DockingFrames 1.0.4 - Core

Benjamin Sigg

April 11, 2008

Contents

Abstract

1 Basics

DockingFrames (or just DF) contains several key elements that must be understood by any developer. This chapter will give an overview of these elements, at the end of this chapter you'll be able to write your first application with DF.

1.1 Dockable

A `Dockable` is a small graphical panel. It contains some `JComponent` and a set of properties like an icon or a title. A `Dockable` represents a "frame", a single view of the application.

Clients will normally use the standard implementation `DefaultDockable`. `DefaultDockable` contains all the functions that are needed in any basic scenario.

Let's give an example:

```
1  DefaultDockable dockable = new DefaultDockable();
2  dockable.setTitleText( "I'm a JTree" );
3  Container content = dockable.getContentPane();
4  content.setLayout( new GridLayout( 1, 1 ) );
5  content.add( new JScrollPane( new JTree() ) );
```

There is not much to say: a `DefaultDockable` is created in line `1`, it's title set in line `2` and in lines `3-5` some component is put onto `dockable`.

1.2 DockStation

A `DockStation`, or just "station", is a parent for a set of `Dockable`s. A `DockStation` might be a `Dockable` as well, but there are exceptions. Different kinds of `DockStation`s have different behaviors.

The next example shows how some `Dockable`s might be put onto a `StackDockStation`:

```
1  StackDockStation stack = new StackDockStation();
2  stack.setTitleText( "Stack" );
3  stack.drop( new DefaultDockable( "One" ) );
4  stack.drop( new DefaultDockable( "Two" ) );
```

Some observations: `StackDockStation` is a `Dockable` as well, in line `2` the title is set. Two `DefaultDockable`s are put onto the station in lines `3,4`, the method `drop` is available in all `DockStation`s.

A list of available `DockStation`s:

StackDockStation This station uses a `JTabbedPane` (or a component behaving like one) to show exactly one of many `Dockable`s.

ScreenDockStation This station puts every `Dockable` onto its own `JDialog`. These dialogs do float around freely.

FlapDockStation A station that presents only a list of buttons to the user. If the user presses one button, a window pops up containing exactly one `Dockable`.

SplitDockStation This complex station puts its `Dockable`s in a grid. The user can modify the size of the cells, and a `Dockable` can span over multiple cells. Clients might use the class `SplitDockGrid` or `SplitDockTree` and the method `SplitDockStation.dropTree` to create an initial layout.

1.3 DockController

The `DockController` is the heart of DF. The `DockController` manages all `Dockable`s and `DockStation`s, and all objects that have an influence on them. The `DockController` seldomly does something by itself, but it "knows" where to find an object that can handle a task that has do be done.

Every `DockController` has its own realm. There can be many `DockController`s in one application, however they can't interact with each other. Normal applications will need only one `DockController`.

Every client has to register the root-`DockStation`s at the `DockController`, otherwise the station will not be able to work.

A standard use of `DockController` looks like this:

```
1  public static void main( String[] args ){
2      DockController controller = new DockController();
3
4      SplitDockStation station = new SplitDockStation();
5      controller.add( station );
6
7      station.drop( new DefaultDockable( "One" ) );
8      station.drop( new DefaultDockable( "Two" ), SplitDockProperty.NORTH
           );
9      station.drop( new DefaultDockable( "Three" ), SplitDockProperty.EAST
           );
10
11     JFrame frame = new JFrame();
12     frame.add( station.getComponent() );
13
14     frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
15     frame.setBounds( 20, 20, 400, 400 );
16     frame.setVisible( true );
17 }
```

What happens here? In line `2`, a `DockController` is created. In lines `4,5` a root-`DockStation` is created and added to `controller`. Then in lines `7-9` some `Dockable`s are dropped onto the root-station. Afterwards in lines `11-16` a `JFrame` is made visible that shows the root-station.

1.4 DockFrontend

`DockFrontend` is a layer before `DockController` and brings a set of helpful methods. Clients to not need to use a `DockFrontend`, but it can be a great aid. `DockFrontend` adds support for storing and loading the layout, and for adding a small "close"-button to each `Dockable`. It is used as a replacement of `DockController`, clients have to add the root-`DockStation`s directly to `DockFrontend` through `addRoot`. Clients can also add some `Dockable`s to the frontend using `add`, calling `setHideable` afterwards will enable the "close"-button.

