JDockers Window Manager
Functional Specification

Author: Winston Prakash
Revision: 1.0
Last Modified: Aug 12, 2003

Introduction

JDockers implementation of Window Manager could be looked as two parts

- Developer Perspective – How module developers could place their windows on a window system and let the window manager manages them without knowing the internals of a window manager. This could be done through providing an Abstract Window System to the developers. The actual implementation of this abstract window system is transparent to the developers. Module developers should never directly call the API of this implementation.

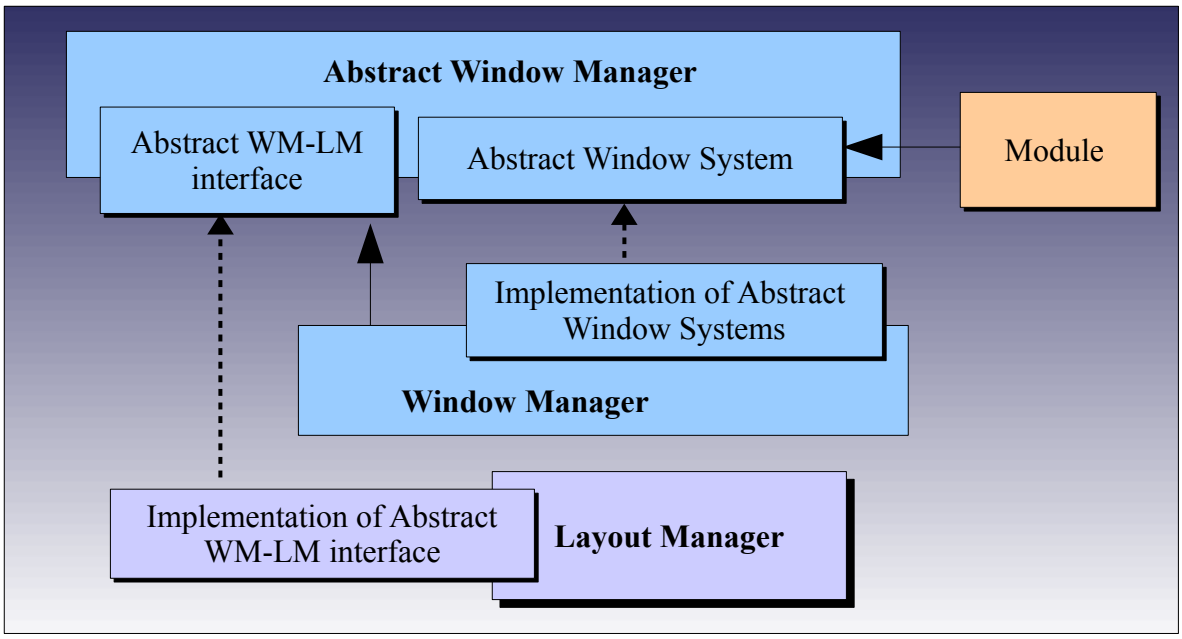
- User perspective – How the layout manager should layout the windows for the users. This is done through a Layout Manager or Layout Manager.

  The interface between the implementation of Abstract Window System and the layout manager should not be hard coded. But itself should be abstracted which the Layout Manager should implement.

Now on, to distinguish the different parts, I'll refer

The Abstract Window System as Abstract Window Manager
The implementation of Abstract Window System as Window Manager
The implementation of Layout manager that also implements the WM-LM interface as Layout Manager.

The following diagram explains the view provided earlier



Window Manager

Window Manager is the implementation of windows Abstract API. Implementation takes care about main window, workspaces, modes, component container, toolbars and menubars.

