Introduction

Asdfasdf

What is a Docking Framework?

A Docking Framework is a set of libraries that collectively add docking capabilities to an application. Before describing a "docking framework", however, it makes sense to explain what is meant by "docking" in the first place. Many applications provide a layout feature whereby a component may be dragged to a given location on the screen and it appears to "snap" into some unseen grid. This process of snapping two components together relative to each other is referred to as "docking". Hence, the component being dragged is "docked" into the application layout at the location where it has been dropped.

This dragging and docking behavior raises some interesting questions. First, how does the application know when to "dock". That is, what set of criteria triggers the application to respond to a particular drag-n-drop operation by deciding to rearrange the screen layout? What mechanism allows the application to decide how to arrange the resulting component layout once docking is complete? Are all areas of the screen capable of supporting this docking behavior or only some? At the very least, this type of behavior seems to imply that the underlying component layout is both interactive and extremely fluid. How much of the application code is devoted solely to managing this sort of dynamic layout versus dealing with the application's functional requirements? And how does the introduction of this type of behavior throughout an application impact the maintainability of its code base?

A Docking Framework is responsible for providing a set of discrete objects that manage these sorts of docking capabilities and behavior on behalf of a host application. It provides a set of services for the application and an environment in which docking layouts and features may be maintained and configured on behalf of the application. Any given application may interact with a Docking Framework to manipulate container layouts within the user interface. Interaction may range from minimal enabling of drag-n-drop docking capabilities for a relatively small number of components to full blown integration in which the framework is responsible for managing the entire window layout, providing multiple layouts that may be loaded and unloaded at will and quite possibly persisted to external storage for future use across application sessions. Ultimately, a Docking Framework assumes responsibility for managing all of the docking behaviors within an application so that the application code itself is freed of this task.

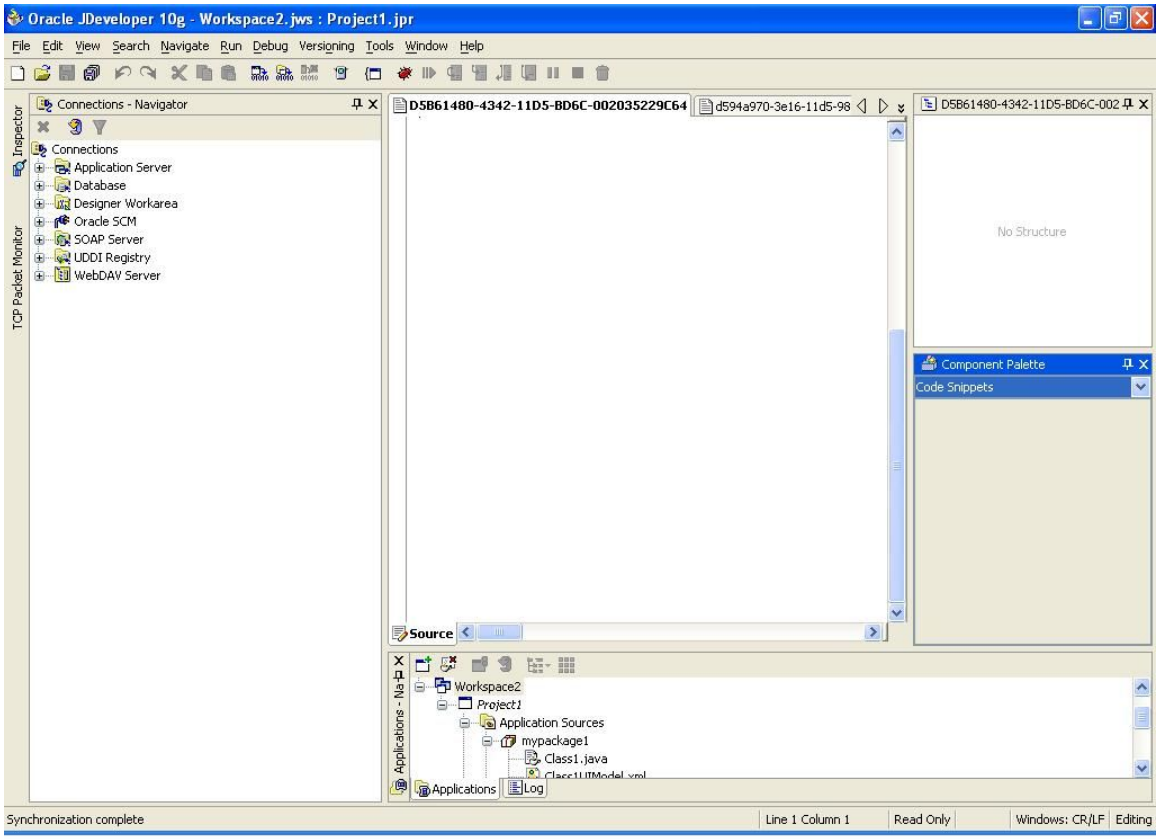
There is no formal industry standard specification that defines exactly what a Docking Framework will and will not do for an application, but there is a general consensus within the desktop development community as to some of the core features provided by any production quality Docking Framework.

- Drag-n-Drop capability
- Tabbed and Split Layouts
- Layout Persistence
- Collapsible Containers to Save Real Estate

Most Docking Frameworks and docking-enabled applications implement each of these features to at least some degree with a varying range of interpretations as to the manner in which they are presented to the end user. They each may also provide additional features to add unique value, such as floating dialog capability, configurable support for visual feedback during drag operations, support for multiple looks and feels, or support for multiple layout perspectives.

Docking Framework Examples

There are many examples in the desktop development space of both docking-enabled applications and full blown Docking Frameworks. Docking capabilities are most often associated with software development tools in the form of Integrated Development Environments (IDE). Notable docking-enabled IDEs include Microsoft Visual Studio, IntelliJ IDEA, JDeveloper, NetBeans, Eclipse, and even Macromedia DreamWeaver.



<insert screenshots>

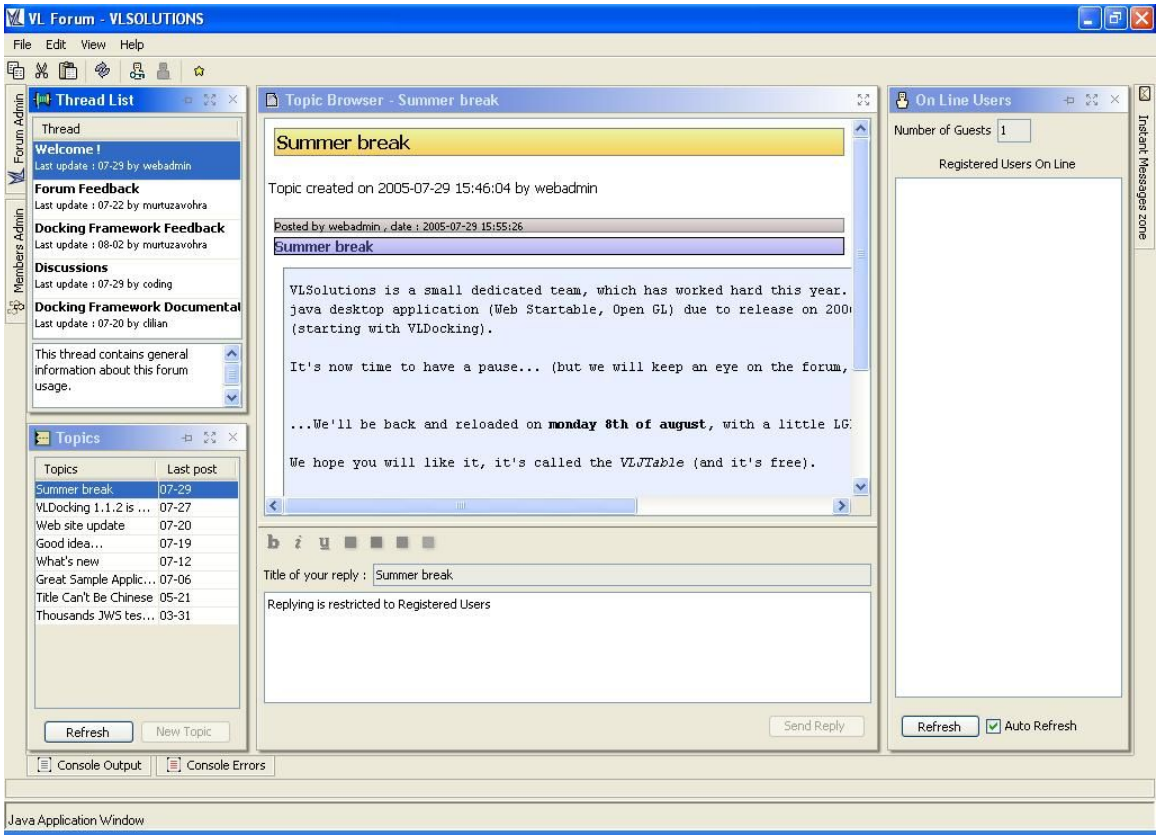
Several successful graphics editing suites provide built-in docking capabilities, such as Adobe PhotoShop and Corel Paint Shop Pro.