2 Load and Save

The layout is the location and size of all `Dockable`s and `DockStation`s, including the relations between the elements. The ability to store this layout is often a requirement.

DF provides several ways to store the layout. There is a distinction between local and global storage methods. Local methods store the location of one `Dockable`, global methods store all locations. Local methods can never store enough information to fully restore a layout, they should only be used for hiding and restoring a single `Dockable`.

2.1 Local: DockableProperty

Every `DockStation` can create a `DockableProperty` for one of its children. A `DockableProperty` describes the location of a `Dockable` on its parent. `DockableProperties` can be be strung together to form a chain. This chain then describes a path from some `DockStation` through many other stations to a `Dockable`.

Let's look at an example:

```
1  Dockable dockable = ...
2
3  DockStation root = DockUtilities.getRoot( dockable );
4  DockableProperty location = DockUtilities.getPropertyChain( root,
       dockable );
5  dockable.getDockParent().drag( dockable );
6  root.drop( dockable, location );
```

In line `1` we get some unknown `Dockable`. In line `3` the `DockStation` which is at the top of the tree of stations and `Dockable`s is searched. Then in line `4` the location of `dockable` in respect to `root` is determined. In line `5` `dockable` is removed from its parent. And finally in line `6` `dockable` is put at its old location using the knowledge gained in lines `3` and `4`.

`DockableProperty`s are not safe to use. If the tree of stations and `Dockable`s is changed, then an earlier created `DockableProperty` might not be consistent anymore. The method `drop` of `DockStation` checks for consistency and returns `false` if a `DockableProperty` is no longer valid. The listing from above should be rewritten as:

```
1  Dockable dockable = ...
2
3  DockStation root = DockUtilities.getRoot( dockable );
4  DockableProperty location = DockUtilities.getPropertyChain( root,
       dockable );
5  dockable.getDockParent().drag( dockable );
6  if( !root.drop( dockable, location )){
7    root.drop( dockable );
8  }
```

If `location` is not valid in line `6` then `dockable` is just added at a random location.

`DockableProperty`s can be stored as byte-stream or in xml-format by a `PropertyTransformer`.

2.2 Global: DockSituation

A `DockSituation` object is a set of `DockFactory`s that are used to write or read a bunch of `DockStation`s and `Dockable`s. A `DockSituation` can handle

missing DockFactorys when reading an old layout.

2.2.1 Plain DockSituation

Clients first need to add new DockFactorys for any new kind of Dockable they introduce. Then they have to collect all root-DockStations, put them into a Map and call one of the write-methods of the DockSituation. Later they can use read to get the same Map pack (filled with new objects).

How does a DockSituation know which factory to use for which Dockable? Every Dockable has a method getFactoryID, the result of this method is a String that should match the identifier of a DockFactory. Clients using DefaultDockable can call setFactoryID to change the id.

Note: clients using ScreenDockStation must add a ScreenDockStationFactory to every DockSituation.

Bottomline: this is a painful solution which should only be used by very small applications.

2.2.2 Better DockSituation

PredefinedDockSituation is a subclass of DockSituation. It allows clients to "predefine" Dockables, meaning that DockSituation will not create new objects when loading these Dockables. A DockFactory is still required to store and load properties. Clients can predefine Dockables using the method put. They should provide a unique identifier for each Dockable they predefine.

An example:

```
1  DockStation station = ...
2  Dockable dockable = ...
3  DataOutputStream out = ...
4
5  PredefinedDockSituation situation = new PredefinedDockSituation();
6
7  situation.put( "root", station );
8  situation.put( "alpha", dockable );
9
10 Map<String, DockStation> roots =
11     new HashMap<String, DockStation>();
12 roots.put( "station", station );
13
14 situation.write( roots, out );
```

Let's analyze this code. In lines 1-3 some variables are defined, their value is given by some unknown code. In line 5 a PredefinedDockSituation is created, and in lines 7-8 station and dockable are predefined. Then in lines 10-12 the Map of root-stations is set up. Note that station can have different keys on lines 7 and 12. Finally in line 13 the layout is written into out.

