on DockingManager:

activateFrame(): Activate a DockableFrame

activateWorkspace(): Deactivate all DockableFrames and activate the Workspace area

showFrame(): Show a window no matter what state it was in and activate it

hideFrame(): Hide a window no matter what state it was in

autohideAll(): Collapse all windows

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

toggleAutohideState(): Toggle between autohide state and docked state

toggleMaximizeState(): Toggle between maximized state and restored state

dockFrame(String frameKey, int side, int index): dock frame at the specified side and index.

floatFrame(String frameKey, Rectangle bounds, boolean isSingle): float frame at the specified bounds.

maximizeFrame(String name): maximize frame. You can right click on the title bar or tab of any dockable frame and choose "Maximize". A frame can be maximized only when it is in STATE_FRAMEDOCKED.

restoreFrame(): restore the maximized frame if any.

notifyFrame() and denotifyFrame(): notifyFrame() gives a dockable frame visual effect without calling showFrame(). A typical use case is some important information was displayed in a dockable frame, you can use this method to grab user attention. The denotifyFrame() method is opposite to notifyFrame().

Please refer to the DockingManager JavaDoc for more details.

Available or unavailable

Imagining you are developing a HTML/JSP editor, you got two set of dockable frames – one set is for HTML editor and the other is for JSP editor. When a HTML editor is in focus, you want to show the set of dockable frames for HTML editor. When JSP editor is in focus, you want to show the set of dockable frames for JSP editor. Now imagine an HTML editor is in focus. You call showFrame() to show all dockable frames for HTML editor and call hideFrame() to hide all the frames for JSP editor. A user feels one of the dockable frame for HTML editor is not very helpful, so he/she clicks on close button to close that dockable frame. However when he/she switched to JSP editor and switched back to HTML editor, that frame is shown again because you call **showFrame()** to show it. Isn't it annoying?

To solve this problem, we introduce available and unavailable into *JIDE Docking Framework*. It can be made available or unavailable by calling **setFrameAvailable(String name)** and **setFrameUnavailable(String name)** respectively. When a dockable frame is initialized, it's always available.

When **setFrameUnavailable(String name)** is called, if the frame is visible, it will be hidden. Any calls, such as **showFrame()**, **hideFrame()**, **autohideFrame()** etc, will be ignored because the frame is unavailable. Later if you call **setFrameAvailable(String name)**, the frame will be put to the exact state and position when **setFrameUnavailable(String name)** was called. If the frame is hidden when **setFrameUnavailable(String name)** is called, it will still be hidden.

Now, let's revisit the HTML/JSP editor problem. All you need to do is to call **setFrameAvailable()** to all dockable frames when HTML editor got focus and call **setFrameUnavailable()** to those dockable frames when HTML editor lost focus.

DockableFrame Events

We support twelve events that are specific to dockable frame.

- DOCKABLE_FRAME_ADDED: when DockableFrame is added to DockingManager.
- DOCKABLE_FRAME_REMOVED: when DockableFrame is removed from DockingManager.
- DOCKABLE_FRAME_SHOWN: when showFrame is called on the DockableFrame.
- DOCKABLE_FRAME_HIDDEN: when hideFrame is called on the DockableFrame.
- DOCKABLE_FRAME_DOCKED: when DockableFrame changes from other states to DOCKED state.

- DOCKABLE_FRAME_FLOATED: when DockableFrame changes from other states to FLOATED state.
- DOCKABLE_FRAME_AUTOHIDDEN: when DockableFrame changes from other states to AUTOHIDED state.
- DOCKABLE_FRAME_AUTOHIDESHOWING: when DockableFrame changes from other states to AUTOHIDE_SHOWING state.
- DOCKABLE_FRAME_ACTIVATED: when DockableFrame becomes active.
- DOCKABLE_FRAME_DEACTIVATED: when DockableFrame becomes inactive.
- DOCKABLE_FRAME_TABSHOWN: when DockableFrame becomes visible because its tab is selected.
- DOCKABLE_FRAME_TABHIDDEN when DockableFrame becomes invisible because its tab is deselected.
- DOCKABLE_FRAME_MAXIMIZED when DockableFrame is maximized.
- DOCKABLE_FRAME_RESTORED when DockableFrame is restored from maximized mode.

DockableFrameDropListener

You can call *DockingManager#addDockableFrameDropListener* to listen to any dock attempt during drag-n-drop. *DockableFrameDropListener* has one method called

```
boolean isDockingAllowed(DockableFrame source, Component target, int side)
```

When user starts to drag a dockable frame, the source will be that dockable frame. When it is dragged over a component, the target will be the component. However we only use certain components as target. These are *DockableFrame*, *Workspace*, *DockableFrameContainer*, and *ContainerContainerDivider*. Let's say user drags a dockable frame and moves it over the center of *Workspace* area. Behind the scenes, the *isDockingAllowed()* method is called with the target parameter set to the *Workspace* and the side parameter will be set to *DockContext.DOCK_SIDE_CENTER*.

Using this *DockableFrameDropListener*, you will be able to have a very fine control over the dock. You can easily do thing like disallowing dock a dockable frame to another dockable frame.

