Drag-And-Dock Design Pattern

Paulo Santos
EFACEC
paulo.santos@efacec.pt

Ademar Aguiar
FEUP
ademar.aguiar@fe.up.pt

ABSTRACT

The *Drag-And-Dock* design pattern provides a structured solution for designing graphical software applications with multiple content views that end users can freely arrange following a dragging and docking interaction approach.

General Terms

Algorithms, Design, Standardization.

Keywords

Design, pattern, drag, drop, dock, graphic user interface usability.

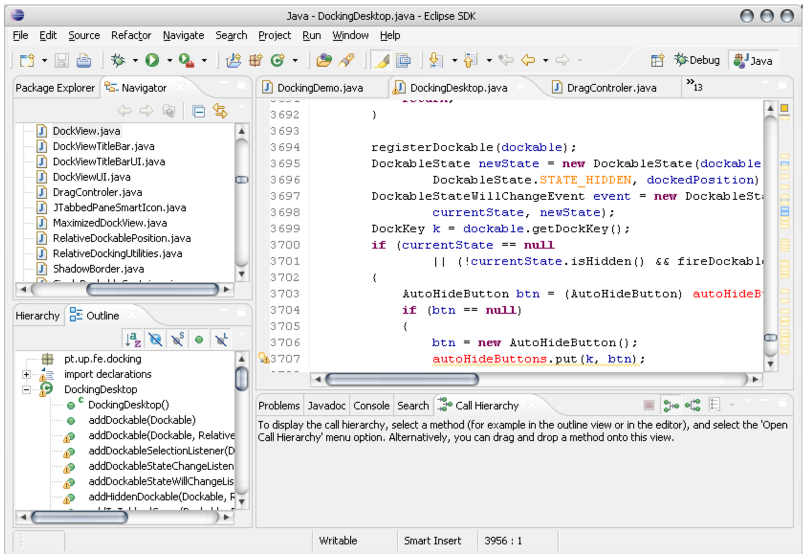
1. Example

More and more usability concerns are in place when it comes to develop complex graphical applications that maximize end users' satisfaction, learnability, and effectiveness while working with them.

Much of these usability aspects can be achieved by an intuitive and well laid out graphical user interface. However, many modern applications aren't focused on a single content view, but rather on several other content views interrelated (or not) with a main one. Integrated Development Environments (IDE), such as NetBeans [1], Visual Studio .NET [2], or Eclipse [3] (see Figure 1), are just examples of such kind of applications. Most IDE's have a main content view the user is mostly focused on, and simultaneously a few others the user commutes focus with, such as navigation views, properties views, or status message views, to mention a few.

At startup, such applications provide their default content views arranged in the way considered the best suited for most of the users. Many times, an easy way to exchange layouts is provided, so that it's not too restrictive and let users switch between predefined content views disposition schemas and order. However, advanced users often demand even more, expecting more freedom to organize the views of the software applications they use everyday as they see as fitting better.

Such user freedom can be accomplish by allowing them to drag individual (or groups of) content views within the software application, and docking it to the sides of any other content view, or even into any other group of content views, thus enabling users to arrange the application content views, into almost an infinite number of different layout schemas (see Figure 2).