Reading a layout would look like this:

```
15 DataInputStream in = ...
16 situation.read( in );
```

We get some stream in line 15, and then read the layout in line 16. The method read returns a new Map, but since all root-stations are predefined, it is safe to just forget about it. Note that dockable will also be in the tree. If dockable were not predefined, then a DockFactory would have created a new element and put at the place dockable was earlier.

2.2.3 Ignoring

Sometimes not every element has to be stored. A client can add a `DockSituationIgnore` to a `DockSituation`. The `DockSituation` will not store any element that is not approved by the `DockSituationIgnore`.

2.3 Local and Global: DockFrontend

A `DockFrontend` uses both local and global methods to store the layout. Local methods are used when a `Dockable` is made visible or invisible through `show` and `hide`. Global methods are used by `write`, `read`, `save` and `load`. A `DockFrontend` behaves much like a `PredefinedDockSituation`, either elements will be created by a `DockFactory` or the `Dockables` have to be registered through `add`.

3 Drag and Drop

Drag and drop normally means grabbing a title of a `Dockable` by pressing the mouse, moving the mouse around, and drop the `Dockable` somewhere by releasing the mouse.

3.1 Core behavior

The sourcecode used for drag and drop operations is located in the `DockRelocator`. A `DockController` normally uses a `DefaultDockRelocator` to handle all operations. Clients seldomly need to replace the `DockRelocator`, but if they do, then they have to implement a new `DockControllerFactory` and a subclass of `DockController`.

```
1  public class MyDockController extends DockController{
2    public MyDockController(){
3      super( null );
4      initiate( new DefaultDockControllerFactory(){
5        @Override
6        public DockRelocator createRelocator( DockController controller )
             {
7          return new MyDockReloactor();
8        });
9      }
10   }
11 }
```

A short review of the code: the argument `null` line `3` prevents the constructor of `DockController` to initialize the fields. In line `4` the fields are initialized using a new `DockControllerFactory`. This factory returns a new implementation of `DockRelocator` in lines `6-8`.

3.2 Remote control

Sometimes the normal mechanism for drag and drop is not enough. The drag and drop operations can be called remotely using a `RemoteRelocator` or a `DirectRemoteRelocator`. Clients can request such a remote control from the `DockRelocator` either using `createRemote` or `createDirectRemote`.

A `DirectRemoteRelocator` can be used to simulate a drag and drop operation that has no real background (like a `MouseEvent`). A client calls `init`

to start the operation, at least one time `drag` to move the grabbed `Dockable` around, and then `drop` to let the `Dockable` fall.

A `RemoteRelocator` is more tricky. The methods of a `RemoteRelocator` match the methods `mousePressed`, `mouseDragged` and `mouseReleased` of a `MouseListener/MouseMotionListener`. The methods `init`, `drag` and `drop` always tell what reaction the event caused, for example whether the operation has stopped or is going on.

3.3 Merging

When a `Dockable` is dragged over an other `Dockable`, then they have to be merged. The default behavior is to create a new `StackDockStation`, put both `Dockable`s onto that station, and then drop the station at the same place where the `Dockable`s would lie.

The creation of the station is handled by a `Combiner`, the `BasicCombiner` to be exact. Many `DockStation`s have a method that allows clients to set their own implementation of a `Combiner`. Clients can exchange the `Combiner` globally by creating a new `DockTheme`, overriding the method `getCombiner` and then registering a new instance at the `DockController` through `setTheme`. Note that all descendants of `BasicDockTheme` have a method called `setCombiner` that exchanges the `Combiner` directly without the need to override `getCombiner`.

3.4 Modes

A `DockRelocator` can have "modes". A mode is some kind of behavior that is activated when the user presses a certain combination of keys. Modes are modeled by the class `DockRelocatorMode`. It is not specified what effect a mode really has, but normally a mode would add some restrictions where to put a `Dockable` during drag and drop. `DockRelocatorMode`s can be added or removed to a `DockRelocator` by the methods `addMode` and `removeMode`.

Currently two modes are installed:

DockRelocatorMode.SCREEN_ONLY (press key *shift*) ensures that a `Dockable` can only be put on a `ScreenDockStation`. That means that a `Dockable` can be directly above a `DockStation` like a `SplitDockStation`, but can't be dropped there.