Workspace Area

From figure 3, you can see a green area which is called workspace. As we mentioned at the beginning of this guide, dockable frames are the secondary window. Workspace is the primary window. Before v1.5 of JIDE Docking Framework, Workspace was not optional. However there are some applications which don't have document concept, nor could find something which is special enough to be put in workspace area. All they need is dockable frames. So in v1.5, we made Workspace area optional.

To hide Workspace, you can call setShowWorkspace(true) method on DockingManager. To show it, call setShowWorkspace(false).

Here is an example of DockingFramework without Workspace area.

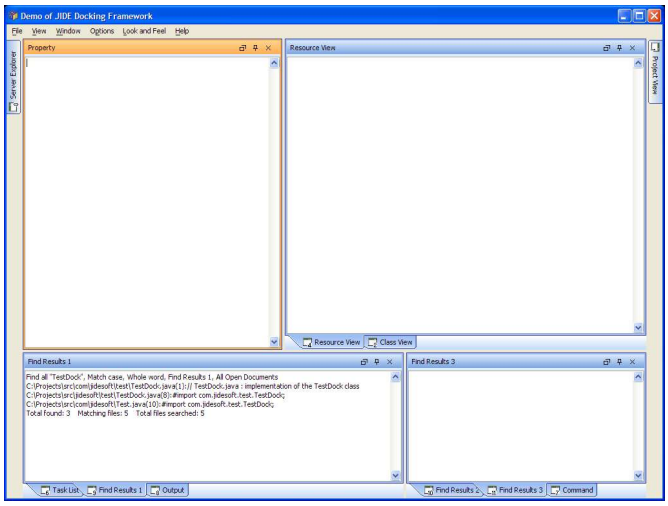


Figure 6 Without Workspace area

In addition, Workspace can also be a place holder for dockable frame. You can change this option by calling *workspace.setAcceptDockableFrame(true/false)*. To put a dockable frame into Workspace area, all you need to do is *setInitSide()* to *DockContext.DOCK_SIDE_CENTER* during initialization. User can also drag-n-drop any dockable frame into Workspace, just like drag-n-drop in other tabbed pane. By default Workspace can accept DockableFrame. However once you call add() method directly on Workspace area, we assume you want to use Workspace area for other purposes and we will automatically sets the *acceptDockableFrame* attribute to false.

Persisting Layout Information

JIDE Docking Framework offers the ability to save windows information and settings between sessions, using the java.util.prefs 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 dockable frame positions as well as the main window's size and location. 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(byte[] layoutData)** are methods allowing you get the layout data as a byte[], in case you want to load/save it without using java.util.prefs.

If you prefer that *JIDE Docking 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 the *JIDE Docking Framework* use its default file location. By default it uses java.util.prefs 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 java.util.prefs. 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. The **isLayoutAvailable(String layoutName)** method will tell you if a layout is available. The **getAvailableLayouts()** method will return you a list of layout names. The **removeLayout(String layoutName)** will remove the saved layout.

JIDE supports two layout formats. One is traditional binary format while another is XML format. You could use the setXmlFormat(boolean xmlFormat) method to configure the target layout format.

If you choose use traditional binary format, 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 **addFrame()** or **removeFrame()**. 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)**.

As you may know, **loadLayoutData()** or **resetToDefault()** will make the main JFrame visible. Sometimes, you prefer to make the frame visible later so that you get a chance to initialize other components in your application. In this case, you can call **dockingManager**.**setShowInitial(false)**

before **loadLayoutData()** is called. After everything is initialized and you are ready to display the main JFrame, call **dockingManager**.**showInitial()**.

Perspectives

JIDE Docking Framework supports the perspective[2] concept. Here is the step in order to fully support perspective in your application.

1. You need to use Visual Designer to define initial layouts for each perspective. Make sure you use the same dockable frame key for all dockable frames if they are shared by different perspective. Then save each initial layout as .ilayout file. Those layouts will be the default layout for each perspective. You should use the perspective name as the name for .ilayout file to avoid confusion. For example, if you have a "debug" perspective, name the .ilayout file as "debug.ilayout".

2. You still add all dockable frames to DockingManager as before, no matter if a dockable frame appears in any of the perspectives. You can skip call to setInitIndex, setInitMode or setInitSide methods as .ilayout file will take care of those.

3. In your code, if you want to switch to a perspective, you call:

   a) DockingManager#loadInitialLayout to load the .ilayout file.

   b) Call DockingManager#loadLayoutDataFrom(perspectiveName) to load from a named layout. You should use the perspective name as the name for the layout to avoid confusion. For example, if you have a "debug" perspective, call loadLayoutDataFrom("debug"). Please note, this call will load the user setting of the perspective as user might drag-n-drop to rearrange the layout already.

   c) (Optional) You may want to disable user access to certain dockable frames in certain perspective. If so, call DockingManager#setFrameUnavailable(frameKey) to disable those dockable frames. Make sure you call it after the **loadLayoutDataFrom()** method is called because this call will make all frames available.