<insert screenshots>

Popular office suites also tend to support docking capabilities to varying degrees, including Microsoft Office and OpenOffice.org.

<insert screenshots>

Within the realm of desktop Java development, there are also several commercial Docking Frameworks that may be licensed, some under proprietary terms and others under dual-licensing systems. Of particular note in this space are JIDE Docking Framework, InfoNode Docking Windows, and VLSolutions' VLDocking Framework.



<insert screenshots and URLs>

This document will address guidelines for use of one particular Docking Framework known as *FlexDock*.

What is FlexDock?

FlexDock is an Open Source Docking Framework written in Java and released under the MIT License. MIT license is very similar to BSD license. It is designed to be a free-of-charge docking solution for both proprietary and Open Source applications written in Java using the Java Foundation Classes (JFC, also referred to as *'Swing'*). FlexDock attempts to meet the needs of the desktop Java development community through pursuit of the following goals:

1) **Simplicity**
   Adding docking capabilities to an existing application should be relatively painless. Basic scaffolding should be provided by the framework to integrate existing non-dockable components in as seamless and transparent a fashion as possible. The complexity inherent to a flexible
   Docking Framework should be hidden from the neophyte behind suitable default settings.
2) **Non-Invasive Enhancement**
   Adding docking capabilities to an existing application should not mandate that existing pieces within the application are to be swapped with special FlexDock replacements. The framework should not require users to subclass any particular FlexDock-specific Window class in order to derive the benefits of the framework. Host applications may use their own customized EventQueue or RepaintManager. FlexDock should not interfere with this behavior by installing its own version of these types of items. Most importantly, the host application should be able to maintain its original design with respect to component relationships as if FlexDock were not present. FlexDock integration should be an added enhancement for an application, not drive the application design itself.
3) **Flexibility**
   FlexDock's default behavior may or may not meet the needs of the end user. When it does not, features and behaviors should be sufficiently customizable as to meet a multitude of user requirements with varying complexity. Varying degrees of abstraction should provide a proper balance between simplicity and flexibility. If an existing public façade does not provide the necessary leverage, it should be possible to either bypass that façade and deal with a lower-level interface or to replace existing behavior by plugging in a custom implementation for a higher-level interface.
4) **Adequate Default Settings**
   FlexDock should provide default settings that integrate well into the existing host environment. On the surface, this includes support for the currently installed Pluggable Look and Feel. At a deeper level, this assumes that the current host environment will not, in many cases, provide all of the information necessary for FlexDock to perform many of its functions perfectly. FlexDock may be required to "display" components within a docking layout without an initial location specified by the application. Minimization may be requested for a component without any particular target window-edge specified. Drag-to-dock support for docking-tabs may not have been explicitly specified within the application code. In these situations, FlexDock should fail gracefully by assuming or "figuring out" an acceptable default behavior rather than throwing an exception, providing the end user a suitable API for altering the default behavior if so desired.

The current incarnation of FlexDock at version 0.4 will not meet each of these goals 100%. These, however, are the key goals around which FlexDock has been and will continue to be developed as the project approaches version 1.0.

Framework Component Overview

FlexDock manages docking behavior through the use of many different pieces working in concert, some of which the user **must** interact with and some of which the user **may** interact with if he or she so chooses. The following list describes a number of the components a FlexDock user may encounter during the development process. This list is not meant to be comprehensive, but it should provide an adequate quick-reference for some of the more common framework pieces. If this list appears somewhat daunting at first, do not worry. Each of these pieces will be described in greater detail throughout this document and intimate knowledge of each component and its use is not necessary to start being productive with FlexDock.

Docking Components

Dockable

This interface provides the public hooks necessary for FlexDock to enable and manage docking behaviors for application-level components. User components may implement this interface directly or may use FlexDock-provided wrapper classes to enable docking capabilities.

- *DockingPort*
  This interface describes the container that holds dockable components. Docking capabilities are enabled for all areas of the screen within an application encompassed by a DockingPort. FlexDock provides a default implementation for this interface, so users need not implement it themselves.
- *DockingManager*
  This class provides a public API to allow the user to configure FlexDock properties and behaviors. It also provides utility methods to allow programmatic docking and simple access to layout management.
- *View*
  This class is a FlexDock-provided container that implements the Dockable interface. It has a draggable titlebar into which Actions may be added. User-defined components that don't implement Dockable may be added to a View container to allow them to obtain docking capabilities.

1) Event Management

- *EventManager*
  This class is responsible for registering docking-related event listeners, maintaining handlers for specific types of listeners, and dispatching docking-related events to the appropriate event handlers.
- *EventHandler*
  This class manages listeners for a specific type of event. EventHandlers may be added to the EventManager to intercept specific event types and dispatch to the appropriate listeners. All DockingEvents pass through a DockingEventHandler. However, custom EventHandlers may be specified by the end user to expand the event-handling capabilities of the framework.
- *DockingListener*
  This interface defines the API used for listen for DockingEvents. Classes implementing this interface may respond to DockingEvents whenever a component is dragged, dropped, docked, or undocked. It provides the capability for detecting canceled docking operations as well.

- *DockingEvent*
  This class defines an event object used to describe a docking operation. DockingEvents are fired whenever a component is dragged, dropped, docked, or undocked. These events are intercepted by any class implementing the DockingListener interface.

2) Property Management

- *PropertyManager*
  This class manages property mappings at multiple scopes for all Dockables and DockingPorts. There are many properties used to customize docking behavior and appearance of dockable components within the framework. Responsibility for defining and maintaining these properties is delegated to the PropertyManager rather than the Dockable and DockingPort interfaces, relieving the end user of this task when implementing these interfaces. The PropertyManager also maintains "scope", managing properties at multiple levels that may override each other. For instance, it may maintain "*default*" versus "*individual*" versus "*global override*" values for the same property used by a Dockable or DockingPort.
- *DockablePropertySet*
  This interface defines a set of properties used by a Dockable instance, such as associated icons, tab-text descriptions, sibling preferences, etc. Implementations of the Dockable interface are relieved from having to define all of the properties needed by FlexDock to govern runtime appearance and behavior. Instead, Dockable implementations merely need to provide a reference to an associated DockablePropertySet through the PropertyManager and the necessary implementation of this interface is provided by the FlexDock framework itself.
- *DockingPortPropertySet*
  This interface defines a set of properties used by a DockingPort instance, such tab placement for stacked layouts, region-size preferences, etc. FlexDock-supplied implementations of the DockingPort interface are automatically provided an associated DockingPortPropertySet by the framework through the PropertyManager. Custom DockingPort implementations may used the PropertyManager to the same end.

3) Layout Management

- *LayoutManager*
  This interface, not to be confused with *java.awt.LayoutManager*, describes the component used to track the docking layout for the entire framework. It is responsible for tracking state for each Dockable and DockingPort in an application, whether a Dockable is currently hidden, embedded within a DockingPort, floating within a dialog, or minimized to an edge of the screen. It is also responsible for providing a persistence mechanism that may send a snapshot of the application's current docking state to external storage and load it back into memory at a later point in time, restoring the docking state to the visible display. FlexDock provides an implementation of this interface and end users are not required to implement it.

- *MinimizationManager*
  This interface describes the component used to "minimize" a Dockable, possibly sending it to a tabbed interface on the edge of the screen or merely collapsing it within the embedded docking layout. The MinimizationManager is responsible for interpreting user-supplied minimization "constraints" when minimizing or unminimizing a Dockable and maintaining an appropriate visual representation on the screen, such as a taskbar-like container for minimized Dockables. The installed LayoutManager will defer maintenance of "minimized" layout state to the MinimizationManager. FlexDock provides an implementation of this interface and end users are not required to implement it.
- *FloatManager*
  This interface describes the component used to "float" a Dockable, sending it to a dialog parented on the main application window as needed. The FloatManager tracks the bounds of floating dialogs, allowing them to be closed and restored as needed. It also maintains groups of floating Dockables, allowing the framework to track which Dockables share common dialogs when floating. The installed LayoutManager will defer maintenance of floating layout state to the FloatManager. FlexDock provides an implementation of this interface and end users are not required to implement it.
- *Perspective*
  This class, along with its associated classes, make up a FlexDock-provided LayoutManager implementation. A Perspective models the entire current docking layout within the application. The Perspective may be stored as an application exits and reloaded when the application restarts, preserving layout state (including minimized and floating Dockables) across application sessions. FlexDock provides support for multiple Perspectives at any given point in time, allowing applications to switch between them at runtime if so desired.

Example Code and Demo Applications

asdf

Basic Concepts

asdf

Dockable, DockingPort, and DockingManager

At the heart of the FlexDock framework are *Dockable*, *DockingPort*, and *DockingManager*. As one might guess, a Dockable is a component that can be docked, a DockingPort is a container into which a component may be docked, and the DockingManager is a class used to manage docking functionality with respect to the other two components. In reality, the DockingManager serves as a façade for other, more complex subsystems within the framework. But it does provide a relatively simple API for configuring and managing docking capabilities.