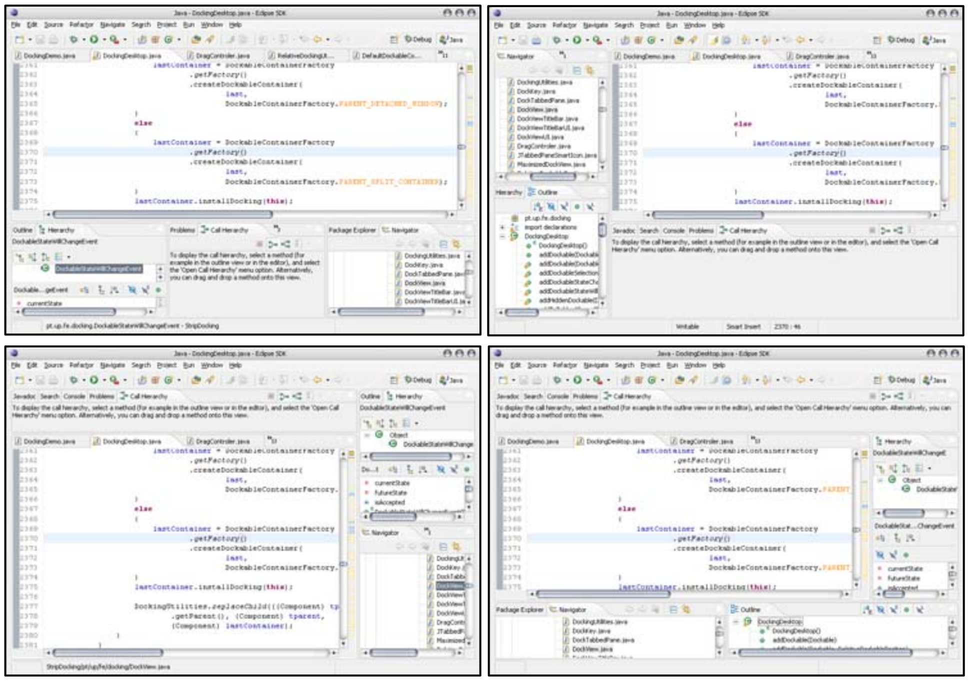


```
        }

        registerDockable(dockable);
        DockableState newState = new DockableState(dockable
                DockableState.STATE_HIDDEN, dockedPosition)
        DockableStateWillChangeEvent event = new DockableSt
                currentState, newState);
        DockKey k = dockable.getDockKey();
        if (currentState == null
                || (!currentState.isHidden() && fireDockabl
        {
            AutoHideButton btn = (AutoHideButton) autoHideB
            if (btn == null)
            {
                btn = new AutoHideButton();
                autoHideButtons.put(k, btn);
```

To display the call hierarchy, select a method (for example in the outline view or in the editor), and select the 'Open Call Hierarchy' menu option. Alternatively, you can drag and drop a method onto this view.

Figure 1. Eclipse IDE



```
                lastContainer = DockableContainerFactory
                        .getFactory()
                        .createDockableContainer(
                                last,
                                DockableContainerFactory.PARENT_DETACHED_WINDOW);
            }
            else
            {
                lastContainer = DockableContainerFactory
                        .getFactory()
                        .createDockableContainer(
                                last,
                                DockableContainerFactory.PARENT_SPLIT_CONTAINER);
            }
            lastContainer.installDocking(this);
```

To display the call hierarchy, select a method (for example in the outline view or in the editor), and select the 'Open Call Hierarchy' menu option. Alternatively, you can drag and drop a method onto this view.

Figure 2. Eclipse: four different arrangement layouts

2. Context

Graphical software applications with multiple and distinct content views displayed at the same time, which end users are able to freely organize by docking them to each other.

Personal Object Space [5] in general, and in particular Movable Panels [4], states a Human-Computer Interaction (HCI) pattern to allow end users organize user interface pieces (content views) at will, within a graphical application. This is accomplished by inducing the user to *grab* an individual content view, *drag* it around the application, and *drop* it wherever they would like (Drag-And-Drop [6]), forming as many different layouts as possible.

3. Problem

Movable Panels allows for content views to float within the application, even superimposing each other. However, restrictions can be applied to where a content view can be dropped, and how it behaves by then.

Such restrictions can state that a content view can only be dropped precisely onto the edges of other content views, or on top of them. After being dropped onto a valid dropping location (edge), the dropped content view will set aside the target content view, with a sliding edge between them that the end user can later use to resize the surrounding views. If dropped on top of other content view, the dropped content view will set on top of it, while the target remains in the same place but now identified by a special graphical handler (such as a tab).

How to structure the implementation of user interfaces employing a Movable Panels HCI with docking behaviour and related restrictions?

4. Forces

A solution to this problem must balance the following forces:

- clear separation between the interaction roles required to support the dragging, and docking mechanism, and the specificities of the content views being manipulated
- flexibility to support any type of content views
- versatility to arrange any layout schema
- maintain layout schema consistency
- independency from the design of graphical user interface libraries

5. Solution

To support a Movable Panels interaction model with docking restrictions, provide the following four key design elements:

- *View*, the content view itself;
- *Draggable*, the content view handler the end user can grab and drag around the application;
- *Dockable*, the content view container where content views can be stacked or docked onto the sides;
- *SplitContainer*, the border between two content view containers.

These four elements can then be managed by a *Mediator* element, that monitors *Draggable* elements being dragged hover *Dockable* elements, at the same time it checks for an eligible docking area, and finally to request undock and dock actions onto the two *Dockable* elements involved (source and target).