4. If you want to switch from a perspective, you call

   a) saveLayoutDataAs(perspectiveName) to save the layout in case user rearranged it. This step will guarantee the step 3.b) above loading the layout successfully.

[2] We refer to the Perspective concept in Eclipse.

5. If you want to reset a perspective to the default layout as defined in the .ilayout file, you call:

   a) DockingManager#loadInitialLayout to load the .ilayout file.

   b) Call DockingManager#resetToDefault() to reset the layout according the information in .ilayout file.

Please note, **saveLayoutDataAs()** and **loadLayoutDataFrom()** methods will use java.util.prefs package to save the layout. If you want to save the layout as file or as stream, you can use **loadLayoutDataFromFile(String filename)** / **saveLayoutDataToFile(String filename)** or **loadLayoutFrom(InputStream in)** / **saveLayoutTo(OutputStream out)** to do it. As long as you use those methods in pair when saving and loading the layout, you will be fine.

Undo and Redo

Dockable window, especially drag-n-drop dockable window, is an advanced UI feature. While technically oriented users can find all features after exploitation, it's hard for non-technical user. When users try to use an application using *JIDE Docking Framework* for the first time, they might make mistakes. After a while, the layout will probably be messed up as they don't know exactly how to return to the old state. This is where the call to the **resetToDefault()** method will bring the layout back to the initial state. However there is no way to bring it back to a state in the middle. As a result, we introduced the undo/redo functionality to the *JIDE Docking Framework* to address this issue.

By default, the undo/redo function is turned off. To turn it on, call **setUndoLimit(int)** and pass in a non-zero value. The larger the number, the more memory will be used. As in the case of 20 dockable frames, each undo/redo will take around 10K memory – just to give you an idea. We suggest setting undo limit to 10.

The undo/redo history is not persisted. So after you close the application, the undo/redo history is gone. If you ever want to clear the history during the same session, you can call **discardAllUndoEdits()**.

undo() will undo the last operation. Those operations include dragging a dockable frame, double clicking on the title bar of dockable frame or tab, hiding a dockable frame, autohiding a dockable frame, floating a dockable frame etc.

redo() will redo the last undone operation.

The undo/redo feature is built on top of Swing's *UndoManager*. If you need to get advanced feature provided by *UndoManger*, you can access the *UndoManager* directly by calling **getUndoManager()**.

There are cases you need to know when an operation is happened so that you can update the menu items to indicate the correct undo/redo state. You can use UndoableEditListener to make it possible (see below).

```
    _frame.getDockingManager().addUndoableEditListener(new
UndoableEditListener(){
        public void undoableEditHappened(UndoableEditEvent e) {
          refreshUndoRedoMenuItems();
        }
    });
```

In the refreshUndoRedoMenuItems, all you need to do is to set the correct state and name to undo/redo menu items. See below.

```
_undoMenuItem.setEnabled(_frame.getDockingManager().getUndoManager().canUndo());
_undoMenuItem.setText(_frame.getDockingManager().getUndoManager().getUndoPresentationName());
_redoMenuItem.setEnabled(_frame.getDockingManager().getUndoManager().canRedo());
_redoMenuItem.setText(_frame.getDockingManager().getUndoManager().getRedoPresentationName());
```

Options

DockingManager has a few options that you can tweak to change its behaviors.

Floatable: This indicates whether the dockable frame(s) can be undocked. If you call **dockingManager.setFloatable(true/false),** all dockable frames will become floatable (or not floatable). DockableFrame also has a method **setFloatable(boolean)**. It will make that dockable frame floatable (or not floatable).

Autohidable: This indicates whether the dockable frame can be automatically hidden against the side of its JFrame. If you call **dockingManager.setAutohidable(true/false)**, all dockable frames will become floatable (or not floatable). DockableFrame also has a method called **setAutohidable(boolean)**. It will make that dockable frame autohidable (or not autohidable).

Hidable: This indicates whether the dockable frames can be closed (or hidden). If you call **dockingManager.setHidable(true/false)**, all dockable frames will closable (or not closable). DockableFrame also has a method called **setHidable(boolean)**. It will make that dockable frame closable (or not closable).

Dockable: This indicates whether the dockable frames can be docked or not. Unlike the previous three options, this is not a global option but an option on each DockableFrame. This means that you have to call **setDockable()** on each DockableFrame to change its behavior. Since one of the main features provided by *JIDE Docking Framework* is the dockable window, it doesn't make sense for you to make all dockable windows not dockable! However if you want to set this attribute for all dockable frames, you need to obtain a list of the dockable frames from the DockingManager and then call **setDockable()** on each DockableFrame (we don't have a convenient method on DockingManager that does it for you).

Rearrangable: This indicates whether the dockable frames can be arranged by users. There are some cases where developers want to layout the dockable frames manually, save this layout, and ship the product to their end users. Once it reaches the end users, they don't want users to change the position of the dockable frames. In this case, just call **setRearrangable(false)** when releasing the product. User still can autohide frames or resize frames etc. However, they can't change the state of the frames or move them around.