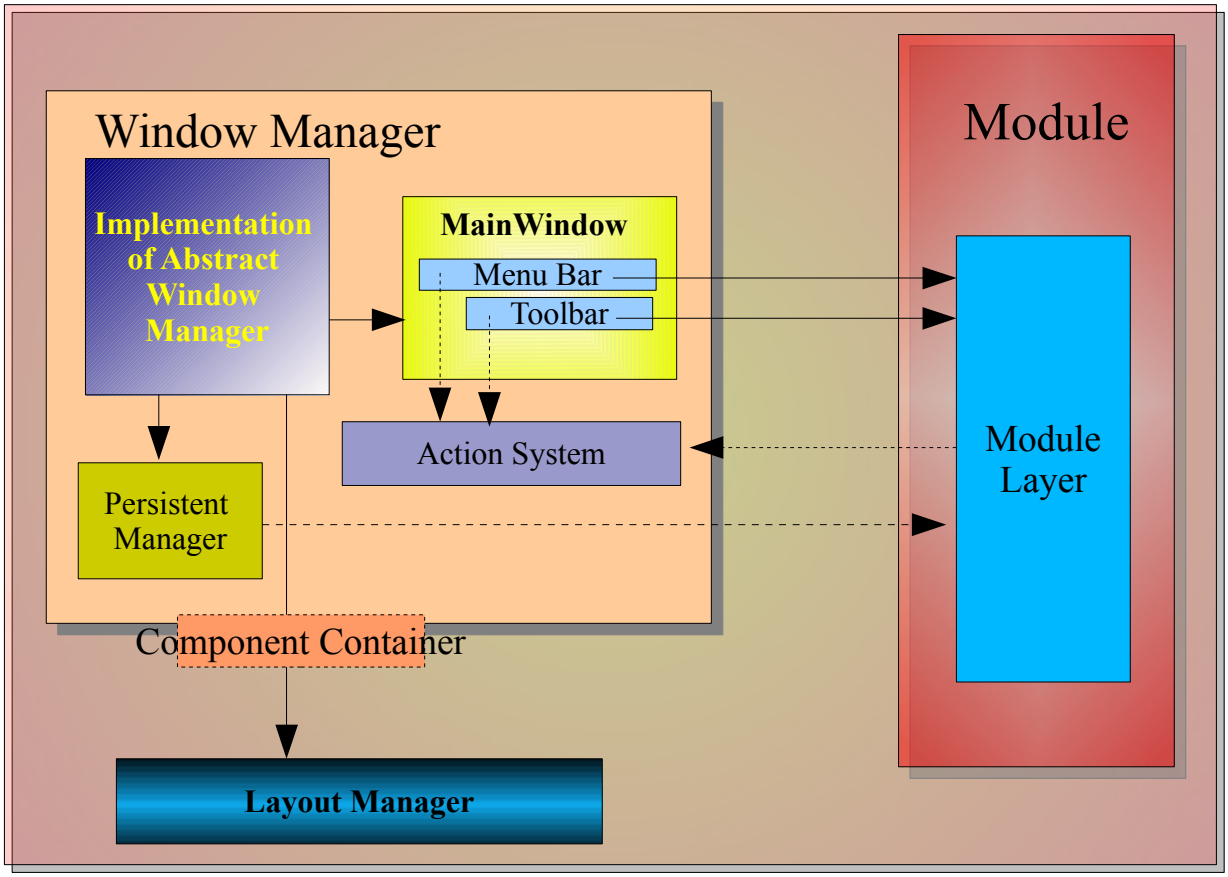
Even though Window Manager manages all the windows, the laying out the windows in the Layout Pane is delegated to the Dockable or layout manager.

Loading: Single point start of the Window Manager. This should be called only once preferably by the core after all the modules are loaded by the Module Manager. At this point window manager should create the Main Window. Then Main menu, Toolbar Pool and Status Bar and add it to the Main Window.

Note: Currently Main Window does the creation of its components such as Main Menu, Toolbar Pool and Status Bar etc.

Saving: Single point to shut down the Window Manager. Also this should be called only once preferably by the core. Window Manager instructs the Layout Manager to save the current Window layout it manages. Then it saves the information about its structure in a XML file.

Note: Window Manager first build its structure from the XML layer of the module (only once). Then it saves it structure in a XML file which is used during next time invokation. If the Window Manager is been reset using -wmreset flag, then the structure is re-read from the XML layer.