Figure 3 identifies the basic design elements of the solution on top of their graphical output, whether Figure 4 illustrates the solution using a possible aggregation of objects.

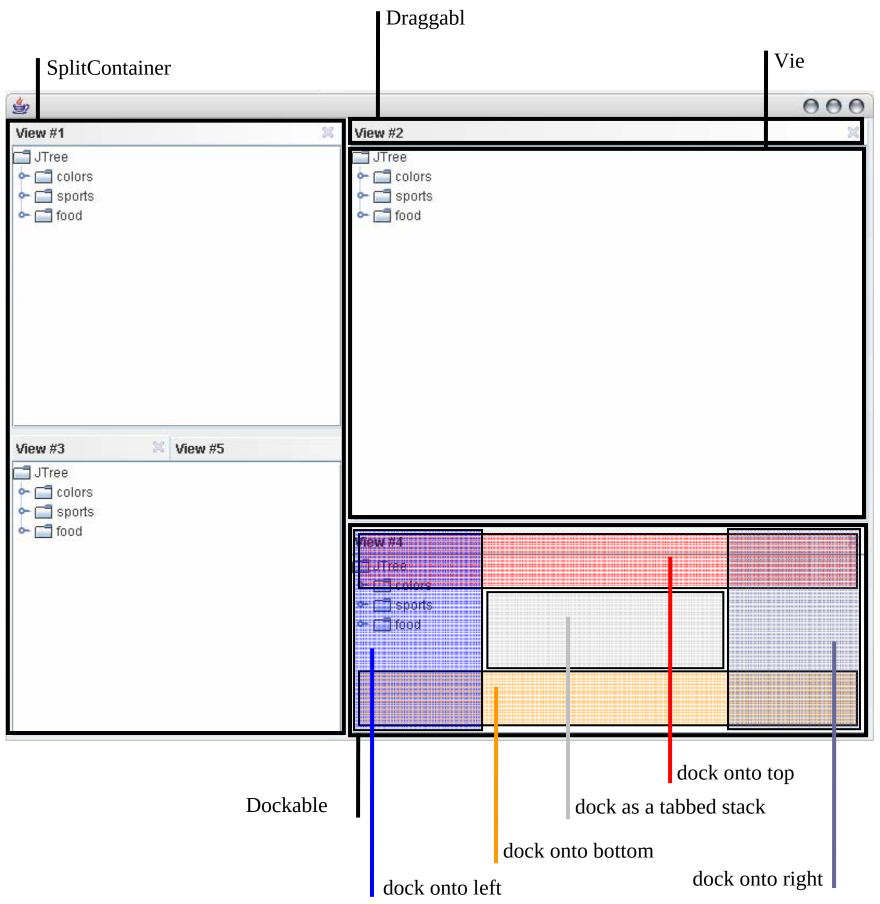


Figure 3. Basic elements for docking multiple content views

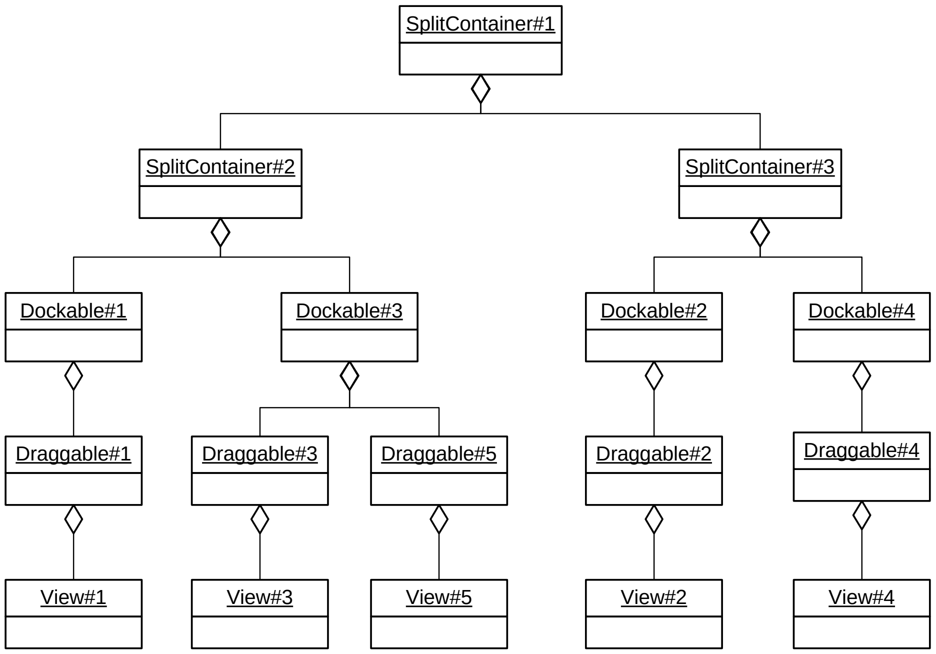


Figure 4. Object aggregation of basic elements

6. Structure

The solution provided defines and relates six major roles to be played by the solution participants, described in this section: *Draggable*, *Dockable*, *Mediator*, *View*, *Container*, and *SplitContainer*. Figure 5 shows an overview of the whole *Drag-And-Dock* design pattern structure.

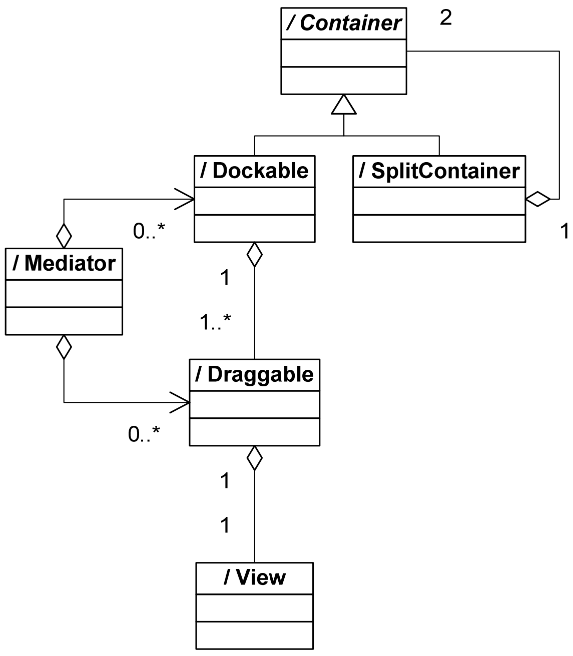


Figure 5. Overview of the *Drag-And-Dock* structure

6.1 Draggable

A *Draggable* has the ability to be visually grabbed and dragged around the application by an end user. It directly represents a unique *View* and can be docked onto a *Dockable*. At all times, a *Draggable* resides within a *Dockable* so it always knows its current parent.

While being dragged, a *Draggable* must publish its current position to a *Mediator*, as well as when that action ends with a release event.

Role	***Collaborations***
Draggable	▪ View ▪ Dockable ▪ Mediator
Responsabilities ▪ Wrap an individual *View* with the ability to be dragged and docked onto a *Dockable* ▪ Knows which *Dockable* it is docked onto ▪ Triggers drag and drop events, into a *Mediator*	

6.2 Dockable

A *Dockable* contains multiple *Draggables* piled altogether (only one content visible at a time), distinguished normally by individual tabs.