Resizable: This indicates whether the dockable frames can be resized by users. If this option is false, none of the dockable frames can be resized, including docked mode, floating mode or autohide mode. Usually you can combine this option with Rearrangable option so that you can rearrange/resize dockable frames freely during development and ship an optimized but fixed layout to users.

ContinuousLayout: This indicates whether the components continuously redraw themselves as the user resizes the split pane (the default is false). Call **setContinuousLayout(true/false)** to change this behavior.

SensitiveAreaSize: When a dockable frame is dragged near the border of a target frame, the outline changes to indicate what the dragged frame will look like if it is 'snapped' into the target frame. The outline is drawn around the frame's contents rather than the frame itself because the frame itself merges into the container walls. This integer value is used to specify how wide the docked frame's border is (by default, its 20 pixels). You can call **setSensitiveAreaSize(int)** to set to a new value.

Available Buttons: DockableFrame can have buttons on the title bar. Although by default, it will have three buttons – Float/Dock, Autohide and Close, you can choose what buttons are visible by calling **dockableFrame.setAvailableButtons(int buttons).** The "buttons" parameter is a bitwise OR of the following values defined in DockableFrame: BUTTON_CLOSE, BUTTON_AUTOHIDE, BUTTON_HIDE_AUTOHIDE, BUTTON_FLOATING and BUTTON_MAXIMIZE. For example, if you want to be compatible with the earlier version of JIDE Docking Framework which didn't show the 'Floating' button, you can do the following:

```
dockableFrame.setAvailableButtons(DockableFrame.BUTTON_CLOSE |
DockableFrame.BUTTON_AUTOHIDE);
```

OutlineMode: When a dockable frame is dragged, an outline of the frame is painted. In early versions of JIDE Docking Framework, only partial outlines were painted if the outlines

extended beyond the main JFrame. In version 1.2.1 of JIDE Docking Framework, we added this option to paint the full outline instead. If OutlineMode is FULL_OUTLINE_MODE, the full outline will be painted even if it extends beyond the main JFrame; if it is PARTIAL_OUTLINE_MODE, the outline will be clipped. In order to avoid changing the behavior of current installations, we set the default OutlineMode to PARTIAL_OUTLINE_MODE. Please note, if you use FULL_OUTLINE_MODE, there will be flickering when you drag the outline under JDK5. On JDK6, because of the true double buffer bug fix, there is no noticeable flickering.

If you are using JDK6u10 and above, there are two more modes you can use. They are HW_TRANSPARENT_OUTLINE_MODE and HW_OUTLINE_MODE. Both leverage the translucent window feature introduced in JDK6u10.

GroupAllowedOnSidePane: This determines if the group is allowed on the SidePane. By default this option is *true*, which means that when you autohide a tabbed pane with several dockable frames on it, all those dockable frames will become one group on the SidePane. However if this option is set *false*, the each dockable frame will become one group. Note that we don't support changing this option on the fly, so if you want change it; you must do so during the initialization stage.

EasyTabDock: This is an option to make the tab-docking of a dockable frame easier. The previous approach requires the user to drag a dockable frame and point to the title bar of another dockable frame in order to tab-dock with it. However if this option set on, then pointing to the middle portion of any dockable frame will tab-dock with that frame (the default is off). Note that if you turn this option on, you should make sure that you warn your users to press the CTRL key during dragging, to prevent it from being docked. If you do not do this then your users will probably feel frustrated when they try to float a dockable frame but find that it always docks!

TabDockAllowed: This is an option to allow/disallow tab dock. If false, the whole tab dock will be disabled, which means you will never see a tabbed pane used in the whole Docking Framework.

SideDockAllowed: This is an option to allow/disallow side dock. Side dock is the only way to create a new tabbed pane. If false, the whole side dock will be disabled, which means user will never be able to create a new tabbed pane area except those already existed as part of initial layout.

Allow Nested Floating Window: This is an option to allow nested windows when in floating mode. *JIDE Docking Framework* can allow you to have as many nested windows in one floating container as you want. However, not all your users want to have that complexity. Therefore we leave this as an option which you can choose to turn on or leave off (the default). In our opinion, it's not very useful to have nested floating windows. However, you can turn this on if your users are very advanced and your application needs to have nested floating windows.

Show Gripper: This is an option to give users a visual hint that the dockable frame can be dragged. To make this obvious to the user, we added an option so that a 'gripper' is painted on the title bar of those dockable frames which can be dragged. However, since the grippers can make the screen looks busy, if you have a lot of dockable frames, we suggest you turn this option off (the default). If you only have a few dockable frames, we suggest you turn it on.

Show/Hide TitleBar: This is an option to hide the title bar of dockable frame. Title bars have two functions in Docking Framework. First, it provides a consistent way to name each dockable frame. User can look at the title bar to find out what the dockable frame is for. If the content pane of each dockable frame said clearly what it is for, you don't really care this function. Secondly, it provides a place for mouse to drag. Please note, the tab of each dockable frame has a similar function. The difference is dragging the title bar will drag all dockable frames in the same tabbed pane vs. dragging the tab only drags one dockable frame. Also when there is only one tab in tabbed pane, the tab will be hidden. User can only drag the title bar. You can choose to hide title bar. However if you still want to keep the drag-n-drop feature, you should call setHideOneTab(false) using TabbedPaneCustomizer.