DockRelocatorMode.NO_COMBINATION (press key *alt*) ensures that a `Dockable` can't be put over another `Dockable`. That means, every operation that would result in a merge is forbidden. Also dropping a `Dockable` on already merged `Dockable`s will not be allowed.

3.5 Restrictions

Sometimes a developer wishes to restrict the set of possible targets for a drop-operation. There are multiple reasons why someone would like to do that:

- Some `Dockable` must always be visible
- Some `DockStation`s represent a special area that can only be used by some `Dockable`s

- Some Dockables can only be presented on a certain kind of DockStation

There are also a lot of ways how to achieve this goal.

- Every Dockable has two methods called accept. One of them tells the system, whether a Dockable accepts some DockStation as parent or not. The other tells whether the Dockable can be merged with another Dockable.
- Each DockStation has a method accept. This method tells whether some Dockable can become a child of the DockStation.
- And then there are DockAcceptances. A DockAcceptance has accept-methods too. These methods get a DockStation and some Dockables, and then have to decide whether the elements can be put together. Each DockAcceptance works on a global scale, and thus they are registered at the DockController through addAcceptance.

4 Themes

A DockTheme is nothing else than a LookAndFeel for DockingFrames. Each DockController can have exactly one DockTheme at any given time. The DockTheme contains a set of icons, painting code, behaviors and other stuff, that changes the way a user interacts with DF.

```
1 DockController controller = ...
2 DockTheme theme = new EclipseTheme();
3 controller.setTheme( theme );
```

The previous listing shows how easy it is to set the theme. All that needs to be done is to create the desired theme (line 2) and set it (line 3).

Several DockThemes are already part of DF. An easy way to access all of them is the method getThemes of DockUI. This method returns a set of ThemeFactory s which then can create some DockThemes.

4.1 Themes of DF

This section lists all DockThemes that are in DF and mentions their specialities, if there are any.

4.1.1 BasicTheme

The BasicTheme is a very simple implementation. Its strength is, that it shows as much features as possible. If there is the possibility to show some button, then some button is shown. If there is the possibility to add a border to a Component, then a border is added. While BasicTheme does not look very nice to the user, it does make debugging a lot easier.

4.1.2 SmoothTheme

SmoothTheme is almost the same as BasicTheme, but the titles that are shown for each Dockable have been replaced. They have now a smooth animation that is triggered whenever the focused Dockable changes.

4.1.3 FlatTheme

The reverse of `BasicTheme`, this theme does not add any borders, buttons or other decorations unless necessary. It's not a very complex theme, and easy to understand by a user.

4.1.4 BubbleTheme

A more experimental theme. It uses animations and graphical gimmicks wherever possible. This theme has some issues with performance, but it is certainly a good demonstration of the potential of the theming-mechanism.

4.1.5 EclipseTheme

The `EclipseTheme` tries to imitate the behavior of the famous Eclipse platform. It changes the behavior of DF massivly. Some properties of `EclipseTheme` can be set through the `DockProperties` as in the following example.

```
1  DockController controller = ...
2  DockProperties properties = controller.getProperties();
3  properties.set(
4      EclipseTheme.PAINT_ICONS_WHEN_DESELECTED,
5      true );
```

Let's have quick look: in line `1` we get some `DockController`. In line `2` we get access to the set of properties. In line `3-5` the property `PAINT_ICONS_WHEN_DESELECTED` is set to `true`.

There are more properties for `EclipseTheme`:

TAB_PAINTER tells how to paint tabs on the `StockDockStation`. Possible values are `ShapedGradientPainter.FACTORY`, `RectGradientPainter.FACTORY`, `DockTitleTab.FACTORY` or any other `TabPainter`.

THEME_CONNECTOR tells which kind of title and border should be used for `Dockables`, and which actions should be displayed on the tabs (actions on the tabs are always visible, other actions are only visible when a `Dockable` is selected). The value can be any `EclipseThemeConnector`.

A note: if no special theme-connector is used, then any action that is marked with the annotation `EclipseTabDockAction` will be shown on the tabs.

4.1.6 NoStackTheme

This `DockTheme` takes another theme and changes its behavior. In particular it removes some titles and ensures, that no `StackDockStations` are put in another. That ensures that merged `Dockables` are not merged again. A behavior that a user might like better then the original behavior, because it is harder to loose a `Dockable`.