The essential concept behind FlexDock ultimately boils down to the notion of a Dockable embedded within a DockingPort. The DockingPort is a container with a very fluid, dynamic layout mechanism that allows multiple Dockables to be embedded and rearranged on the fly. By placing a DockingPort within an application window, the entire area covered by the DockingPort becomes docking-enabled. The DockingPort is designed to respond to drag-events by notifying the system of available docking space, and then providing an appropriate visual layout once a component has been added or removed.

<insert visual aid>

Although DockingPort is an interface, FlexDock provides a default implementation called *DefaultDockingPort*. Users are encouraged to use DefaultDockingPort rather than attempt to implement the DockingPort interface, as building a fully functional implementation can be a time-consuming task.

DockingPorts may contain any type of Component. However, FlexDock requires embedded Components to either implement the Dockable interface directly or provide valid Dockable implementations associated with each Component. This allows the DockingPort, as well as other lower-level aspects of the framework to manage the behavior and characteristics a component's layout and interactive functionality when embedded within a DockingPort. For instance, a dockable Component must provide a list of subcomponents that may be dragged to initiate a docking operation. The Component must provide an associated DockablePropertySet to help the parent DockingPort control the manner in which is shares space with other Components. The Component must also provide methods to allow the parent DockingPort to programmatically install "sibling" Components within the layout.

<insert visual aid>

Although implementing the necessary Dockable interface methods may sound complicated at first, the DockingManager actually handles most of these concerns automatically. For any given Component, the DockingManager is capable of creating, registering, and associating an appropriate Dockable instance with the source component. FlexDock will then track and manipulate all registered dockable Components through their associated Dockable instances.

The DockingManager is responsible for maintaining an internal repository of all Dockable instances used by the system. It provides an API for looking up Dockables by either unique String ID or by the source Component itself. It transparently registers all necessary mouse listeners to enable drag-to-dock functionality. It provides public *dock()* and *undock()* methods to allow programmatic access to the framework's docking capabilities between Dockables and DockingPorts, or between Dockables and other Dockables. Through the DockingManager, developers may enable docking capabilities on any Component and build complex docking layouts programmatically, all with a few relatively simple method calls.

Regions and Docking Layout

The term "docking layout" may be somewhat ambiguous at times. In a broad sense, it refers to the current "layout state" of every Dockable and DockingPort visible on the screen. This includes Dockables that are floating within dialogs or minimized to the edges of the application window. However, with respect to any given DockingPort, the term "docking layout" actually implies "embedded docking layout", which is the visual layout of Dockables embedded within a DockingPort. In this section, the embedded layout is what is meant by "docking layout".

Your average *java.awt.LayoutManager* defines a component layout based upon a set of rigid constraints. A BorderLayout divides the container into 5 very specific areas. A GridLayout breaks the container down into a very well defined grid where each subcomponent occupies a space within the grid. A docking layout, on the other hand, is ill-defined and very dynamic. Rather than breaking the DockingPort into concrete areas that occupy specific child Dockables, the docking layout is constructed by a set of child Dockables arranged *relative* to one another. The relative relationship between Dockables within a docking layout is referred to as a "*region*".

There are five Docking Regions used to construct a docking layout; **NORTH**, **SOUTH**, **EAST**, **WEST**, and **CENTER**. These regions are defined on the public interface *org.flexdock.docking.DockingConstants*. As a docking layout is constructed, Dockables are added to the parent DockingPort using the desired region of either the DockingPort itself, or of an embedded child Dockable.

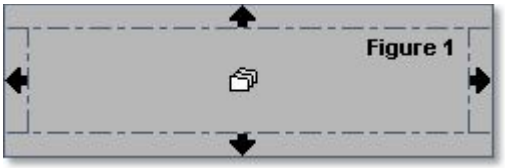


Figure 1. Docking regions.

As Dockables are dragged across a DockingPort, FlexDock makes determinations as to which region of the target DockingPort is currently under the mouse. Or, if the target DockingPort contains child Dockables, and one of these Dockables is currently under the mouse, then the current region of the child Dockable may be reported. When the mouse is released, a docking operation is initiated for the dragged Dockable into the target DockingPort for the determined docking region, relative to any existing Dockables in the DockingPort.

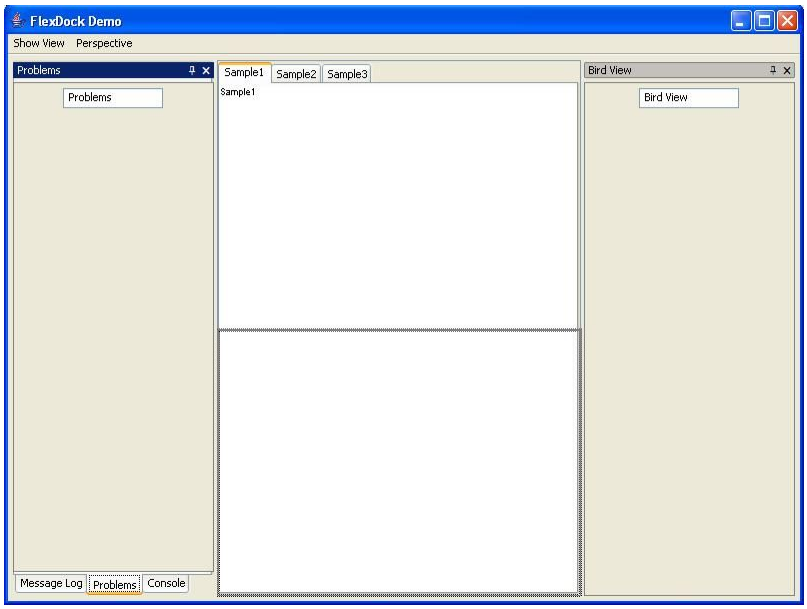


Figure 2. Before docking operation

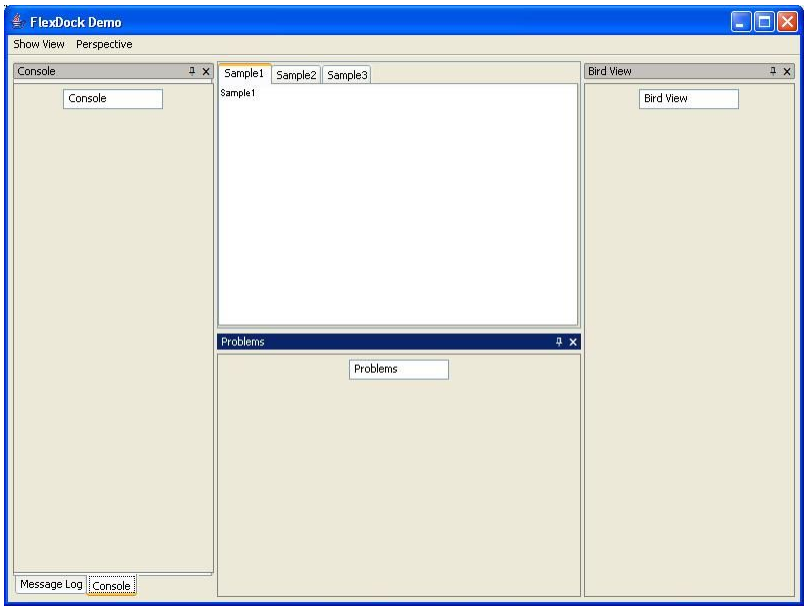


Figure 3. After docking operation

Docking layouts may be constructed programmatically through the DockingManager as well using its various *dock()* methods. This class allows the developer to manually dock any number of Dockables into a DockingPort, or relative to one another. These programmatic docking operations also require relative docking regions to be specified in order to construct the resulting docking layout.

Dockable Registration

Before a component may become docking-enabled, it must be registered with the framework through the DockingManager. The DockingManager provides three registration methods for new components wishing to have docking capabilities enabled.

```
public static Dockable registerDockable(Component comp)

public static Dockable registerDockable(Dockable dockable)

public static Dockable registerDockable(Component comp, String tabText)
```

Listing 1. Part of DockingManager interface responsible for registering Dockable

These critical method are where the FlexDock "magic" occurs that allows a component to become Dockable. For both of the "Component" methods that do not require a Dockable argument, the required Dockable instances are created and associated with the component during the registration process. Mouse listeners are registered with the component to respond to certain drag events by initiating a docking operation. Docking listeners are registered to allow the Dockable to respond to its own docking operations. PropertySets are initialized and associated with the Dockable. And a RegistrationEvent is fired for any interested parties that may wish to know when a new Dockable has been registered.

Before any component or Dockable may be used by the system, `DockingManager.registerDockable()` **must** be invoked to make it available to the framework.

Drag-n-Drop

Perhaps the most obvious means of manipulating a docking layout within an application is simply to drag the requisite components around until the desired layout is achieved. The Dockable interface provides two drag related methods, only one of which is necessary for manipulating the embedded docking layout.

```
public List getDragSources()

public Set getFrameDragSources()
```

Listing 2. Part of Dockable interface responsible for mapipulating embedded docking layout

The method `getDragSources()` is used during the registration process for drag listener initialization. The basic concept is relatively straightforward. In order to drag a component, there must be a drag listener registered for the component. If you wish to drag a panel around, you register a listener with the panel to allow it to respond to drag events. Therefore, at some point a set of listeners must be registered with the Dockable to allow it to respond to drag events. This listener-initialization process occurs when a Dockable is registered with the DockingManager.