DoubleClickAction: This is an option to define what will happen after user double clicks on title bar of dockable frame. By default, the value is DOUBLE_CLICK_TO_FLOAT which means double click will toggle between floating state and docked state. You can set it to DOUBLE_CLICK_TO_MAXIMIZE so that double click will maximize dockable frame.

TabbedPane options: By default dockable frames are put into a tabbed pane in which tabPlacement is set to 'top'. If you want a different behavior, you can call **setTabbedPaneCustomizer(customizer)**, as shown below:

```
DefaultDockingManager _dockingManager = // init here
dockingManager.setTabbedPaneCustomer(new
DefaultDockingManager.TabbedPaneCustomizer() {
    public void customize(JideTabbedPane tabbedPane) {
        tabbedPane.setTabPlacement(SwingConstants.TOP); // put tab on top
        tabbedPane.setShrinkTabs(false); // don't shrink tab
        tabbedPane.setBoxStyleTabs(true); // use box style
        ……
    }
});
```

Popup Menu: When user right clicks on the title bar or tab of dockable frame, a context menu pops up. We provided a default menu, which allows you to float, hide, or auto-hide the frame. You can call **setPopupMenuCustomizer()** to modify this menu and create your own popup menu choices. The popupMenu, the dockingManager, and the dockableFrame parameters are all pretty much self-explanatory, with the caveat that the dockableFrame parameter refers to the Frame that is currently visible. The onTab parameter allows you to have separate menus for the tabs themselves and the title bar. A special case is you don't want to have any popup menu for a dockable frame. In this case, just call popupMenu.removeAll(). Since there are no items in the popup menu, the menu will not be shown.

```
public interface PopupMenuCustomizer {
```

```
    void customize(JPopupMenu popupMenu,
                   DefaultDockingManager dockingManager,
                   DockableFrame dockableFrame,
                   boolean onTab);
}
```

Title Bar Component: You can add any component to the title bar. A typical use case is to add a toolbar. Note that we handle the position of the title bar component differently, depending on the Look and Feels. When the dockable frame is wide enough, we will insert the title bar component between the title text and the three default buttons. If this is not the case then the title bar component will be put just below the title bar (as is the default in the EclipseLookAndFeel). In VsnetLookAndFeel, since the title bar of the dockable frame is very thin, we always put the title bar component below the title bar (it doesn't look good if the title bar component is at the same line as title bar). However you can modify this behaviour by changing the UIDefaults of "DockableFrameTitlePane.titleBarComponent". This is a *boolean*, where *true* means they can be on the same line if width permits, and *false* means they must always be on different lines.

See below for example source code to add a JToolBar to the dockable frame as title bar components.

```
JToolBar toolBar = new JToolBar();
toolBar.add(createTitleBarButton(…));
//…. Add whatever you want to the toolbar
toolBar.setFloatable(false);
toolBar.setRollover(true);
dockableFrame.setTitleBarComponent(toolBar);
```

Customize TabbedPane: DockingManager heavily uses JideTabbedPane when dockable frames are docked together as tabs. To customize the tabbed pane, you can call DockingManager's **setTabbedPaneCustomizer()** method. For example, to move tabbed pane's tabs from bottom to top, you can call

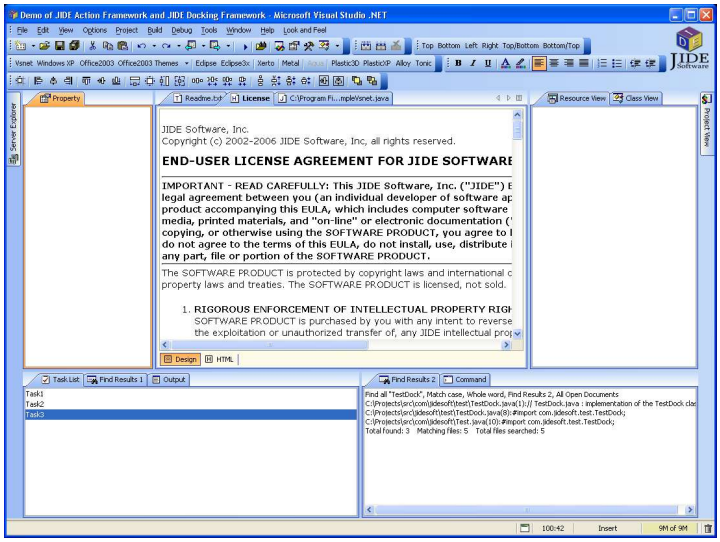
```
dockingManager.setTabbedPaneCustomizer(new
DockingManager.TabbedPaneCustomizer() {
  public void customize(JideTabbedPane tabbedPane) {
    tabbedPane.setTabPlacement(JideTabbedPane.TOP);
  }
```

```
});
```

In fact, you can also combine with other options to hide the title bars and use tabs to do all the drag-n-drop operations.

```
dockingManager.setTabbedPaneCustomizer(new
DockingManager.TabbedPaneCustomizer() {
   public void customize(JideTabbedPane tabbedPane) {
      tabbedPane.setTabPlacement(JideTabbedPane.TOP);
      tabbedPane.setHideOneTab(false);
   }
});
dockingManager.setShowTitleBar(true);
dockingManager.setEasyTabDock(true);
```

Here is the result. As you can see, all tabs are on top now. Dockable frame doesn't have title pane which saves more vertical spaces for you. However users can still drag-n-drop to rearrange the dockable frames by dragging the tab. Or right click to show the popup menu to do hide/autohide/float/maximize the dockable frames although the title pane buttons are not there anymore.



