JIDE Docking Framework Developer Guide

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 *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.

We found several open-source or commercial products that implement dockable windows, but unfortunately none of them are built using Swing components. To meet this need, we created *JIDE Docking Framework*, which allows you to produce similar docking functionality to what you see in the Visual Studio.NET IDE but using only Swing.

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.

UNDERSTAND DOCKINGMANAGER

DockingManager is an interface for managing DockableFrames. 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);
```

rootContainer is the main window of your application. It could be JFrame, JWindow, JDialog or JApplet. However to simplify the usage of Docking Framework, we will use JFrame as an example.

The contentContainer 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.

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*. 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 3, which is a screenshot we took from Swing Sightings. The areas that marked in red can be tool windows, while each 'tab' can be considerd a tool window – in fact, there are a total of eight tool windows.

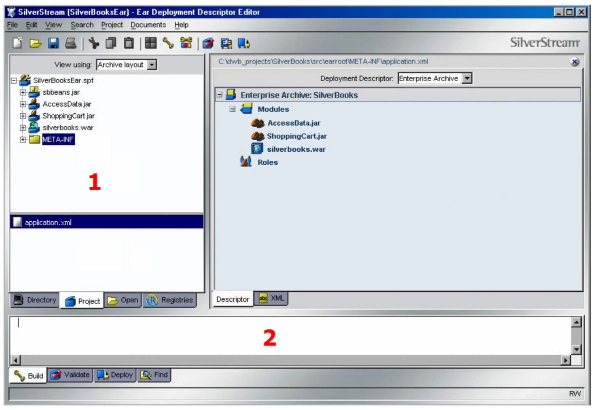


Figure 1 eXtend Workbench – SilverStream (this screenshot is copyrighted by SilverStream)

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 to be visible all the time 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*.

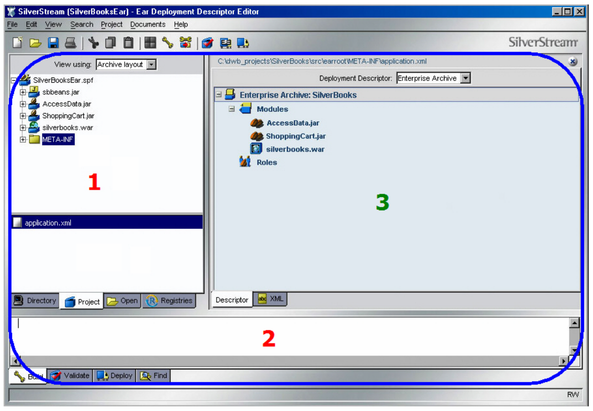
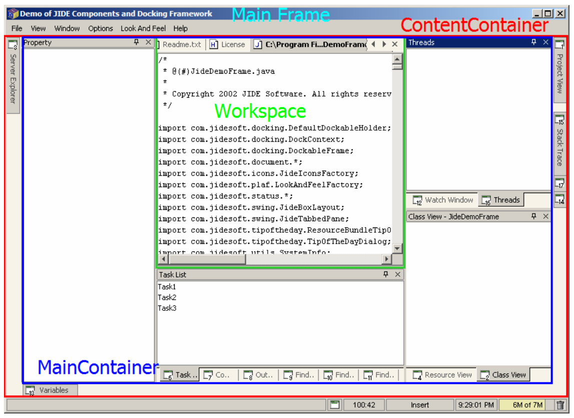


Figure 2 eXtend Workbench – SilverStream (this screenshot is copyrighted by SilverStream)

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 4), 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 5 provides another 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 *ContentContainer* 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*.



```
/*
 * @(#)JideDemoFrame.java
 *
 * Copyright 2002 JIDE Software. All rights reserv
 */

import com.jidesoft.docking.DefaultDockableHolder;
import com.jidesoft.docking.DockContext;
import com.jidesoft.docking.DockableFrame;
import com.jidesoft.document.*;
import com.jidesoft.icons.JideIconsFactory;
import com.jidesoft.plaf.LookAndFeelFactory;
import com.jidesoft.status.*;
import com.jidesoft.swing.JideBoxLayout;
import com.jidesoft.swing.JideTabbedPane;
import com.jidesoft.tipoftheday.ResourceBundleTip0
import com.jidesoft.tipoftheday.TipOfTheDayDialog;
import com.jidesoft.utils.SystemInfo;
```

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 DockableFrames, using code of the form shown below:

```
DockableFrame dockableFrame = new DockableFrame("Name of Frame",
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 name of the dockable frame is passed in as a parameter to the constructor. Since we use the name as a key internally, it must uniquely identify a dockable frame in your application. If you attempt to add another frame with same name, the framework will print out an error message and do nothing. Note that since the name is not displayed anywhere on screen, you don't need to localize it. There are two more strings which will be displayed on screen – the tab title and window title. The tab title appears in the tab area along the bottom of the panel. The window title, on the other hand, is displayed on the title bar of your dockable frame. You can call setTabTitle(String) to set the tab title and setTitle(String) to set the window title. Note that the frame name, the tab title, and the title can all be different, if you wish. If you do not call setTitle(String) or setTabTitle(String) on a dockable frame then the tab title and window title will default to the name given in the constructor.

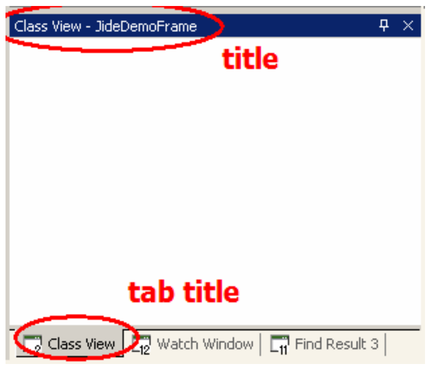


Figure 4 tab title v.s. title of DockableFrame

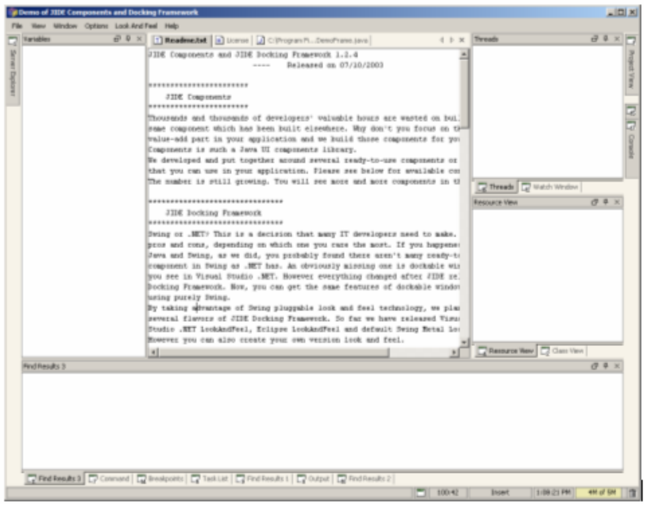
There are several methods you can use to set the initial default setting: setInitMode(), setInitSide() and setInitIndex(). Dockable windows are placed related to the *Workspace*. For example, if you want to put the dockable window to the south of workspace, you just need to set the init side to DOCK_SIDE_SOUTH. 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	0 or 1	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 _frame.getContentPane() to get the ContentPane and add whatever you want to it.

Once you have added all your components to the ContentPane, call loadLayoutData() to layout the frames. If there is no saved layout data, the Docking Framework calls resetToDefault() internally to layout the frames based on its initial default settings.

Once all dockable frames are added to the docking manager, loadLayoutData() 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_EAST_WEST_SOUTH_NORTH. If you change this to SPLIT_SOUTH_NORTH_EAST_WEST then the content pane will split vertically first, then horizontally, as shown below:



```
JIDE Components and JIDE Docking Framework 1.2.4
                        ----    Released on 07/10/2003

**********************
    JIDE Components
**********************
Thousands and thousands of developers' valuable hours are wasted on bui
same component which has been built elsewhere. Why don't you focus on t
value-add part in your application and we build those components for yo
Components is such a Java UI components library.
We developed and put together around several ready-to-use components or
that you can use in your application. Please see below for available co
The number is still growing. You will see more and more components in t

******************************
    JIDE Docking Framework
******************************
Swing or .NET? This is a decision that many IT developers need to make.
pros and cons, depending on which one you care the most. If you happene
Java and Swing, as we did, you probably found there aren't many ready-t
component in Swing as .NET has. An obviously missing one is dockable wi
you see in Visual Studio .NET. However everything changed after JIDE re
Docking Framework. Now, you can get the same features of dockable windo
using purely Swing.
By taking advantage of Swing pluggable look and feel technology, we pla
several flavors of JIDE Docking Framework. So far we have released Visu
Studio .NET LookAndFeel, Eclipse LookAndFeel and default Swing Metal Lo
However you can also create your own version look and feel.
```

Figure 5 With different split priority

You can compare this layout with Figure 3 above and see the difference.