The use of `NoStackTheme` is simple:

```
1  DockController controller = ...
2  DockTheme theme = ...
3  conroller.setTheme( new NoStackTheme( theme ));
```

4.2 How to write your own DockTheme

Writing a `DockTheme` is a complex matter. If you'd like to write a theme then you should make some preparations:

1. Write at least one application using DF
2. Read this document, twice
3. Download the source of DF, download the API-documentation
4. Have a look how other themes are made, `FlatTheme` is a good mix of simplicity and small features. You can learn a lot just analyzing `FlatTheme`
5. Look up any unknown interface in the API-documentation or in the source

The best way to start is by creating a subclass of `BasicTheme`. `BasicTheme` will ensure that you have someting that works and that you can modifie step by step. As you will see, `BasicTheme` has many `setXYZ`-methods, refer to step 5 of your preparations and look at the API-documentation to find out, what these methods do.

There is method called `install`. This method can be overriden (don't forget to call `super.install`) and changes any property of a `DockController`. The most often used objects by `install` are:

IconManager contains all `Icon`s that are used, the `Icon`s can be exchanged.

DockTitleManager contains factories which will create the titles for some `Dockable`s.

ActionViewConverter contains factories which create views for actions (for example a `JButton` for a `ButtonDockAction`)

DockProperties is a map for all sorts of properties, can be used as cheap distribution system for values that must be known globally

Don't forget to undo the changes in the method `uninstall`.

4.3 Colors

Since DF is a graphical user interface, colors are used often and by a all sorts of components. The most important colors are collected in a map, and components get their colors from that color-map.

There is no repository for the keys of the color-map, nor any rule how to choose them. But each class that uses the color-map, should have the annotation `ColorCodes`. All keys should be listed as arguments. The annotation is for documentation only.

Components do not ask the color-map directly for a color, they register a `DockColor`-object at the map. `DockColor` is just an observer which gets informed whenever a color in the map changes. That allows to exchange colors while the components are visible, and the changes are immediately visible.

There are different subclasses of `DockColor`. Each subclass tells by whom the color is used, for example a `TitleColor` would be used by a specific `DockTitle`.

Clients can exchange and influence the colors on different levels.

4.3.1 ColorScheme

The `BasicTheme` uses a `ColorScheme` to put up its set of colors. `ColorScheme` has one important method: `getColor`. That method gets a key for a color, and has to return the color that matches. The `ColorScheme` of `BasicTheme` can be replaced at any time calling the method `setColorScheme`. Since all `DockThemes` of the core library inherit `BasicTheme`, the possibility to use a `ColorScheme` is available for all themes.

4.3.2 ColorManager

Instead of putting all colors at once, using a `ColorScheme`, a single color can be set with the `ColorManager`. The `ColorManager` is accessible through the method `DockController.getColors`. Clients can just call the method `put` to put a new color into the color-map.

4.3.3 ColorProvider

Normally everyone that wants a color gets the same color. Put the color-map forces everyone to use a `DockColor`-object to get to its color. That object contains a lot of information, and a `ColorProvider` can use that information to return different colors for the same key.

Each subclass of `DockColor` can have its own `ColorProvider`, but only one `ColorProvider` per subclass is allowed. `ColorProviders` are added to the `ColorManager` through the `publish`-method.

A `ColorProvider` has two methods to add and remove observers. The observers need to be stored only if the provider wants to change the colors actively. If the provider only reacts to its `set`-method, then no observers need to be stored.

The `set`-method is invoked by the `ColorManager` whenever either `Color` or `DockColor` changes. The `set`-method then decides which color the `DockColor` should get. When the decision is made, it calls `setColor` on the `DockColor`.

5 Actions

A `DockAction` is an object which is related to one or many `Dockables`, they describe some action like "close a `Dockable`". Every `DockAction` has the ability to create one or many views of itself. A view might be a `JButton`, a `JCheckBox`, a `JMenuItem` or other objects.

Every `Dockable` has a list of associated `DockActions`. This list is modeled by a `DockActionSource`.

If some module wants to show the actions of a `Dockable`, it asks for the `Dockables` global `DockActionSource`, then it commands each `DockAction` to create a view that can be displayed by the module. A `JMenu` will ask for another type of view than a `DockTitle` would. So the menu might get a `JMenuItem`, the title a `JButton`.