Structure of Window Manager

Component Container

A top component is a Swing component (usually a panel or the like, though not necessarily) which might exist in its own window; or it might be docked along with other top components into one tab of a multi-tabbed window. It is the basic panel where the modules will place their GUI components. From the module developer perspective Component Container are attached to the Mode.

Some component container can be *cloned*, meaning that a new top component created will have the same data and the cloned component will simply be a new view on it. For example, Editor panes can be cloned to create split views of the same Data.

Note: Only Document type Window (see below) should be allowed to be cloned.

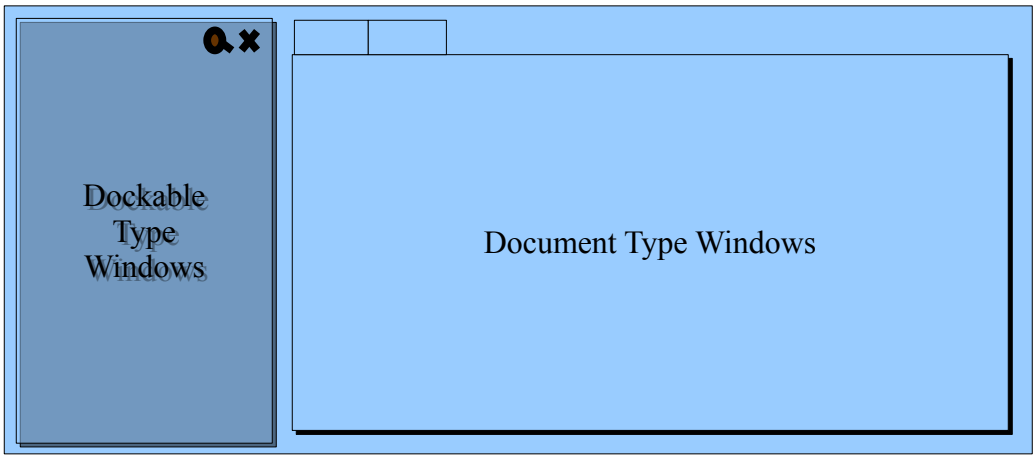
Mode

Defines the type of Windows the modules could place for the manager to manage them. Also defines other attributes of the Window, such as constrain hints for placing the window, if the window can be minimzed to the side etc. Each mode can hold one or more topComponents. There are two types of Windows defined modes

Mode API

- Dockable Type – These windows can be drag and droped and docked on the four sides of the Layout Panel. Usually they hold panels of non-editor type panel such as explorer, properties, project manager etc

- Document Type – Document panels are usually tabbed (tab is usaually at the top of the Pane) at the center of the Layout Pane and usually holds the documents that are been edited. The mode that holds document type can have single view (i.e each tab has one document Panel) or multi view (each tab has more than one document), the view switching could be done through tabs at the bottom. In this case the document pane will have another tabbed pane with tabs at the bottom.



It is entirely up to the layout manager to place them. No restriction will be imposed by the window manager. However, Window Manager will pass on the hints to the Layout manager through the WM-LM interface

Workspaces

Workspaces fork the users work flow from one state to another – say editing state to debugging state. Workspace could be defined as collection of modes.

Workspace switching is accomplished through Layout Manager as Layout switching. Once Window Manager is about to switch workspace, it would ask the Layout Manager to save the current layout (editor state layout) and would ask the Layout Manager to put another set of modes in the new layout. If the new layout is not existing already, then the initial hints to place the modes will be provided by the Window manager.

Each workspace groups modes, not top components directly, so users desiring that similar multi-tabbed windows with different contents appear in their own workspaces, should create new mode to hold them. This is only for initial palcing. Once the user changes the layout then these initial settings will get disturbed and layout at that time will be saved by the Layout Manager.

Typically IDE starts with a defualt workspaces – "Editing" for editing state.

The WM API provides access to a set of Workspace and WS API set of modes for each workspace, each of which contains some top components. The API user can create new modes using Workspace API, and attach top components into a single mode and set the mode name and icon and initial constraint - the Layout Manager implementation automatically displays the modes in an appropriate fashion, possibly with user's saved layout.