A *Dockable* is able to accept docking actions of a *Draggable*, required by a *Mediator*, to one of its four edges or to add it to its stack of *Draggables*. It is also capable of undocking a specific *Draggable* from its stack.

Furthermore, a *Dockable* provides visual feedback of docking possibilities, given a set of coordinates within its boundaries.

Role	***Collaborations***
Dockable	▪ Draggable ▪ Mediator
Responsabilities ▪ Containing *Draggables* ▪ Docking *Draggable* onto its sides ▪ Adding *Draggables* to its stack ▪ Undocking *Draggables* from its stack ▪ Validate docking possibility	

6.3 Mediator

A *Mediator* monitors when and where a *Draggable* is being dragged, at the same time it verifies which *Dockable* is directly under the dragging position, calling for docking validation onto the target *Dockable*.

Moreover, it monitors when and where a *Draggable* being dragged is released, calling for undocking and docking actions (onto source and target *Dockables*, respectively), once docking possibility is confirmed by the *Dockable* directly under the release position.

Role	***Collaborations***
Mediator	▪ Draggable ▪ Dockable
Responsabilities ▪ Monitor drag and drop events from *Draggables* ▪ Assess which *Dockable* is the target of a drag hover and docking action ▪ Request docking and undocking actions	

6.4 View

A *View* is in fact a unique content view integrated within the multiple content view application environment. It is part of a single *Draggable* so it can take advantage of *Drag-And-Dock* capabilities.