Heavy weight component support: JIDE Docking Framework is a light weight Swing component. In real application, you might need to use heavyweight component in order to support native widget such as JDIC web browser or even ActiveX or 3D canvas such as JOGL. There will be some conflicts as heavyweight component tends to obscure lightweight

component in certain condition. JIDE Docking Framework solves those issues. All you need to do is to use ***DockingManager#setHeavyweightComponentEnabled(true)***.

Additional Methods

DockingManager has several additional methods that may be useful:

getAllFrames(): Gets a collection of all the keys of dockable frames.

getFrame(String name): If you know the key of the dockable frame (in most cases you do), you can call this method to get the actual dockable frame.

getActiveFrameKey(): If there is one active frame in the DockingManager, this method will give you that frame (it returns null if no frame is active).

updateComponentTreeUI(): this method is for switching the Look and Feel, without restarting. This method will call **SwingUtilities.updateComponentTreeUI ()** on all top level containers it knows about.

removeAllFrames(): this method will remove all frames from DockingManager. You can call this method before closing the main JFrame to get DOCKABLE_FRAME_REMOVED events sent to all your registered listeners, so that you can do clean-up, for example. However, make sure you only call it after you have saved the layout data.

setEscapeKeyTargetComponent(Component): If this is never called, workspace will be the escape key target component. When ESC key is pressed in a dockable frame, the dockable frame will lost focus and this component will get focus.

Multiple DockingManagers

Most applications only need one *DockingManager* with multiple *DockableFrames*. However JIDE Docking Framework supports multiple *DockingManagers* in the same application. They could be in the same *JFrame* or in different *JFrames*. They could be two independent *DockingManager* or one *DockingManager* is inside a *DockableFrame* from another *DockingManager*.

Active Frame

A *DockingManager* can have at most one active *DockableFrame*. If you don't do additional coding and have several *DockingManagers* in one application, each *DockingManager* can have its own active frame. This is OK in some cases, for example, when two DockingManagers manage two panels that are independent in the application. However in other cases, user might want only one frame can be active in several DockingManagers. For example, there are menu items that should apply to the active frame. If there are two active frames, user will be confused about which frame the menu items will act on. In order to support this, we introduced *DockingManagerGroup* class. *DockingManagerGroup* works just like *ButtonGroup*. You just add *DockingManager* to *DockingManagerGroup*. *DockingManagerGroup* will make sure one frame is active at a time. You can run *SideBySideDockingFrameworkDemo* under examples\D2. TwoDockingFramework to find out how to use it.

Dragging DockableFrame Across DockingManagers

JIDE Docking Framework supports dragging dockable frame from one *DockingManager* to another *DockingManager*. By default, this feature is disabled. However if you designed your application to have several *DockingManager*s and you want to enable this feature, you can call

```
dockingManager.setCrossDraggingAllowed(true);
dockingManager.setCrossDroppingAllowed(true);
```

The first line enables dragging a dockable frame out of the *DockingManager*. The second line enables dropping a dockable frame into the *DockingMangaer*. You can run two demos[3] under examples\D2.TwoDockingFramework to find out how it behaves.

Look And Feel

To support the dockable windows feature in Docking Framework, we created three new components that are not provided as standard Swing Components: DockableFrame, SidePane and JideTabbedPane. Since all three have their own ComponentUI, if you want to use your own

[3] The two demos are TwoFramesDockingFrameworkDemo.java and SideBySideDockingFrameworkDemo.java.

LookAndFeel, you just need to create an appropriate ComponentUI and add the mapping to UIClassmap of UIDefaults. Alternatively, if only some minor modifications are needed, you can simply modify some values of the Component Defaults, as shown below:

DockableFrame UIDefaults

Name	Type	Description
DockableFrame.background	Color	Background
DockableFrame.border	Border	Border
DockableFrame.slidingEastBorder	Border	The border when frame is sliding from east side
DockableFrame.slidingWestBorder	Border	The border when frame is sliding from west side
DockableFrame.slidingSouthBorder	Border	The border when frame is sliding from south side
DockableFrame.slidingNorthBorder	Border	The border when frame is sliding from north side
DockableFrame.activeTitleBackground	Color	Active title bar background color
DockableFrame.activeTitleForeground	Color	Active title bar foreground color
DockableFrame.inactiveTitleBackground	Color	Inactive title bar background color
DockableFrame.inactiveTitleForeground	Color	Inactive title bar foreground color
DockableFrame.titleBorder	Border	The Border of title
DockableFrame.activeTitleBorderColor	Color	Active title bar border color
DockableFrame.inactiveTitleBorderColor	Color	Inactive title bar border color
DockableFrameTitlePane.font	Font	Font used by title bar
DockableFrameTitlePane.titleBarComponent	Boolean	If the title bar component will be put the same line as title bar if the is enough space. It's true only under Eclipse L&F.
DockableFrameTitlePane.alwaysShowAllButtons	Boolean	If all title bars are visible no matter what. Usually when a button is not used or not applicable, it's hidden.
DockableFrameTitlePane.buttonsAlignment	Integer	The alignment of buttons. It could be either TRAILING or LEADNING.
DockableFrameTitlePane.titleAlignment	Integer	The alignment of title bar text. It could be either TRAILING, LEADNING or CENTER.
DockableFrameTitlePane.buttonGap	Integer	The gap in pixels between buttons
DockableFrameTitlePane.showIcon	Boolean	If the dockable frame icon is shown on the title bar. Under Vsnet L&F, it's false. Under Eclipse L&F, it's true.
DockableFrameTitlePane.margin	Insets	The margin of title bar.
DockableFrameTitlePane.hideIcon	Icon	The icon used on the button to hide the dockable frame
DockableFrameTitlePane.unfloatIcon	Icon	The icon used on the button to dock the dockable

		frame from floating mode
DockableFrameTitlePane.floatIcon	Icon	The icon used on the button to float the dockable frame
DockableFrameTitlePane.autohideIcon	Icon	The icon used on the button to auto-hide the dockable frame
DockableFrameTitlePane.stopAutohideIcon	Icon	The icon used on the button to pin the dockable frame from auto-hidden mode to docked mode
DockableFrameTitlePane.hideAutohideIcon	Icon	The icon used on the button to hide the showing auto-hide the dockable frame
DockableFrameTitlePane.maximizeIcon	Icon	The icon used on the button to maximize the dockable frame
DockableFrameTitlePane.restoreIcon	Icon	The icon used on the button to restore the dockable frame from maximized mode.

SidePane UIDefaults

Name	Type	Description
SidePane.margin	Insets	Margin of SidePane. Only top and left is used.
SidePane.iconTextGap	Integer	Gap between icon and text
SidePane.textBorderGap	Integer	The distance between end of the longest title and the border of the button
SidePane.itemGap	Integer	Gap between two buttons
SidePane.groupGap	Integer	Gap between two button groups
SidePane.foreground	Color	Foreground
SidePane.background	Color	Background
SidePane.lineColor	Color	Line color of each button
SidePane.buttonBackground	Color	Button Background
SidePane.font	Font	Font used by SidePane
SidePane.showSelectedTabText	Boolean	If the value is true, it means the text of selected item will be visible. By default, it is true under VSNET style or false under Eclipse style.
SidePane.alwaysShowTabText	Boolean	If the value is true, it means the text of all items will be visible. By default, it is false under all styles.
SidePane.orientation	Integer	0 means the labels, when displaying vertically, are rotated 90 degree clock-wise. This is the way that Vsnet chose. 1 means for east side, its rotated 90 degree clockwise and for west side, its 90 degree counter-clockwise. This is the way IntelliJ IDEA chose.

JideTabbedPane UIDefaults

Name	Type	Description
JideTabbedPane.background	Color	Background
JideTabbedPane.foreground	Color	Foreground
JideTabbedPane.light	Color	Color to draw light area of tabs
JideTabbedPane.highlight	Color	Color to draw highlight area of tabs
JideTabbedPane.shadow	Color	Color to draw shadow area of tabs
JideTabbedPane.darkShadow	Color	Color to draw dark shadow area of tabs
JideTabbedPane.tabInsets	Insets	Insets of tab
JideTabbedPane.contentBorderInsets	Insets	Insets of tab content pane
JideTabbedPane.tabAreaInsets	Insets	Insets of the tab area
JideTabbedPane.tabAreaBackground	Color	Tb area background
JideTabbedPane.font	Font	Font
JideTabbedPane.unselectedTabTextForeground	Color	Text color of unselected tab
JideTabbedPane.selectedTabBackground	Color	Selected tab background
JideTabbedPane.textIconGap	Integer	Gap between icon and text, in pixels
JideTabbedPane.showIconOnTab	Boolean	A Boolean flag. If it's true, the tabs will show the icon. By default, it is true under VSNET, OFFICE2003 and OFFICE2007 style and false under Eclipse style.
JideTabbedPane.showCloseButtonOnTab	Boolean	A Boolean flag. If it's true, the close icon will be visible on each tab. If it's false, the close icon will be visible along with the left and right scroll button on the right side. By default, it is false under VSNET, OFFICE2003 and OFFICE2007 style and true under Eclipse style.
JideTabbedPane.closeButtonAlignment	Integer	The valid values are SwingConstants.TRAILING or SwingConstants.LEADING

Miscellaneous UIDefaults

Name	Type	Description
Contour.color	Color	The contour outline color
Contour.thickness	Integer	The contour thichness.
ContentContainer.background	Color	Background color of the container. Please refer to figure 3 to see what content container area is .
Workspace.background	Color	Background color for workspace. Again, you can refer to figure 3 to see what workspace area is.
DockingFramework.changeCursor	Boolean	A flag for cursor shape change while dragging. By default it's true under Eclipse style and false under all

		others.