Finding workspaces, modes, and component container

A module developer may want to inspect the structure of Window Manager thus the workspaces, modes, and top components. Window Manager and its components provides API for

- Retrieving the currently active Window Manager
- Retrieving all or current workspaces
- The elements in a workspace i.e the modes displayed in it
- All the topcomponent attached to a single Mode. Also currently active topComponent.

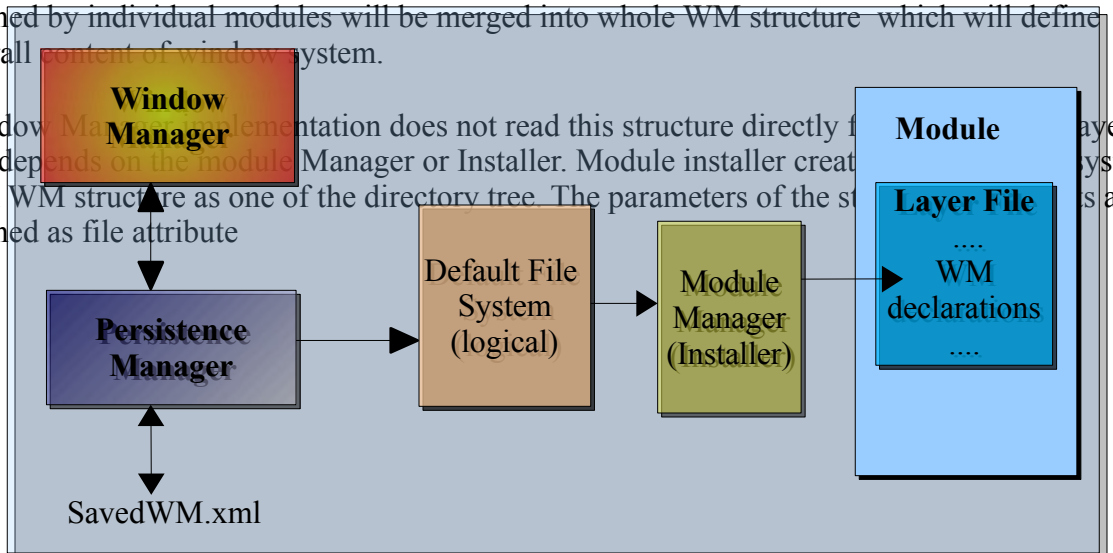
Window manager implements the following componets from openide WM

Interface (Abstract WM)	Implementation (Concrete WM)
Window manager	Window manager Impl
Workspace	Workspace Impl
Mode	Mode Impl
TopComponent Manager	TopComponent Manager Impl
TopComponent Container	Default TopComponent Container Impl

Persisting Window Manager Structure

Each module will have a possibility to define part of WM structure in their module layer, expressed as XML document.

This uses the advantages of XML layers architecture. The workspaces, mode and topcomponent defined by individual modules will be merged into whole WM structure which will define overall content of window system.



Window Manager implementation does not read this structure directly from layer. But depends on the module Manager or Installer. Module installer creates file system with WM structure as one of the directory tree. The parameters of the structures are defined as file attribute

The persistence manager also offers method for retrieving serialized TopComponent using specified ID (the one specified in TopComponent)

Automatic deinstallation support:

Modules that define workspaces, modes or component references can "mark" these items so that the system can perform automatic removal of these items when the module is uninstalled.

Localization support in module layer

Candidates for localization are display names of and modes and top component. Both kind of names are highly visible in running system, so if the module can be run under various locales, then right localization.

Localization support is represented by "display name" elements in XML describing workspaces and modes.

Probable attributes

Attribute `displayNameKey`
: Key for the Localizable display name. Usually contains string bundle key like `CTL_Mode_Name`. The attribute displayName could be used in case no localization is required.

Attribute `bundle`
: This attribute must be specified, if displayNameKey is specified. The bundle must contain string bundle key and attribute `bundle` must contain class name of the localization bundle.