When desired, it is the *View* responsibility to define integration restrictions, such as preferred dimensions, maximize and minimize permission, display name, etc.

Role View	***Collaborations*** ▪ Draggable
Responsabilities ▪ Display contents ▪ Define integration restrictions	

6.5 Container

The *Container* role is an abstraction representing a generic container in the layout hierarchy of the *Drag-And-Dock* design pattern. Its only duty is to retain a reference to a *Container* parent, a *SplitContainer*.

Role Container	***Collaborations*** ▪ Dockable ▪ SplitContainer
Responsabilities ▪ Keep track of its parent *SplitContainer*	

6.6 SplitContainer

A *SplitContainer* main purpose is to provide a visual sliding edge between two other *Containers*, in a vertical or horizontal manner. Furthermore, it is able to set and remove its *Containers*, as well as to replace one *Container* with another. Thus, allowing for any layout schema, based on rectangular *Views*.

Role SplitContainer	***Collaborations*** ▪ Container
Responsabilities ▪ Separate vertically or horizontally two other *Containers* ▪ Resize *Containers* immediately within ▪ Exchange *Containers* immediately within	

7. Dynamics

There are four main actions in this pattern: dragging a *Draggable* hover a *Dockable*; dropping a *Draggable* on a Dockable; docking a *Draggable* onto a *Dockable* specific location; and undocking a *Draggable* from a *Dockable*.

7.1 Dragging

A *Mediator* gets notified whenever a *Draggable* is being dragged over some coordinates. It then assesses which *Dockable* is directly under those coordinates, reporting to it that there is a *Draggable* hover. The target *Dockable*, then verifies if it’s possible to dock the *Draggable*, providing some visual feedback to the end user (mouse pointer indication, or drawing the target docking area), and returns the possible docking location (left, top, right, bottom, or stack), if any (see Figure 6).

7.2 Dropping

When a *Draggable* ends a dragging action, by being released, a *Mediator* gets notified. It then assesses which *Dockable* is directly under those coordinates, reporting to it that there is a *Draggable* hover. If a valid docking location was returned, it then starts by undocking the released *Draggable* from its parent, and finishes by docking it onto the target *Dockable* specific location (see Figure 7).

7.3 Undocking

Undocking a *Draggable* from its parent *Dockable* is as straightforward as unsetting its parent.

However, there’s more to it if the *Dockable* has no more *Draggables* stacked on. In this case the *Dockable* itself must be destroyed, while the layout structure remains coherent. The *Dockable* removes itself from its parent *Container* (*SplitContainer*), which in turn replaces itself with the remaining *Container* on its own parent *Container* (*SplitContainer*). In the end, both *Dockable* and its parent get destroyed (see Figure 8).

7.4 Docking

Docking a *Draggable* occurs on one of several docking locations, usually five, in a *Dockable*.

Docking onto stack requires setting the *Draggable* parent to the new *Dockable* one and adding the *Draggable* to the *Dockable* pile of *Draggables*. Graphically, only one of the *Dockable* stack of *Draggables* is visible, but all are graphically accessible and identified, frequently through tabs.

Docking onto top or left edges, requires the creation of a new *Dockable* to hold (stack) the *Draggable*, and the creation of a new *SplitContainer* (vertical fashion, if top location, horizontal otherwise). Then there’s the need for the new *SplitContainer*, to take the place of the *Dockable* on its parent *SplitContainer*. Finally the new *Dockable* must be set as the first element (top/left) on the new *SplitContainer*, and the *Dockable* the second element (bottom/right).

Docking onto bottom or right edges is like docking onto top or left, it has the same steps, except for setting the first and second elements on the new *SplitContainer*. In this case, the first element (top/left) must be the *Dockable*, and the second (bottom/right) the new *Dockable* (see Figure 9).