How to use other LookAndFeel

JIDE Docking Framework can work with any existing LookAndFeels, either those that come with JDK, or third-party LookAndFeels. However, as you can see the UIDefault tables from previous section, you must somehow insert additional UIDefault values into LookAndFeel UIDefault table for *JIDE Docking Framework* to work correctly.

The easiest way to do so is to call ***LookAndFeelFactory.installJideExtension().*** This call will add necessary UIDefaults not only for *JIDE Docking Framework* but also for all other JIDE products (see below):

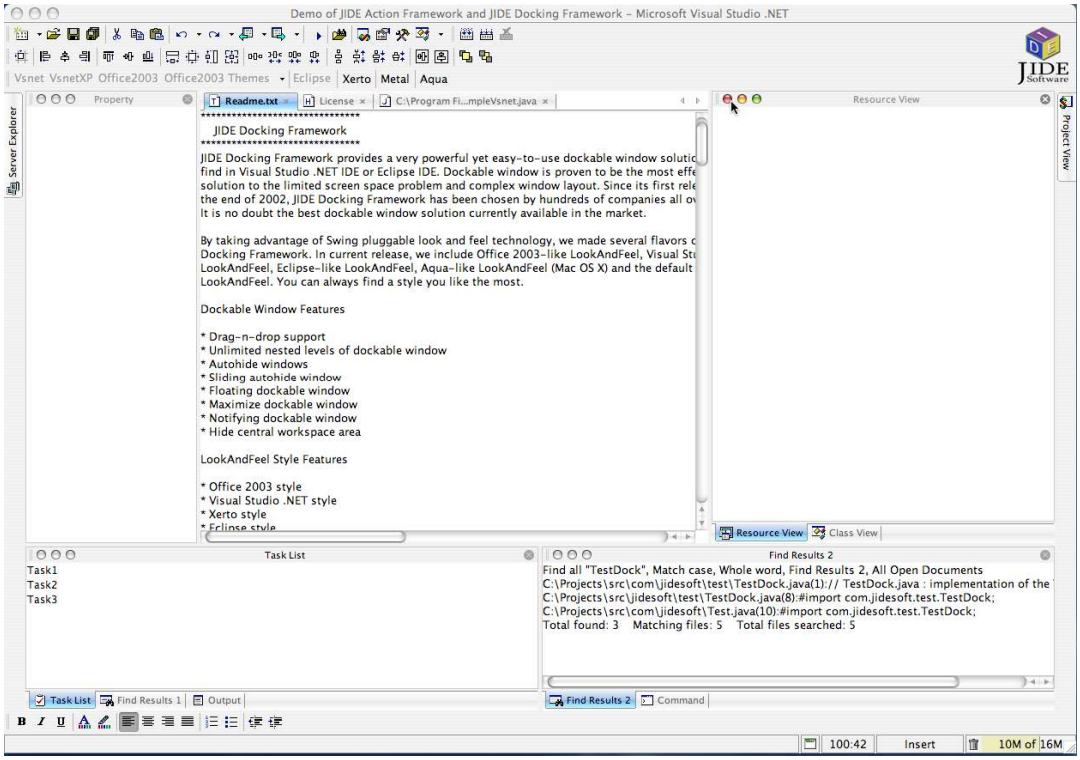
```
UIManager.setLookAndFeel("<whatever L&F>");
LookAndFeelFactory.installJideExtension();
```

If your application only use one L&F, you just need to call ***installJideExtension()*** once when application starts. However if you allow users to change L&F on fly, you need to make sure every time you call ***UIManager.setLookAndFeel()*** to change L&F, you also call ***installJideExtension()*** immediately. The ***installJideExtension()*** method also has an overload method which takes an int parameter. You can pass in a style to this method such as LookAndFeelFactory.VSNET_STYLE, OFFICE2003_STYLE, OFFICE2007_STYLE, XERTO_STYLE, ECLIPSE_STYLE or ECLIPSE3X_STYLE. Those styles will determine what styles you want to use on top of the existing L&Fs.

To make it convenient to you, no matter you purchased source code license or not, you will have source code LookAndFeelFactory.java from the Developer Forum's custom-only area. You can download and put in your source code repository so that you can customize to fit your need.

Support for Mac OS X

In the 1.2.6 release we added support for *AquaLookAndFeel* from Apple Inc on Mac OS X. Below is a screenshot of JIDE demo on Mac OS X using the *AquaLookAndFeel*. Since *AquaLookAndFeel* is only available under Mac OS X, you can only get this L&F on Mac OSX. All you need to do is set to *AquaLookAndFeel* using *UIManager* then using ***LookAndFeelFactory.installJideExtension()*** method (see below).



Internationalization Support

All Strings used in *JIDE Docking Framework* are contained in one properties file called basic.properties under com/jidesoft/plaf/basic. 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.