Events fired by Window Manager to the Component Container

Component Opened

Called only when top component is opened first time for the first time on some workspace. The intent is to let the subclass of Topcomponent to perform initializing tasks.

Component Closed

Called only when top component is closed and removed completely from the Window System The intent is to let the subclass of Topcomponent to perform cleaning tasks here.

Component Shown

Called every time the top component is about to be shown. Shown here means the Dockable Window is activated or shown and visible. This event will be fired if the Dockable Window is expanded from its collapsed state.

Component Hidden

Called every time the top component is hidden. Hidden means the TopComponent Container still exists in the system but not visible.

Component Actiavted

Called when this component is activated. This happens when the parent window (Dockable Window or document) of this component gets focus and this component is active tab if it is in a tab group.

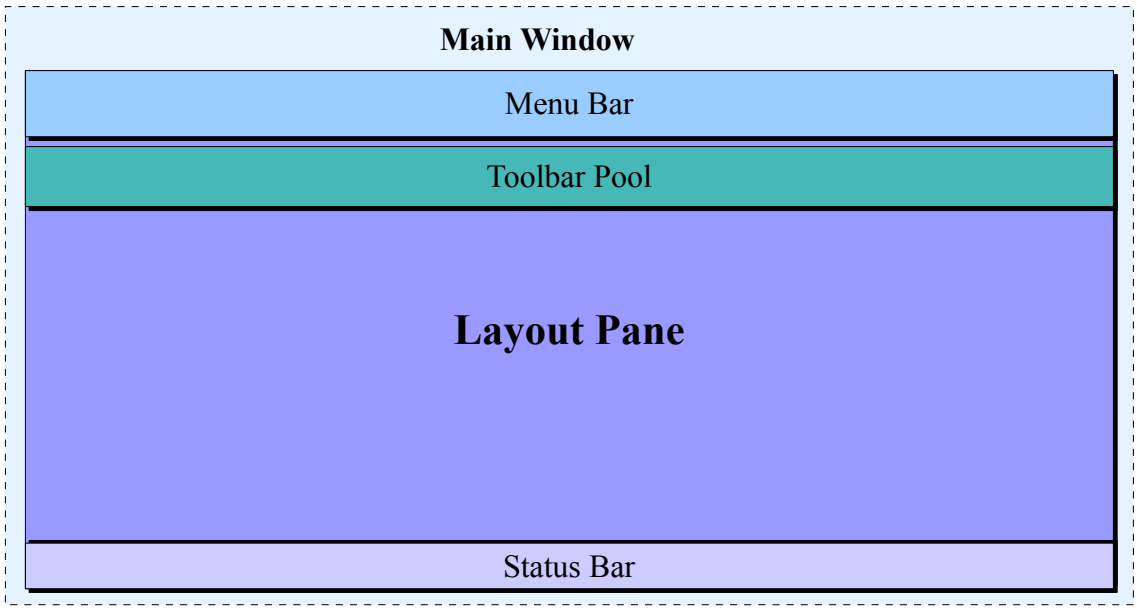
Component Deactivated

Called when this component is deactivated. This happens when the parent window of this component loses focus and this component is the current active tab in case of tab group.

Main Window

Window Manager provides the Main Window. It should be possible to obtain the Main Window object directly from the Window Manager. Useful to set the Main Window as the Parent of some user defined dialogs.

Mainwindow is a container for application menu bar , toolbar pool, status bar and area for Layout Pane, which are provided and controlled by the window Manager.



ToolbarPool

ToolbarPool is nothing but a set of toolbars governed by toolbarpool configuration. Only one ToolbarPool associated with a configuration will be visible at a time. A customizer popup dialog allows the user to customize this configuration. The same customizer also allows the user to

customize each of the toolbar in that toolbar pool.

Let us trying define some terminology here ..

Layout Manager provider will be providing the following that will be accessed by the WM

- Layout Manager (LM)
- Layout Pane (LP - LayoutManager.getLayoutPane() will give the associated LP)
- Dockable Window (DW- available in two forms - dockable & non-dockable).