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 window

deactivateFrame(): Deactivate a window

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

dockFrame(DockableFrame dockableFrame, int side, int index): dock frame at the specified side and index.

floatFrame(DockableFrame dockableFrame, Rectangle bounds, boolean isSingle): float frame at the specified bounds.

maximizeFrame(final 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 showFrame(). A typical use case is some important information was displayed in a dockable frame, you can use this method to grab user attention. denotifyFrame() 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 feel 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_MAXIMIZED when DockableFrame is restored from maximized mode.

PERSISTING LAYOUT INFORMATION

JIDE Docking Framework offers the ability to save windows information and settings 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**.**

getLayoutData() and **setLayoutData(String layoutData)** are methods allowing you get the layout data as a string, in case you want to load/save it without using javax.util.pref.

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 *JIDE Docking 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 **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)**.

UNDO AND REDO

Dockable window, especially drag-n-drop dockable window, is an advanced UI feature. While professional computer users can find all features after exploitation, it's hard to non-computer 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. There is resetToDefault()call will bring the layout back to the initial state. However there is no way to bring it back to a state in middle. So we introduce undo/redo function to *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 to set 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), this method is supposed to make that dockable frame floatable (or not floatable). However please do not call it for now because there are some technical difficulties in the current implementation.

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); this method is supposed to make that dockable frame autohidable (or not autohidable). Once again however, please do not call it for now because there are some technical difficulties in the current implementation.

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); this method is supposed to make that dockable frame closable (or not closable). Again, please do not call it for now because there are some technical difficulties in the current implementation.

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.

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, it's 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 and BUTTON_FLOATING. 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 1, the full outline will be painted even if it extends beyond the main JFrame; if it is 0, the outline will be clipped. In order to avoid changing the behavior of current installations, we set the default OutlineMode to 0.

GroupAllowedOnSidePane: This determine 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 initialisation stage.

Easy Tab Docking: 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!

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. Normal tabs in JTabbedPane cannot be dragged. However in *JIDE Docking Framework*, most of them can be dragged. If an application has both draggable tabs and un-draggable tabs, the user will feel confused. To make this obvious to the user, we added an option so that a 'gripper' is painted on the tab or 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) and try use other means to indicate whether the tab can be dragged or not. For example, try not to use JideTabbedPane when the tab cannot be dragged. If you only have a few dockable frames, we suggest you turn it on.