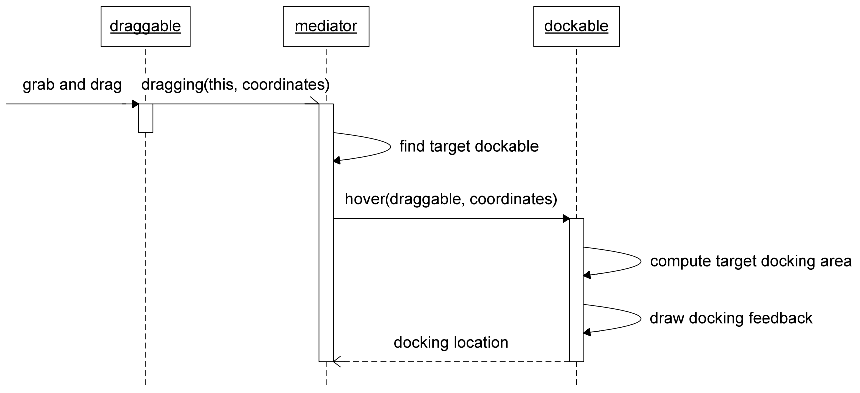


Figure 6. Dragging action

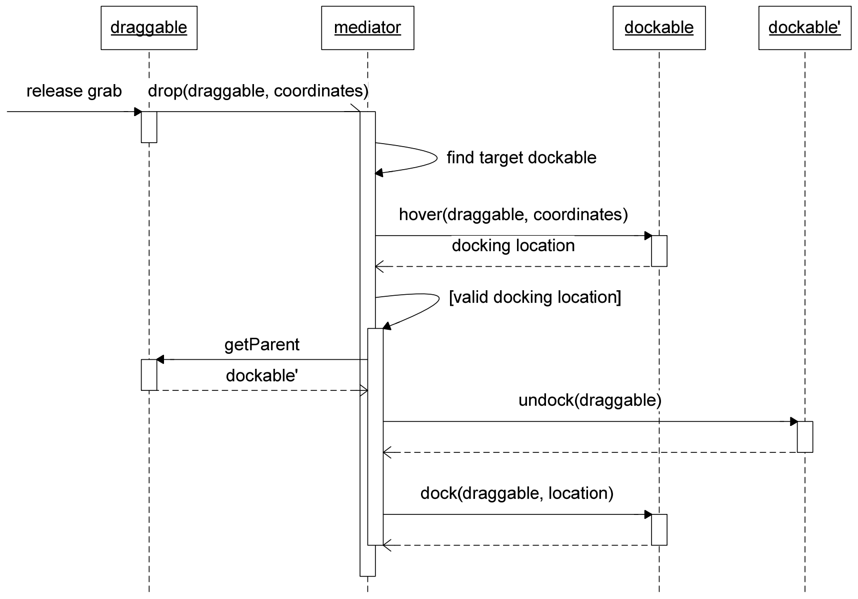


Figure 7. Dropping action

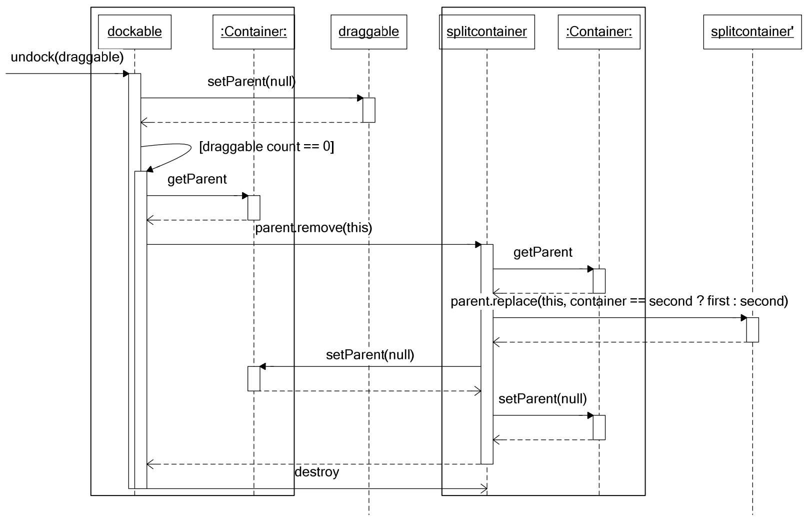


Figure 8. Undocking action

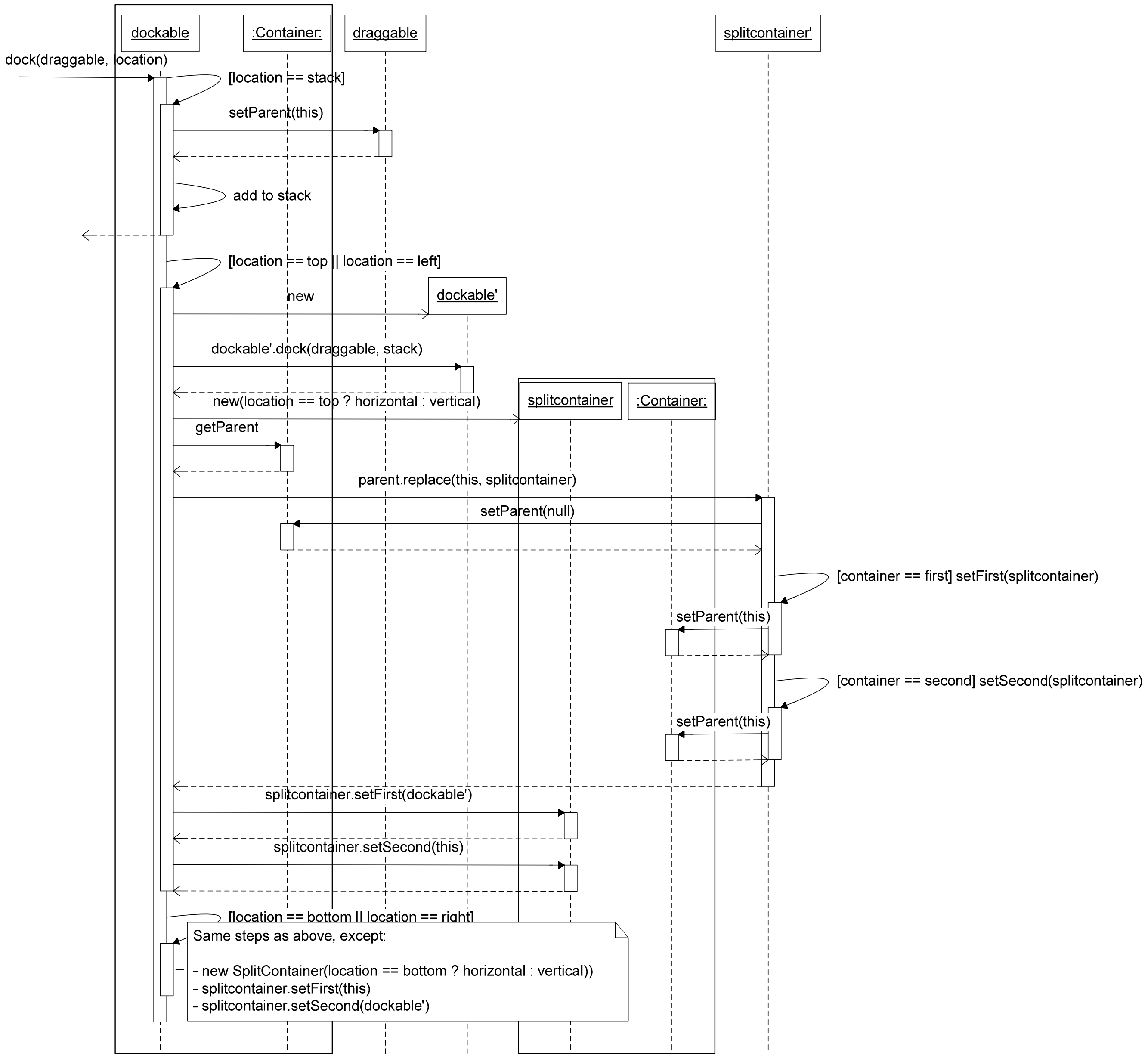

dockable
:Container:
draggable
splitcontainer'
dock(draggable, location)
[location == stack]
setParent(this)
add to stack
[location == top || location == left]
new
dockable'
dockable'.dock(draggable, stack)
new(location == top ? horizontal : vertical)
splitcontainer
:Container:
getParent
parent.replace(this, splitcontainer)
setParent(null)
[container == first] setFirst(splitcontainer)
setParent(this)
[container == second] setSecond(splitcontainer)
setParent(this)
splitcontainer.setFirst(dockable')
splitcontainer.setSecond(this)
[location == bottom || location == right]
Same steps as above, except:
- new SplitContainer(location == bottom ? horizontal : vertical))
- splitcontainer.setFirst(this)
- splitcontainer.setSecond(dockable')

Figure 9. Docking action

8. Implementation

A possible implementation of the *Drag-And-Dock* design pattern using Java and Swing can be like the following:

- *SplitContainer*, a specialization of a *javax.swing.JSplitPane* class.
- *Dockabl*e, a specialization of a *javax.swing.JTabbedPane* class, which implements a hover listener triggered by a *Mediator*, and a dock listener also triggered by a *Mediator* object.
- *Draggable*, a *JTabbedPane* tab that triggers *java.awt.event.MouseEvent* when it is being dragged or released.
- *View*, any kind of *java.awt.Component* class, e.g. a *javax.swing.JPanel*.
- *Mediator*, a listener for *Draggable* objects' drag and drop events that assesses which *Dockable* object has the mouse cursor within its boundaries and triggers a hover or dock event onto it.