Components that may be internal to LM, that are not accessed directly by WM are

- Document container
- Dockable Container
- Autohide Container

LayoutPane

LayoutPane is the place where the LayoutManager would place and manage the windows. Layout Pane is not instantiated directly. Window Manager would get it from Layout Manager using

LayoutManager.getLayoutPane();

Layout Pane is placed at the center of the Main Window.

Layout Manager

Layout Manager implementation is available to the window manager through WM-LM interface. Using Lookup Layout Manager will attach itself as the service provider.

An event model is defined in the Layout Manager for interaction between Window manager and the windows in the Layout Manager. Layout manager will never talk to the Window Manager directly, but fire events that will be listened by WM. However, WM will send messages to LM by invoking its methods directly.

```
public class LayoutFactory {
  private static LayoutManager lmInstance;
  private LayoutFactory() { }
  public static final LayoutManager getLayoutManager() {}
}
```

Interface for LayoutManager

```
public interface LayoutManager {
  JComponent getLayoutPane();
  void setLayoutPane(JComponent jComponent);
  DockableWindow createDockableWindow(String string, String string1, ImageIcon imageIcon);
```

```
  void addListener(LayoutManagerListener layoutManagerListener);
  void removeListener(LayoutManagerListener layoutManagerListener);
  void openDockableWindow(DockableWindow dockableWindow);
  void closeDockableWindow(DockableWindow dockableWindow);
  void showDockableWindow(DockableWindow dockableWindow);
  void hideDockableWindow(DockableWindow dockableWindow);
  void activateDockableWindow(DockableWindow dockableWindow);
  void updateDockableWindow(DockableWindow dockableWindow);
  void autohideDockableWindow(DockableWindow dockableWindow);
  DockableWindow findDockableWindow(String string);
  DockableWindow[] getAllDockableWindows();
  void hideAllDesktopWindows();
  DocumentWindow createDocumentWindow(String string, String string1, ImageIcon
imageIcon);
  void addDocumentWindow(DocumentWindow documentWindow);
  void removeDocumentWindow(DocumentWindow documentWindow);
  void activateDocumentWindow(DocumentWindow documentWindow);
  void updateDocumentWindow(DocumentWindow documentWindow);
  DocumentWindow findDocumentWindow(String string);
  DocumentWindow[] getAllDocumentWindows();
  void removeAllDocumentWindow();
  void addWindowSet(String string, Vector vector);
  void removeWindowSet(String string);
  void showWindowSet(String string);
  void hideWindowSet(String string);
  void showWindowSet(Vector vector);
  void saveLayout(File file);
  void loadLayout(File file);
  void saveLayout(String string);
  void loadLayout(String string);
  void enableDebug(boolean boolean0);
}
```

Events from Layout Manager related to Dockable Window

Layout Manager fires Dockable Window Event to its Listener as defined below

```
public interface LayoutManagerListener {
  void dockableWindowOpened(DockableWindowEvent dockableWindowEvent);
  void dockableWindowClosed(DockableWindowEvent dockableWindowEvent);
  void dockableWindowShown(DockableWindowEvent dockableWindowEvent);
  void dockableWindowHidden(DockableWindowEvent dockableWindowEvent);
  void dockableWindowActivated(DockableWindowEvent dockableWindowEvent);
  void dockableWindowChanged(DockableWindowEvent dockableWindowEvent);
}
```

DockableWindow

Dockable Window is where the layout manager will place the components given to the layout manager by the window manager.

DW takes two forms..

- In non-dockable (or document) form, it resides in the splittable document container as tab. Document container does not support D&D. But the DW in the document container support D&D. i.e, when the document container is splitted it may be possible to D&D a tab (corresponding to a DW) in one split area to the other.

- In dockable form, it resides in a splittable dockable containers as tab. Dockable container, as well as their tabs support D&D. Dockable container exists in two modes.

    - Autohidable docked mode. In this mode it is always docked to the sides of the LP. When dockable container is autohided its tabs are placed in a Autohide container which is docked to the side of LP

    - Floating mode. In this mode it floats independent of the LP.