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 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);
```

ADDITIONAL METHODS

DockingManager has several additional methods that may be useful:

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

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

getSelectedFrame(): If there is a frame selected in the DockingManager, this method will give you that frame (it returns null if no frame is selected).

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.

setDefaultFocusComponent(Component): If setDefaultFocusComponent is never called, workspace will be the default focus component. When application started, this component will get focus.

setEscapeKeyTargetComponent(Component): If setEscapeKeyTargetComponent 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.

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 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

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

SidePane

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

JideTabbedPane

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

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. We provide many ways to do it.

The easiest way to do so is to call LookAndFeelFactory.installJideExtension(). This call will add necessary UIDefault not only for JIDE Docking Framework but also for all other JIDE products. Here is how you do.

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

If your application only use one L&F, you just need to call installJideExtension() once. However if you allow users to change L&F on fly, you need make sure every time you call UIManager.setLookAndFeel(), call installJideExtension() immediately.

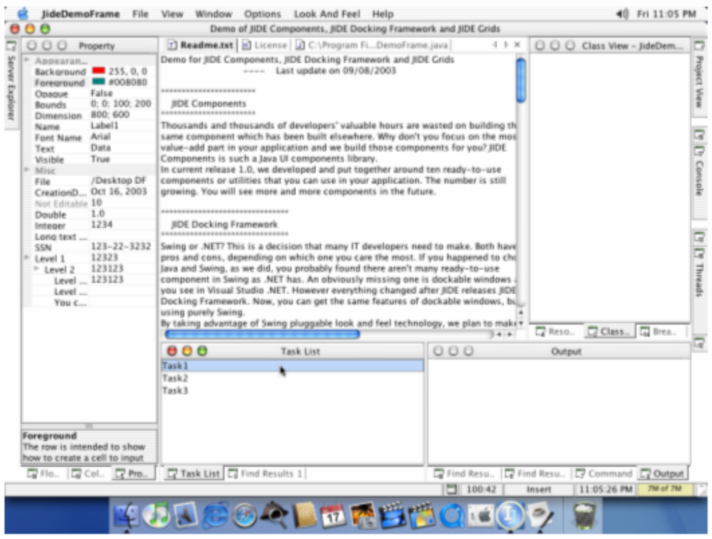
There is limitation of this method call. For example, you can only get VSNET style docking framework on Windows operation system using installJideExtension. If you want to get Eclipse style docking framework, you have to go for another way.

We also create L&F classes which already includes the necessary UIDefaults for JIDE components. Those L&F classes are VsnetWindowsLookAndFeel, VsnetMetalLookAndFeel, EclipseWindowsLookAndFeel, EclipseMetalLookAndFeel, and AquaJideLookAndFeel. The first two are based on WindowsLookAndFeel and MetalLookAndFeel respectively but both with VSNET style. The next two have Eclipse style. The last one is based AquaLookAndFeel and is only available on Mac OS X. You can use those L&Fs just like you use any other L&F. There is no need to call installJideExtension() in this case. For example, you can call UIManager.setLookAndFeel(EclipseWindowsLookAndFeel.class.getName()) in order to use Eclipse style L&F.

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, we only have one implementation of it, called AquaJideLookAndFeel. You can either set this look and feel by calling UIManager directly, or you can set to AquaLookAndFeel then using LookAndFeelFactory.installJideExtension().



Appearan...	
Background	🟥 255, 0, 0
Foreground	🟩 #008080
Opaque	False
Bounds	0; 0; 100; 200
Dimension	800; 600
Name	Label1
Font Name	Arial
Text	Data
Visible	True
Misc	
File	/Desktop DF
CreationD...	Oct 16, 2003
Not Editable	10
Double	1.0
Integer	1234
Long text ...	
SSN	123-22-3232
Level 1	12323
Level 2	123123
Level ...	123123
Level ...	
You c...	

Foreground
The row is intended to show how to create a cell to input

```
Demo for JIDE Components, JIDE Docking Framework and JIDE Grids
             ----   Last update on 09/08/2003

**********************
  JIDE Components
**********************
Thousands and thousands of developers' valuable hours are wasted on building th
same component which has been built elsewhere. Why don't you focus on the mos
value-add part in your application and we build those components for you? JIDE
Components is such a Java UI components library.
In current release 1.0, we developed and put together around ten ready-to-use
components or utilities that you can use in your application. The number is still
growing. You will see more and more components in the future.

******************************
  JIDE Docking Framework
******************************
Swing or .NET? This is a decision that many IT developers need to make. Both have
pros and cons, depending on which one you care the most. If you happened to ch
Java and Swing, as we did, you probably found there aren't many ready-to-use
component in Swing as .NET has. An obviously missing one is dockable windows
you see in Visual Studio .NET. However everything changed after JIDE releases JIDE
Docking Framework. Now, you can get the same features of dockable windows, bu
using purely Swing.
By taking advantage of Swing pluggable look and feel technology, we plan to make
```

Task1
Task2
Task3

100:42 Insert 11:05:26 PM 7M of 7M

Figure 6 Aqua LookAndFeel on Mac OS X

The support for AquaLookAndFeel is far from perfect. For example, we use the three standard color buttons to make the UI looks consistent. However they have totally different meaning. We will try to improve it in the future.

INTERNATIONALIZATION SUPPORT

All Strings used in *JIDE Docking Framework* are contained in one properties file called dock.properties under com/jidesoft/dock. We didn't do any localization for it. If you want to support languages other than English, just extract the properties file, translated to the language you want, add the correct postfix and then jar it back into jide.jar. You are welcome to send the translated properties file back to us if you want to share it.