In a real world application, it's highly unlikely users will want an entire Dockable component to respond to drag events. Typically, only a subcomponent (or a set of subcomponents) should respond to drag events on behalf of the entire Dockable. One

may think of this in terms of a real world scenario with the operating system's windowing manager. Your average window may be dragged across the screen using its titlebar. Not every area within the window responds to drag events. Dragging the menubar does nothing. Dragging the window content does not move the window around. Only the titlebar responds to drag events by repositioning the window to follow the mouse.

And so it is with the Dockable in FlexDock. If a panel is made to be Dockable, chances are users don't want the entire panel to respond to drag events. Especially if the panel contains its own draggable content. Drag-to-dock events would interfere with the expected behavior of the Dockable content in this scenario (*imagine you drag the mouse to select some text in a text field and the entire enclosing panel starts moving*). Instead, only specific pieces of the Dockable are made to respond to drag events. The `getDragSources()` method is designed to return a `java.util.List` of Components that should be used as drag sources for a given Dockable. These will typically be subcomponents within the enclosing Dockable, and in many cases the list will only contain a single Component, such as a type of titlebar.

During the registration process within `DockingManager.registerDockable()`, each of the Components returned by `getDragSources()` will have appropriate drag listeners attached, automatically allowing them to respond to drag events on behalf of the Dockable. This, of course, begs the question, *"What if I didn't implement the Dockable interface and just registered a regular java.awt.Component?"*. The answer may be variable. During registration, a Dockable wrapper instance is associated with Components that don't implement the Dockable interface. By default, the wrapper will place the Component itself within the `getDragSources()` list. This default behavior may be customized, however, as we will see later within this document.

Programmatic Docking API

Aside from drag-n-drop docking capabilities, FlexDock provides a programmatic docking API to allow developers to construct docking layouts of varying complexity. Of particular note are the various *dock()* and *undock()* methods provided on the DockingManager.

DockingManager

`public static boolean dock(Component dockable, DockingPort port)`
`public static boolean dock(Component dockable , DockingPort port, String region)`
`public static boolean dock(Dockable dockable, DockingPort port, String region)`

`public static boolean dock(Component dockable, Component parent)`
`public static boolean dock(Dockable dockable, Dockable parent)`
`public static boolean dock(Component dockable, Component parent, String region)`
`public static boolean dock(Dockable dockable, Dockable parent, String region)`
`public static boolean dock(Component dockable, Component parent, String region, float proportion)`
`public static boolean dock(Dockable dockable, Dockable parent, String region, float proportion)`

`public static boolean undock(Dockable dockable)`

The simplest method, obviously, is `undock(Dockable dockable)`, of which there is only a single version. Undocking is inherently a much simpler operation than docking. Docking requires that one ask questions such as, *"Which DockingPort would you like to dock to?"*, *"What region of the DockingPort would you like to dock into?"*, or *"Which Dockable would you like to dock next to?"* and *"How much space should be allotted for your Dockable after docking is complete?"* Undocking, on the other hand, requires only the Dockable to be undocked. If it's currently inside of a DockingPort, then it should be removed from that DockingPort regardless of its current region or allotted visual space.

There are two types of *dock()* methods provided by the DockingManager. One specifies *absolute* docking, whereas the other specified *relative* docking. In using *absolute* docking, a developer indicates that he or she would like a Dockable to be docked within a particular region of a specific DockingPort. *Relative* docking, on the other hand, implies that the developer does not care which DockingPort is in use. He or she only cares that the specified Dockable be docked relative to another Dockable which is already inside of some DockingPort. In reality, there isn't a large technical difference between *absolute* and *relative* docking. During *relative* docking, the DockingPort for the currently embedded Dockable is determined and an *absolute* docking operation is invoked behind the scenes. The real difference is the manner in which developers interact with the API. *Relative* docking hides more complexity from the end user and generally makes it much easier to construct complex docking layouts.

Developers should note that most of these methods are overloaded with Dockable versus Component parameters. These methods perform the same underlying functions and are overloaded strictly for convenience. For every registered component, there is an associated Dockable. Sometimes this may be a Dockable wrapper, sometimes not. In any case, when a Component is docked, its associated Dockable is resolved and used for the docking operation.

Creating a Working Example

In the following example we will explore the basic docking concepts described in this section. Please note that this example is designed to demonstrate the core concepts behind the FlexDock framework. As such, the code within this example may actually be **more** complicated than what is strictly necessary to enable docking capabilities within an application, since it deals with docking at a slightly lower level. Future examples will deal with stripping additional complexity from the docking process. For the present, however, we intend to focus on outlining the core docking concepts in action.

<begin example here>
asdf

Working with Raw Dockables

We have already learned that FlexDock requires docking-enabled components to either implement or be associated with a Dockable instance. However, we have not yet explored the requirements for implementing Dockable. Nor has there been any real discussion regarding alternatives to implementing Dockable directly. This section will

cover the use of raw Dockables within an application. The term "raw Dockable" is used to describe the creation of application-level classes that either implement the Dockable interface directly or make use of some type of Dockable adapter class. This is as opposed to high-level View containers, which obviate the need to implement Dockable and are discussed in a later section.

This section will cover the Dockable interface, the steps that must be taken to implement it properly, and how to work with Dockables in code. It will also cover some alternatives to implementing Dockable directly in order to simplify the process of adding docking capabilities to existing applications.

Dockable Interface Overview

The Dockable interface contains 20 methods, each of which help define the essential contract to which any component must adhere in order to be properly managed by FlexDock. The Dockable interface itself extends both DockingListener and DockingMonitor interfaces, inheriting 9 of those twenty methods from its superinterfaces. These 9 methods exist merely for event management purposes and FlexDock provides default implementations for them. This leaves 11 docking-related methods that must be implemented by the developer.

Dockable Methods

```
boolean dock(Dockable d);
boolean dock(Dockable d, String relativeRegion);
boolean dock(Dockable d, String relativeRegion, float ratio);
Object getClientProperty(Object key);
Component getComponent();
DockingPort getDockingPort();
DockablePropertySet getDockingProperties();
List getDragSources();
Set getFrameDragSources();
String getPersistentId();
void putClientProperty(Object key, Object value);
```

DockingListener Methods

```
void dockingCanceled(DockingEvent dockingEvent);
void dockingComplete(DockingEvent dockingEvent);
void dragStarted(DockingEvent dockingEvent);
void dropStarted(DockingEvent dockingEvent);
void undockingComplete(DockingEvent dockingEvent);
void undockingStarted(DockingEvent dockingEvent);
```

DockingMonitor Methods

```
void addDockingListener(DockingListener listener);
DockingListener[] getDockingListeners();
void removeDockingListener(DockingListener listener);
```

At first glance, a strategy for creating a usable Dockable implementation might not be all that apparent. Having to implement a 20-method interface hardly sounds like the sort of thing one would expect from a framework that aims to be "simple" and "non-invasive". In fact, implementing Dockable is much easier than it appears on the surface. FlexDock already provides implementations for the large majority of methods, leaving a small few that must absolutely be implemented by the application developer.

In the next two sections, we will see how to approach implementing the Dockable interface directly. Although the actual boilerplate code used for implementing Dockable is not particularly complicated, implementing the Dockable interface still requires that one add a full 20 methods to their application-level classes. This may have the effect of cluttering one's source code. To alleviate this problem, we will follow the Dockable implementation section with a demonstration of some alternatives that further simplify and automate the process.

User-Implemented Methods

Only a small number of Dockable methods must actually be implemented by the developer. They are `getComponent()`, `getDragSources()`, `getFrameDragSources()`, and `getPersistentId()`.

1) **`getComponent()`**
   A Dockable is ultimately intended to provide a docking API for some `java.awt.Component`. This method returns the Component that the Dockable represents. Usually, a Component subclass is implementing the Dockable interface directly and this method may simply return *'this'*. If, however, a wrapper class is being created, then this method should return the wrapped Component.
2) **`getDragSources()`**
   Dockable components are rearranged when the end user drags them around. However, the entire Dockable component isn't usually what responds to drag events; only certain subcomponents. For instance, if an embedded panel is to be dragged and docked, it typically has some type of titlebar that responds to drag events for the entire containing panel. In this case, this method would return a `java.util.List` instance containing the titlebar Component. In returning a List, this method sets no particular limit to the number of drag sources any given Dockable may have.
3) **`getFrameDragSources()`**
   This method is only used for floating Dockables. FlexDock provides the capability to send a Dockable into its own undecorated floating dialog. In these situations, if the Dockable provides some sort of titlebar-type component that is already part of the `getDragSources()` List, this method instructs FlexDock to treat the component as a frame-reposition drag source (such as a frame titlebar) rather than a docking drag source. All Components within this returned Set are treated as frame-reposition drag sources while the Dockable is floating, even if they are also inside of the `getDragSources()` List. If the Dockable is not floating, then this method has no bearing on the system. For those applications in which "floating" is never intended as a user-level feature, this method may simply return *null*.
4) **`getPersistentId()`**
   This method is critical to proper operation of the framework. Each Dockable must have a unique ID that is persistent across application sessions. As a "unique" ID, no two Dockables within the same JVM may share the same ID. FlexDock will rely on this ID in order to lookup and manage Dockables properly.