It should be possible to D&D DW (i.e by dragging the corresponding tab) between a docked or a floating dockable container but not between document container and dockable container.

It should be possible to convert a DW from one form to other . (i.e non-dockable to dockable and vice versa)., through context menu of the tab (corresponding to the DW) in the dockable container or document container.

When no dockable container is available the whole layout pane will be availabe to document container. The document container resizes to accomodate the dockable containers and Autohide Container.

Each Dockable Window itself is a tabbed pane for multiView support

DnD for docking: docking between containers (as tab group) by dragging the tab of Dockable Window and the whole group by dragging their container title.

DnD for container split: Container split by dragging frame tab to proper place (see details in the UI spec)

Interface for Dockable Window

```
public interface DockableWindow {
  void addDockableWindowListener(DockableWindowListener dockableWindowListener);
  void removeDockableWindowListener();
  void setName(String string);
  String getName();
  void setTitle(String string);
  String getTitle();
  void setIcon(ImageIcon imageIcon);
```

```
  ImageIcon getIcon();
  void setBounds(Rectangle rectangle);
  Rectangle getBounds();
  void setInitialDockSide(String string);
  String getDockSide();
  void setInitialState(String string);
  String geState(String string);
  void addComponent(JComponent jComponent, String string);
  void removeComponent(String string);
  void removeAllComponents();
  JComponent[] getAllComponents();
  void setClosable(boolean boolean0);
  boolean canClose();
  void setTabName(String string);
  String getTabName();
}
```

Object of this event will be fired when events occurs in the dockable window

```
public abstract interface DockableWindowEvent {
  void setDockableWindow(DockableWindow dockableWindow);
  DockableWindow getDockableWindow();
  void setActiveComponent(JComponent jComponent);
  JComponent getActiveComponent();
}
```

This is a convenient listener. Not necessary to implement this unless additional events directly from the dockable window is required. Otherwise use the Layout Manager listener that fires required events.

```
public interface DockableWindowListener {
  void dockableWindowDeactivated(DockableWindowEvent dockableWindowEvent);
  void dockableWindowAutoHidden(DockableWindowEvent dockableWindowEvent);
  void dockableWindowFloated(DockableWindowEvent dockableWindowEvent);
  void dockableWindowDocked(DockableWindowEvent dockableWindowEvent);
}
```

Document Window

This is a conveinent interface for providing specialized dockable window. Useful to create document window that get tabbed at the center.

```
public interface DocumentWindow extends DockableWindow {
  void addDocumentWindowListener(DocumentWindowListener documentWindowListener);
  void removeDocumentWindowListener();
}
```

This is a conveinent interface for providing specialized dockable window event. Useful to create

document style window event.

public interface DocumentWindowEvent extends DockableWindowEvent {
}

This is a conveinent interface for providing specialized dockable window listener. Useful to create document window style listener.

public interface DocumentWindowListener extends DockableWindowListener {
}

Document Window Area

Document Window area is a Layout Pane area that keeps all of opened document windows. It is upto the Layout Manager to lay these windows. However, the suggested layout is to place them in a tabbed pane with one document visible at a time. Also Document area could be implemented as one or more tabbed pane (document group)and placed in a split panes. This allows more than one document to be visible at a time. In this case the document tabs need to have context menu that allows moving the document among the window group. The same effect could be achieved through DnD.

Opening and closing a document window should not affect the size of the document window area. When the last document window is closed, document window area stays empty at the same size.

Adding a Dockable Window in pinned (docked) mode resizes the document area. However, Dockable Window added in collapsible state does not affect the size of the document area.

DockableWindow area

There is no specific area for DockableWindows unless they are in pinned or collapsible state.

The area allocated to the Dockable Windows if they are pinned are always in the sides of the LayoutPane depending on the docking side of the DockableWindow.

A small "gutter" area could be allocated for placing the icon and names of the Dockable Windows if one of the Dockable Window is in collapsible state. If both pinned & collapsible mode exist on the same side, then the outermost area is allocated to "gutter" for collapsible frames.