Responsabilities

- Keep track of its parent *SplitContainer r*

9. Variants

One slight variant of *Drag-And-Dock* design pattern is for a *Dockable* to be itself a *Draggable* also. This means, a stack of *Draggables* within a *Dockable* could be dragged as a whole, and docked onto some other *Dockable*.

Also, depending on the programming language capabilities, the *Mediator* role can be directly carried out by each *Dockable*, thus ceasing to exist in the design structure. For that a *Dockable* has to recognize itself, if it has a *Draggable* dragged hover it, or dropped onto.

10. Known Uses

Commercial applications/solutions tend not to disclose their internal architecture or source code, so is quite hard to ascertain their use of *Drag-And-Dock* design pattern, unless they recognize so.

The exception are open source applications/solutions, in particular Java based solutions. Among them, three well known docking frameworks can be traced as having their core based on *Drag-And-Dock* design pattern. They are: VLDocking Framework [8]; FlexDock [9]; and JDock [10].

A few other solutions are expected to use *Drag-And-Dock* design pattern, but due to the lack of available architecture documentation it wasn't possible to confirm: Eclipse [3]; LidorSystems Collector [11]; SandDock [12]; DotNetMagic [13].

11. Consequences

- Independency from graphical libraries and programming languages is easy to achieve, considering that beyond minimal support for mouse events (positioning, button state), and graphical tabbed components, there's no additional restrictions/requirements on programming languages and graphical libraries.
- Flexibility to assemble any layout schema and integrate any type of content view. The five docking areas, in conjunction with the *SplitContainer*, provides broad options to assemble any rectangular based layout schema. Also, as long as each content view shares a common type of graphical component, it is guaranteed their integration.
- Consistency of layout schema upon drag and dock actions. Removing and replacing an empty *Dockable* within its parent assures the layout schema integrity because it can be ruled consistently by the mediator participant.

12. See Also

Decorator, Composite, Mediator, Observer, Personal Object Space [4], Movable Panels [5], Drag-And-Drop [6][7].

13. Credits

The authors would like to thank our shepherd Tim Wellhausen for the valuable comments, discussion promoted and feedback provided during the shepherding of this pattern. We thank also Rebecca Wirfs-Brock, Kyle Brown, Uwe Zdun, Tobias Windbrake, Christian Kohls, Ayman Mahfouz, Kay Odeyemi, and all the other participants of the writer's workshops at PLoP'2006, for the motivation, comments and suggestions for improvement they provided.

14. References

[1] Sun Microsystems, NetBeans
http://www.netbeans.org

[2] Microsoft, Visual Studio .NET
http://msdn.microsoft.com/vstudio

[3] The Eclipse Foundation, Eclipse
http://www.eclipse.org

[4] Jenifer Tidwell, Personal Object Space
http://www.mit.edu/~jtidwell/language/personal_object_space.html

[5] Jenifer Tidwell, Movable Panels
http://designinginterfaces.com/Movable_Panels

[6] AjaxPatterns, Drag-And-Drop
http://ajaxpatterns.org/Drag-And-Drop

[7] Yahoo! Design Pattern Library, Drag and Drop Modules Pattern
http://developer.yahoo.com/ypatterns/pattern.php?pattern=dragdrop modules

[8] VLSolutions, VLDocking Framework
http://www.vlsolutions.com/en/products/docking/index.php

[9] FlexDock
https://flexdock.dev.java.net

[10] JAPISoft, JDock
http://www.swingall.com/jdock.html

[11] LidorSystems, LidorSystems.Collector
http://www.lidorsystems.com/products/collector/default.html

[12] Divelements, SandDock
http://www.divil.co.uk/net/controls/sanddockwpf/

[13] Crownwood Software, DotNetMagic
http://www.crownwood.net/features_docking.html

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides
Design Patterns – Elements of Reusable Object-Oriented Software
Addison-Wesley, 1995

[15] Eric Freeman, Elizabeth Freeman
Head First - Design Patterns
O'Reilly, 2004