Let's write a simple action.

```
1  public class CloseAction extends SimpleButtonAction{
2      public CloseAction(){
3          setText( "Close" );
4          setTooltip( "Removes this panel from the view" );
```

```
5           setIcon( new ImageIcon( "close.png" ));
6           setAccelerator(
7                KeyStroke.getKeyStroke(
8                   KeyEvent.VK_C,
9                   KeyEvent.CTRL_DOWN_MASK ) );
10      }
11
12      @Override
13      public void action( Dockable dockable ){
14          super.action( dockable );
15          DockStation parent = dockable.getDockParent();
16          if( parent != null )
17              parent.drag( dockable );
18      }
19  }
```

One of the predefined `DockActions` is used to implement the new kind of action. The `SimpleButtonAction` is an action that behaves like a push-down-button. In lines `3-9` some properties are set that help the user to understand and access `CloseAction`. The logic of the action is written down in lines `14-17`. In this case, a `Dockable` is removed from its parent.

And now lets add `CloseAction` to the list of actions some `Dockable` offers. Since `DefaultDockable` will be the most often used implementation of `Dockable`, the example uses a `DefaultDockable` as well.

```
1  DefaultDockable dockable = ...
2  CloseAction action = new CloseAction();
3
4  DefaultDockActionSource source = new DefaultDockActionSource(
5      new LocationHint(
6          LocationHint.DOCKABLE,
7          LocationHint.RIGHT_OF_ALL ));
8  source.add( action );
9  dockable.setActionOffers( source );
```

In line `4` a new `DockActionSource` is created. The `LocationHint` in lines `5-7` tells everyone, that the origin of `source` is a `Dockable`, and that `source` should be on the right side if the content of many `DockActionSources` are displayed in a row. The new `CloseAction` is inserted into `source` at line `8`. Then the list of actions of `dockables` is changed to `source` in line `9`. Note that lines `8` and `9` could be exchanged without any effect to the rest of the program.

5.1 Sources of DockActions

So how exactly does a module find out, which `DockActions` to show for a `Dockable`? The module uses the method `Dockable.getGlobalActionOffers` to optain a `DockActionSource`. The result of `getGlobalActionOffers` is a composite of `DockActionSources`. The children of the result come from different sources:

Local DockActionSource Every `Dockable` should have a local list of actions, this list can be accessed through `getLocalActionOffers`. Some implementations of `Dockable` have a method that allows clients to exchange that local list. For example `setActionOffers` in `DefaultDockable`.

Through the parents Most `Dockables` have one or more `DockStations` as parents. Each `DockStation` can offer `direct` (if direct parent) or `indirect` (if grandparent) `DockActionSources` for each child. Clients rarely interfere in that mechanism.

Guards `ActionGuard`s observe all `Dockable`s of a `DockController`. They can react to a `Dockable` and add additional `DockActionSource`s. An `ActionGuard` has to be made registered by calling `addActionGuard` of `DockController`.

Alternative sources The `ActionOffer` normally is the authority that creates the content of the global `DockActionSource`. A `Dockable` will get one `ActionOffer` and give that offer all `DockActionSource`s that were gathered. Then the `ActionOffer` will determine in which order the `DockActionSource`s appear and create a new composite of the sources. Clients can add new `ActionOffer`s by calling `addActionOffer` of `DockController`.

5.2 Kinds of DockActions

There are different kinds of `DockAction`, all with different behavior.

There is a list of concepts that describe the most often used kinds of actions:

Button-DockAction This kind of action reacts like a button. They can be triggered over and over again, always calling the same piece of code.

CheckBox-DockAction This kind has two states: selected and not-selected. Every time the action is triggered, the state changes.

RadioButton-DockAction Like the `CheckBox`-kind, but many `RadioButton`s are grouped together, and only one of them can be selected. Triggering a not-selected button will deselect the currently selected button.

Menu-DockAction These actions just open some pop-up menu that contains another set of actions.

DropDown-DockAction Like the `Menu`-kind, but this action also remembers which child was triggered earlier. This last triggered child can be called again without the need to open the pop-up menu.