As a "persistent" ID, any given Dockable within an application must have the same ID each time the application is run. FlexDock will use this ID for layout persistence. If a docking layout containing the Dockable "foo" is persisted and the application is shut down and subsequently restarted, the Dockable in question must continue to have the ID "foo" in order for the previous layout to be properly restored. This means for persistence to operate properly, this method should not return UUID's or ID's based on `Object.hashCode()` or system time, since these values will be different for each JVM session.

Framework-Provided Method Implementations

The application developer must still implement the remainder of methods on the Dockable interface. In most cases, however, each implementation may be reduced to a single line of code that dispatches to a FlexDock-provided method.

Dockable Methods

```
public boolean dock(Dockable dockable) {
    return DockingManager.dock(dockable, this);
}

public boolean dock(Dockable dockable, String relativeRegion) {
    return DockingManager.dock(dockable, this, relativeRegion);
}

public boolean dock(Dockable dockable, String relativeRegion, float ratio) {
    return DockingManager.dock(dockable, this, relativeRegion, ratio);
}

public Object getClientProperty(Object key) {
    return PropertyManager.getClientProperty(this, key);
}

public DockingPort getDockingPort() {
    return DockingManager.getDockingPort(this);
}

public DockablePropertySet getDockingProperties() {
    return PropertyManager.getDockablePropertySet(this);
}

public void putClientProperty(Object key, Object value) {
    PropertyManager.putClientProperty(this, key, value);
}
```

DockingListener Methods

```
public void dockingCanceled(DockingEvent evt) {
}

public void dockingComplete(DockingEvent evt) {
}

public void undockingComplete(DockingEvent evt) {
}

public void undockingStarted(DockingEvent evt) {
}

public void dragStarted(DockingEvent evt) {
}

public void dropStarted(DockingEvent evt) {
}
```

DockingMonitor Methods

```
public void addDockingListener(DockingListener listener) {
    DockingEventHandler.addDockingListener(this, listener);
}

public DockingListener[] getDockingListeners() {
    return DockingEventHandler.getDockingListeners(this);
}

public void removeDockingListener(DockingListener listener) {
    DockingEventHandler.removeDockingListener(this, listener);
}
```

Developers may, of course, provide their own implementations for these methods. These implementations are merely provided by the framework for those who wish to spend more time with their application and less time with docking. With regard to the DockingListener methods, the interface dictates only that the methods exist on the implementing class, not that they actually perform any operation. These methods should be customized according to specific user needs.

An Example Dockable

The following code shows how one might go about creating a rudimentary Dockable implementation.

DockingStub versus Dockable

Because there are framework-provided implementations for the large majority of Dockable methods, FlexDock provides a DockingStub interface. This interface defines some basic methods needed for user-level implementation, leaving the rest of the Dockable methods to be provided by the framework.

```
Component getDragSource();
Component getFrameDragSource();
String getPersistentId();
String getTabText();
```

Any class that implements DockingStub must be some type of `java.awt.Component`. Through `registerDockable()`, the DockingManager will automatically create and initialize a valid Dockable for any DockingStub component.

The first thing to notice about DockingStub is `getPersistentId()`. This method is critical to the proper functioning of FlexDock and cannot be implemented by the framework itself. It must still be implemented at the application level.

The second thing we notice is that `getDragSource()` and `getFrameDragSource()` both return a Component, not a type of Collection. DockingStub assumes that in the majority of cases, there will really only be a single drag source for any given Dockable. This will typically be some type of titlebar component. The resulting Dockable that is created from this DockingStub will be initialized with single-item collections containing the drag source and frame drag source respectively. If floating is not an application-level requirement, then `getFrameDragSource()` may simply return *null*.

There is also the conspicuous absence of a `getComponent()` method. Unlike a Dockable, which may be a Component itself or may be a wrapper object that points to a Component,

a DockingStub **must** by implemented by a `java.awt.Component`. A DockingStub instance **is** a Component. Therefore, there is no `getComponent()` method. The DockingStub itself is used internally by the resulting Dockable's `getComponent()` method.

Finally, there is the `getTabText()` method, which does not directly correspond to anything on the Dockable interface. As Dockables are positioned within a tabbed layout inside of the docking layout, some String value must be determined for the tab-label for each Dockable. There is a means of controlling this value for raw Dockables, as we will see later. The DockingStub, however, adds tab-text directly to the public interface so that it's value cannot be neglected by the application developer.

Defining an XML-Based DockingAdapter

While the DockingStub may greatly simplify the process of creating a valid Dockable implementation, this does not necessarily address the needs of existing applications that may want to introduce docking capabilities. An existing codebase already has an established class structure and it may not be feasible to modify the code in order to start adding Dockable or DockingStub methods. This is especially true if the components are provided by a third party or are otherwise only available in binary form. Or perhaps one has access to the application code, but merely wishes to evaluate FlexDock before taking the plunge by adding FlexDock interfaces to their codebase. In these types of situations, a DockingAdapter may be used to link existing components into the docking system without modifying their code.

A DockingAdapter is similar to a DockingStub in that is provides a simplified bridge between an existing Component and the Dockable interface. It differs from DockingStub, however, in that it requires no modification of the Component class. Rather, it allows the application developer to specify a mapping between a Component class and the Dockable interface using an XML descriptor.

The DockingAdapter descriptor file has the following schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.flexdock.org" xmlns="http://www.flexdock.org">
    <!-- <adapters> is our document root -->
    <xs:element name="adapters">
        <!-- the document contains a list of <adapter> elements -->
        <xs:element name="adapter" minOccurs="0" maxOccurs="unbounded">
            <!-- each <adapter> must have a "class" attribute -->
            <xs:attribute name="class" type="xs:string" use="required"/>
            <!-- each <adapter> contains a list of <method> elements -->
            <xs:element name="method" minOccurs="0" maxOccurs="unbounded">
                <!-- each <method> has a "flexdock" attribute that maps to a -->
                <!-- specific docking property used by flexdock -->
                <xs:attribute name="flexdock" use="required">
                    <xs:restriction base="xs:string">
                        <xs:enumeration value="dockbarIcon"/>
                        <xs:enumeration value="dragSource"/>
                        <xs:enumeration value="dragSourceList"/>
                        <xs:enumeration value="frameDragSource"/>
                        <xs:enumeration value="frameDragSourceList"/>
                        <xs:enumeration value="persistentId"/>
                        <xs:enumeration value="tabText"/>
                    </xs:restriction>
                </xs:attribute>
              <!-- each <method> has a "client" attribute that maps the "flexdock" -->
              <!-- property to a method name on the client class, specified in the -->
              <!-- adapter's "class" attribute -->
                <xs:attribute name="client" type="xs:string" use="required">
            </xs:element>
        </xs:element>
    </xs:element>
</xs:schema>
```

A DockingAdapter descriptor file might then have the following structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<adapters>
    <adapter class="com.mycompany.components.SomePanel">
        <method flexdock="persistentId" client="getName" />
        <method flexdock="dragSource" client="getTitlebar" />
        <method flexdock="frameDragSource" client="getTitlebar" />
        <method flexdock="tabText" client="getName" />
    </adapter>
</adapters>
```

The idea is relatively straightforward. A user supplies an XML file containing the adapter descriptor. FlexDock loads the descriptor and, using reflection for each adapter "class", it dynamically maps the "client" method on the class to the specified "flexdock" property. In this fashion, the user need not modify their component class to implement Dockable or DockingStub. FlexDock can use the XML descriptor to create a DockingAdapter at runtime that automatically figures out the mappings between user-provided component methods and the Dockable interface.

Note that the schema defines flexdock method values of both `dragSource` and `dragSourceList`, as well as `frameDragSource` and `frameDragSourceList`. FlexDock will first attempt to find mappings for the collection-based properties and use those. If they cannot be resolved, then FlexDock will use the single-item properties instead.

The selection of descriptor file can be controlled using two constants on the class `org.flexdock.docking.adapter.AdapterFactory`.

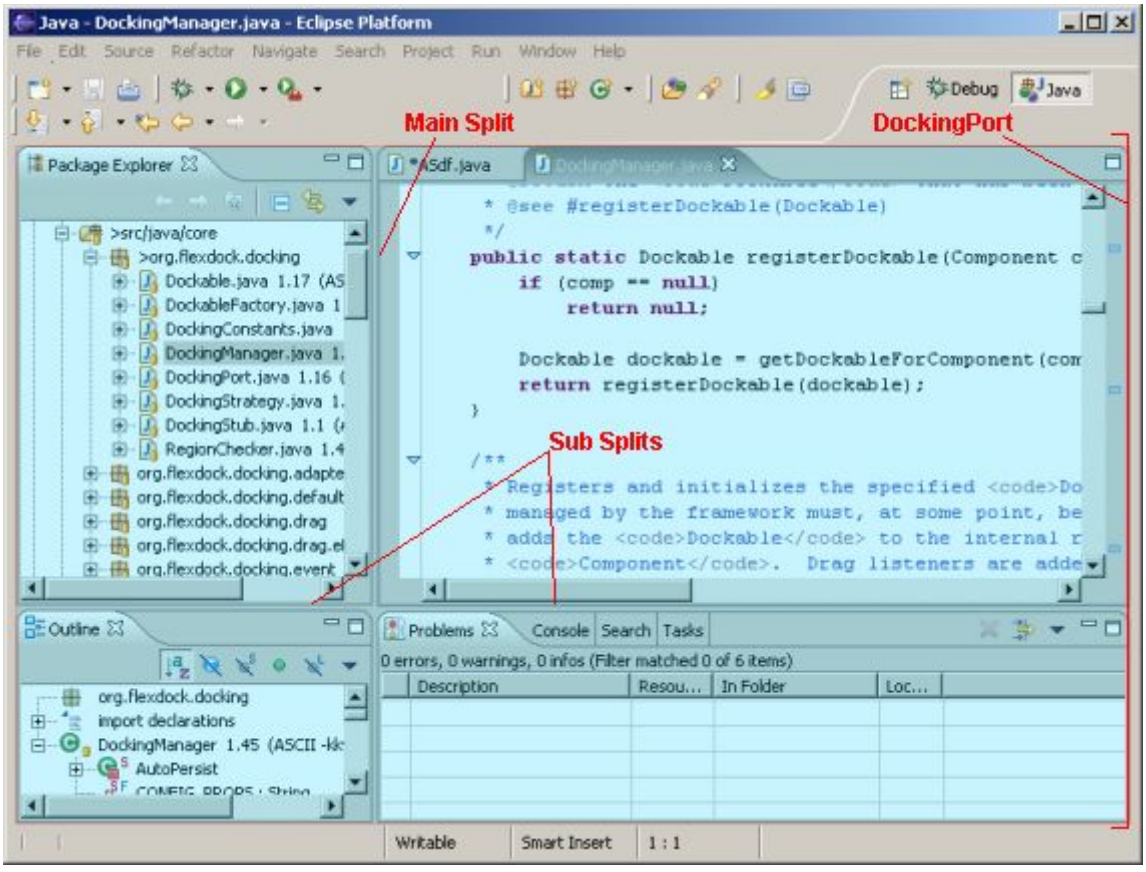
- `public static final String ADAPTER_RESOURCE_KEY = "flexdock.adapters";`
- `public static final String DEFAULT_ADAPTER_RESOURCE = "flexdock-adapters.xml";`

FlexDock will use the specified classpath resource for its DockingAdapter descriptor. By default, it searches the classpath for a resource named *"flexdock-adapters.xml"*. However, this setting may be overridden using the system property *"flexdock.adapters"* with a custom classpath URI (or an absolute file pathname). If the *"flexdock.adapters"* system property is set, FlexDock will attempt to resolve the custom descriptor. If it is not set, or the lookup fails, FlexDock will attempt to use the default *"flexdock.adapters.xml"* resource. Failing that, FlexDock will simply disable the built-in DockingAdapter feature.

Creating a Layout

While Dockables, DockingStubs, and DockingAdapters are useful for adding docking behaviors to specific application-level components, they all must still function within the context of a docking layout. Specifically, this will mean the embedded docking layout within a DockingPort. Layouts may be manipulated at runtime using drag-n-drop, but any given application must still provide some initial layout for its components.

FlexDock encourages users to build relative docking layouts within their applications. Consider the following layout from Eclipse:



This type of layout might be broken down into 8 different Dockables; *Package Explorer*, *Outline*, *Asdf.java*, *DockingManager.java*, *Problems*, *Console*, *Search*, and *Tasks*. The four quadrants in this layout may be characterized by three split dividers; a main vertical split divider (horizontal layout) and two horizontal sub-dividers (vertical layouts).

In order to achieve this layout, components may be docked "*in sequence*" and relative to on another. Any Dockable within the layout may act as the origin of the docking sequence, but all subsequent Dockables added to the layout will be docked relative to those that have come before.

In this example, we might consider *DockingManager.java* to be the sequence origin. It would be the first to be docked within the DockingPort.

```
DockingManager.dock(dockingManager, mainDockingPort);
```

Asdf.java may come next. Because the desired layout shows *Asdf.java* sharing the same space as *DockingManager.java* in a tabbed layout, we would say that *Asdf.java* should be docked relative to *DockingManager.java* in its CENTER_REGION.

```
dockingManager.dock(asdf);
```

The relative `dock(Dockable dockable)` method on the Dockable interface causes the supplied parameter to be docked to the CENTER_REGION. This will result in the desired tabbed layout.

Next, the entire layout must be split between the left and right sides of the screen. Since *DockingManager.java* and *Asdf.java* are already part of the docking layout, *Package Explorer* must be added relative to one of these components.

```
dockingManager.dock(packageExplorer, DockingConstants.WEST_REGION);
```

By adding *Package Explorer* to the WEST_REGION of *DockingManager.java*, the docking layout is split between the tabbed layout containing *DockingManager.java* and *Package Explorer*.

Now that the main split has been established, the two sub-split dividers may be added.

```
packageExplorer.dock(outline, DockingConstants.SOUTH_REGION);
dockingManager.dock(problems, DockingConstants.SOUTH_REGION);
```

This places *Outline* to the south of *Package Explorer* and *Problems* to the south of *DockingManager.java's* tabbed layout. This leaves tabbed docking for *Console*, *Search*, and *Tasks*.

```
problems.dock(console);
problems.dock(search);
problems.dock(tasks);
```

This nearly completes our docking layout. However, nothing has been done yet to address the divider locations. There are several ways to approach setting the divider locations. The layout image shows divider locations of roughly 30% for the main split and 70% for both sub-splits. These values can be addressed either during docking or after the fact.

Since our existing code does not address divider locations during docking, we will first attempt to set them after the fact.

```
DockingManager.setSplitProportion(mainDockingPort, 0.3f);
DockingManager.setSplitProportion(packageExplorer, 0.7f);
DockingManager.setSplitProportion(dockingManager, 0.7f);
```

This code is added after the docking layout has been completed. The first line finds the immediate split divider ***contained by*** the main DockingPort and sets it location to 30%. The next two lines find the split dividers ***containing*** both *Package Explorer* and *DockingManager.java* and sets their locations to 70%. (see the API documentation for the differences between the DockingPort versus Dockable parameter versions of `DockingManager.setSplitProportion()`).

It's important to note here for the latter two method calls that the split proportions supplied will be applied directly to the resolved split divider with respect to the available layout space. 70% means *"set the divider to 70% of the available space"*, not *"make packageExplorer take up 70% of the available space"*. The end result is the same for this example, but if *Package Explorer* happened to be in the lower portion of the split layout, then the result would be for it to take up the remaining 30% of the final layout.

This is in contrast to setting the split locations **during** docking. To accomplish this, `DockingManager.setSplitProportion()` is ignored. Instead, the original docking code is modified to read as:

```
1)  DockingManager.dock(dockingManager, mainDockingPort);
2)  dockingManager.dock(asdf);
3)  dockingManager.dock(packageExplorer, DockingConstants.WEST_REGION, 0.3f);
4)  packageExplorer.dock(outline, DockingConstants.SOUTH_REGION, 0.3f);
5)  dockingManager.dock(problems, DockingConstants.SOUTH_REGION, 0.3f);
6)  problems.dock(console);
7)  problems.dock(search);
8)  problems.dock(tasks);
```

Line 3 instructs FlexDock to dock *Package Explorer* to the WEST_REGION of *DockingManager.java* and for *Package Explorer* to take up 30% of the resulting split layout. Lines 4 and 5 create the sub-splits as before, but this time they instruct FlexDock to allot 30% of the resulting space to *Outline* and *Problems*. This is a different behavior than that displayed by after-the-fact divider modification. The divider itself isn't set to 30% of the available space. Rather, the new Dockables themselves are allotted 30% of the space, regardless of which docking region is specified. This may put the divider at 30% in some regions and 70% in others.

Managing Nested Borders

…will put this off for now because writing this section is boring.

Using a DockableFactory

The process of Dockable creation and registration may be automated through the use of a DockableFactory. Remember that before a Dockable may be used by the system, it must be registered with the DockingManager. Afterward, it may be looked up via its ID. This may lead to lots of application code in the form of:

```
Dockable viewPane = createViewPane("view.pane");
DockingManager.registerDockable(viewPane);
…
[some later method]
…
Dockable viewPane = DockingManager.getDockable("view.pane");
```

This entire process may be automated through the use of a DockableFactory. DockableFactory is an interface used for creating dockable Components on demand. It contains a single method:

```
public Component getDockableComponent(String dockableId);
```

When DockingManager performs a lookup via `getDockable(String dockableId)`, it's internal registry is checked for an existing Dockable with the specified ID. Once a Dockable as been registered, this lookup will return the requested Dockable. If the lookup fails, however, the DockingManager will check to see if a DockableFactory has been installed. If so, the specified ID will be passed to the installed DockableFactory requesting a Component. If a Component is returned, it will be automatically registered as a Dockable and the resulting Dockable will be returned by `DockingManager.getDockable(String dockableId)`.

In this fashion, applications code may simply forgo the explicit creation and registration of Dockable. By providing an application-specific DockableFactory, all calls to `DockingManager.getDockable(String dockableId)` will automatically create and register Dockables as needed before returning. By default, FlexDock does not install a DockableFactory. This option is left up to the application developer to use at his or her discretion. To manage the currently installed DockableFactory, use the following methods on the DockingManager class:

- `public static DockableFactory getDockableFactory()`
- `public static void setDockableFactory(DockableFactory factory)`

One may implement and install a DockableFactory as follows. First, assume there is some type of Dockable component used within an application called DockingPanel.

```
public class DockingPanel extends JPanel implements DockingStub {
    private String id;

    public DockingPanel(String panelId) {
        this.id = panelId;
    }

    public Component getDragSource() {
        ...
    }
    public Component getFrameDragSource() {
        ...
    }
    public String getPersistentId() {
        return id;
    }
    public String getTabText() {
        ...
    }
}
```

This example happens to implement DockingStub for the sake of brevity, but it may just as well have implemented Dockable or gone the DockingAdapter route. The next step is to implement DockableFactory.

```
public class DockingPanelFactory implements DockableFactory {
    public Component getDockableComponent(String dockableId) {
        return new DockingPanel(dockableId);
    }
}
```

Finally, within the application, install the DockableFactory:

```
DockingManager.setDockableFactory(new DockingPanelFactory());
```

Having configured the application in this manner ensures that any call to `DockingManager.getDockable("foo")` will yield the valid "*foo*" Dockable at all times, regardless of whether it has yet been instantiated or registered.

Of particular note is the fact that `DockableFactory.getDockableComponent`() returns a `java.awt.Component` rather than a Dockable instance. This allows developers implementing the DockableFactory interface the option to have their components implement Dockable directly, implement DockingStub, or merely provide a DockingAdapter XML descriptor for otherwise non-Dockable components.

Creating a Working Example

asdf

Working with Views

asdf

What is a View?

asdf

Adding User Content
asdf

Adding View Actions
asdf

Defining View Actions
asdf

Changing the Theme
asdf

Creating a Working Example
asdf

Managing the Docking Layout
asdf

Working with Minimization
asdf

Working with Floating
asdf

Working with the LayoutManager
asdf

Persisting Docking State
asdf

Creating a Working Example
asdf

Minimization and Collapsable Toolbars
asdf

Dockbar as a Minimization Manager
asdf

Dockbar Edges and Constraints
asdf

Dockable Preview

The visual aid for the user can be altered through using the DragPreview interface. The DragPreview interface is used during drag and drop operation for a better visual feedback for the user. Thanks to it user can find out what will be the final view after the docking operation is completed. By default FlexDock has couple of DragPreview implementation. The DragPreview can be changed through using the *EffectsFactory.setPreview (DragPreview)* method.

The simplest implementation is **XORPreview,** its visual feedback is simple, ie. border of the component is highlighted. It is not very eye-catchy but the final goal is achieved, user can see what is the final view when the drag operation is completed. The advantage of this DragPreview implementation is that it is small, simple and consumes little computer resources, such as memory and processor power.

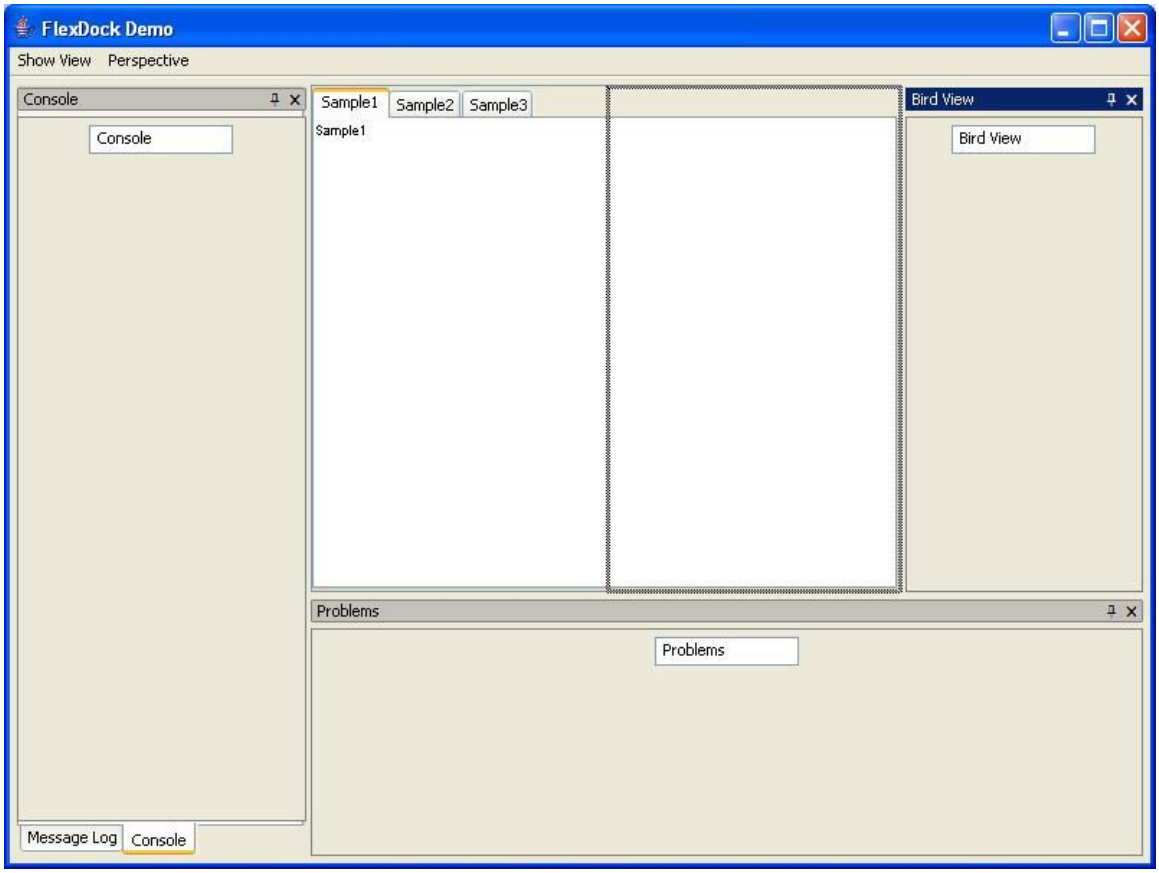


Figure 6. XORPreview in action.

More sophisticated DragPreview implementation is called **AlphaPreview** and it provides more fancy visual effect. Basicaly the AlphaPreview constructor takes two colors as arguments and the floating value from 0.0 to 1.0 indicating how transparent should be the view. As the name suggests it uses alpha channel. By default there are two predefinied alpha previews, the black and the blue preview, **AlphaPreview.BLACK** and **AlphaPreview.BLUE**. Of course the user can provide cunstructor arguments to alter the final looks of the preview. The above screenshot was taken using **AlphaPreview.BLUE**:

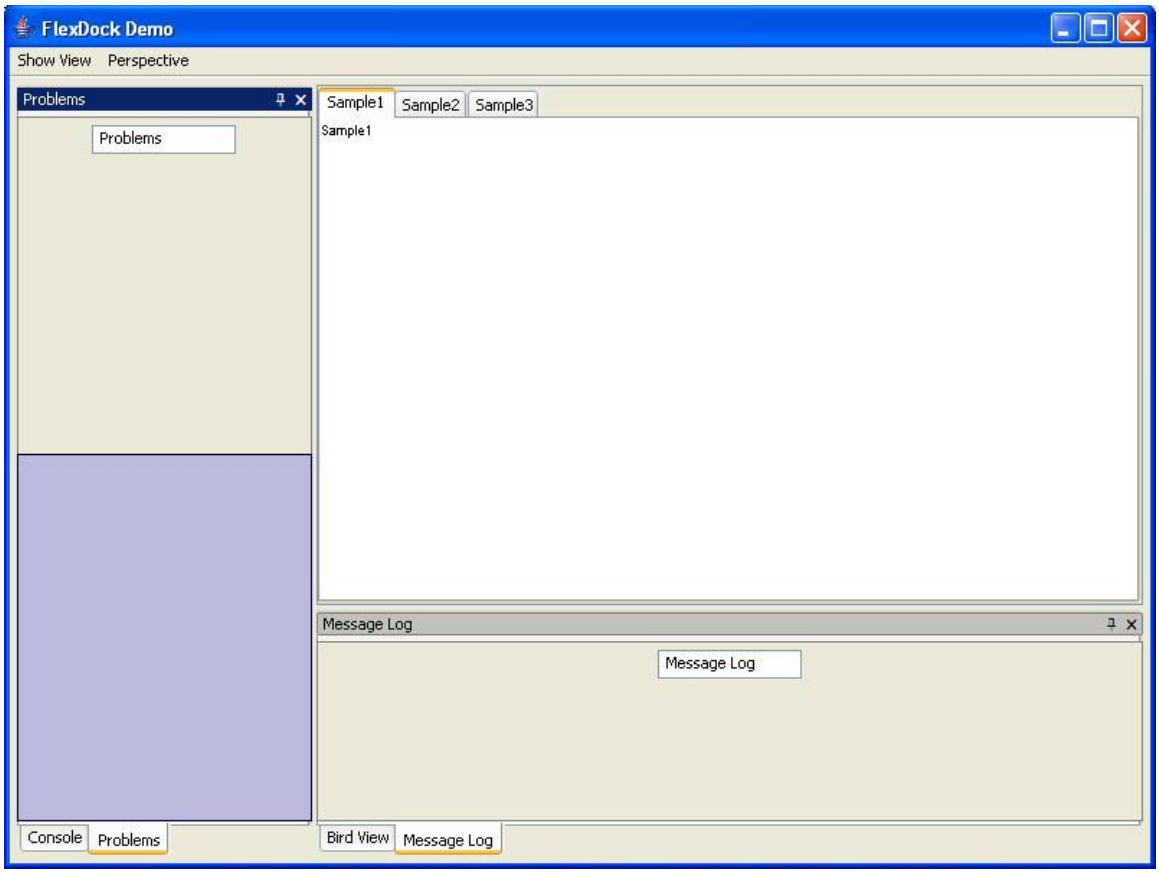


Figure 7. AlphaPreview in action.

The last and probably the most sophisticated DragPreview is called **GhostPreview**. What is quite special about this preview is that it tries to predict what the final view will look like by placing the dragged Dockable withing docking region of a target port**.** This DragPreview was probably inspired by VLDocking Framework.

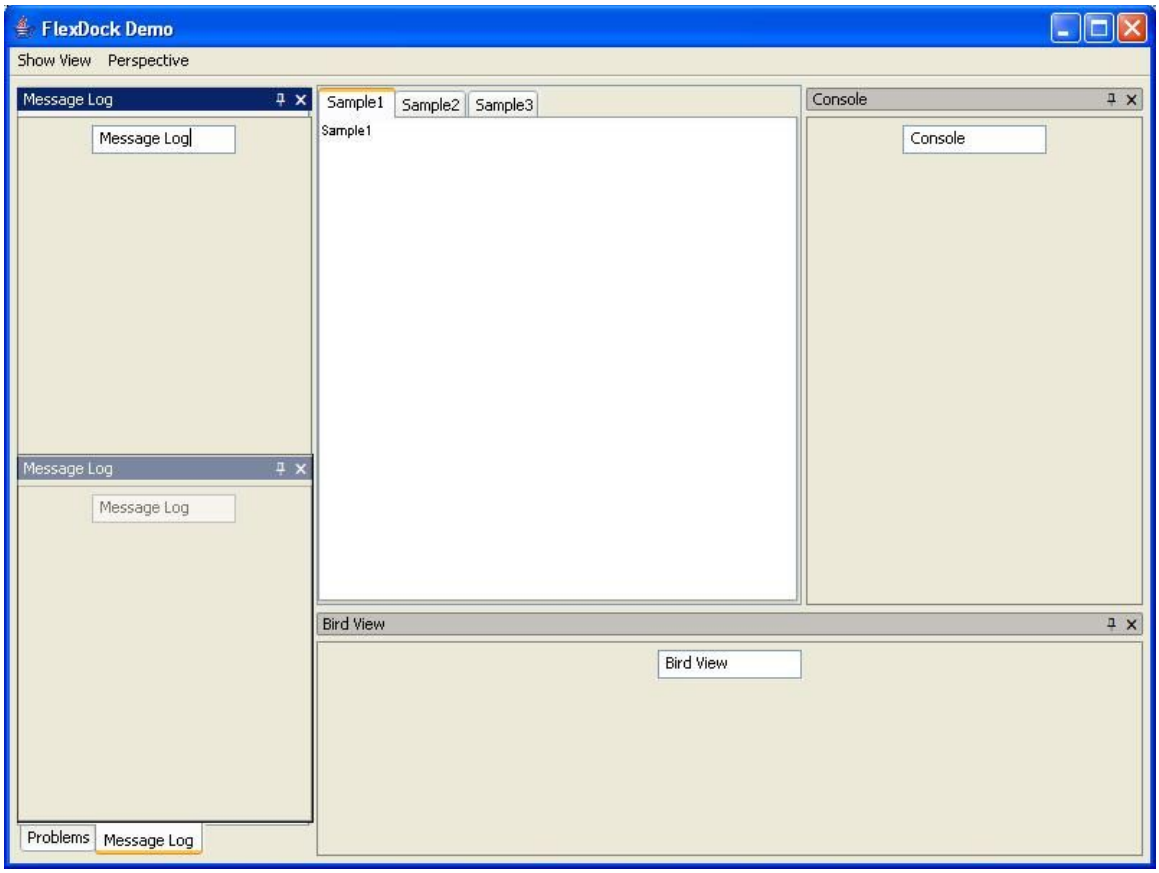


Figure 8. GhostPreview in action.

Rubber Band

Imagine you drag dockable outside the application frame. What would you expect? I assume you would expect the visual preview effect similar to the one you can see within application bounds. Unforunately, it is not possible in AWT/Swing without falling back to native code. I am not aware of any Swing based docking framework that would actually support drawing preview outside the application's frame. FlexDock has native implementation for Windows and *nix systems. If for some reasons it is not possible to initialize rubber band on a specific computer the preview outside the application bounds, so-called RubberBand is turned off.

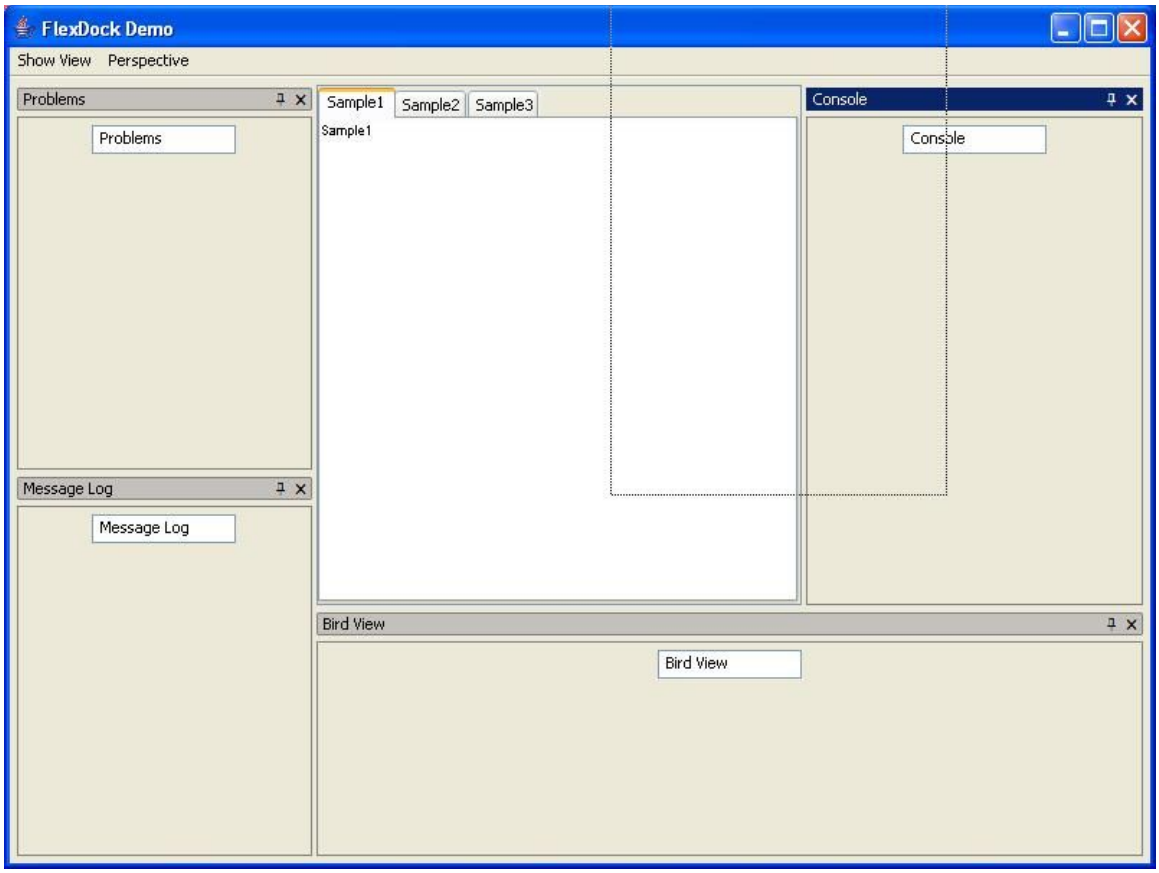


Figure 9. RubberBand in action under Windows

Dockable Restoration
asdf,

Creating a Working Example
asdf

Working with Perspectives
asdf

Perspectives as a LayoutManager
asdf

Usinga PerspectiveFactory
asdf

Customizing Persistence Location
asdf

Customizing Persistence Format
asdf

Creating a Working Example
asdf