All these concepts are implemented by the "simple" `DockAction`s:

Kind	**Action**
Button	`SimpleButtonAction`
CheckBox	`SimpleSelectableAction.Check`
RadioButton	`SimpleSelectableAction.Radio`
Menu	`SimpleMenuAction`
DropDown	`SimpleDropDownAction`

There is also a more complex series of actions, called the "grouped" `DockAction`s. The grouped actions do not store single properties like the simple actions, they store maps of properties. Each `Dockable` that is bound to a grouped action is then associated with one key, and that key is used to read the maps.

As an example: a grouped action that counts for each `Dockable` how many times the action was triggered. When testing this action you will note that certain events (like changing the `DockTheme`) set the counter back to 0. It is never safe to store information in a grouped action.

```
1  public class CountingAction extends GroupedButtonDockAction<Integer>{
2      public CountingAction() {
3          super( null );
4          setGenerator( new GroupKeyGenerator<Integer>(){
5              public Integer generateKey( Dockable dockable ) {
6                  return 0;
7              }
8          });
9          setRemoveEmptyGroups( true );
10     }
11
12     @Override
13     protected SimpleButtonAction createGroup( Integer key ) {
14         SimpleButtonAction group = super.createGroup( key );
15         group.setText( String.valueOf( key ) );
16         return group;
17     }
18
19     public void action( Dockable dockable ) {
20         String text = getText( dockable );
21         int count = Integer.valueOf( text );
22         count++;
23         setGroup( count, dockable );
24     }
25 }
```

In lines `4-8` a `GroupKeyGenerator` is set. This generator will determine the initial group of each new `Dockable`. In line `9` the fate of empty groups is defined. Empty groups are to be deleted. That is a good behavior if groups are generated automatically and the number of groups is unknown. The code in lines `13-17` defines how a new group is created. And finally in lines `20-23` the count-event is handled. The action will be triggered for `dockable`, and putting `dockable` in a new group changes the text on each view that shows the action for `dockable`.

There are a few grouped actions defined in DF:

Kind	Action
Button	`GroupedButtonDockAction`
CheckBox	`GroupedSelectableDockAction.Check`
RadioButton	`GroupedSelectableDockAction.Radio`
Menu	-
DropDown	-

Finally there is a very small action called `SeparatorAction`. This action just adds a line or space in the view, acting as a separator between other actions.

5.3 Lifecycle

Eventually each `DockAction` is instantiated and stored at a place where it can be found. While a `DockAction` enters and leaves the realm of a `DockController`, these things might happen.

1. Every time some module is going to use an action, it connects the `DockAction` with one or many `Dockable`s (the method `bind` is called). This call informs the `DockAction` that it is related to the `Dockable`s.

2. A module normally wants to show some view for an action. Therefore it calls `DockAction.createView`. It gives `createView` a `ViewTarget`. A `ViewTarget` tells what kind of view is requested, one for a menu, one for a title or even something that is defined by the client. The module also gives

an `ActionViewConverter` to `createView`. The `ActionViewConverter` is a set of factories which can create the views that are often needed. Most `DockAction`s will tell the converter what type of action they are (with an argument of type `ActionType`) and what `ViewTarget` the module requests. Then the converter will create a view matching the parameters.

3. Most views have some binding mechanism that has to be used by the module. This binding mechanism will install or uninstall some listeners when needed.

4. When a module no longer uses an action, it disconnects the `DockAction` from one or many `Dockable`s (the method `unbind` is called). That informs the `DockAction` to remove all ties to these `Dockable`s, releasing as many resources as possible.

Clients might be interested to introduce new kinds of views or new types of actions.

- When a client adds a new kind of view, it has to define a new `ViewTarget`. The client then has to register a new `ViewGenerator` for each type of action at the `ActionViewConverter`.

- When a client adds a new type of action, it has to define a new `ActionType`. The client then has to register a new `ViewGenerator` for each kind of view at the `ActionViewConverter`.

Let's have a look at an example. In the example a new kind of view and a new kind of action will be introduced.

```
1   ViewTarget<JButton> TOOLBAR =
2       new ViewTarget<JButton>( "toolbar" );
3
4   ActionType<TextAction> TEXT_ACTION =
5       new ActionType<TextAction>( "text_action" );
```

First the new kind of view `TOOLBAR` and the new type of action `TEXT_ACTION` is defined. Lines `1,2` say that the view will only consist of `JButton`s. Lines `4,5` define that the new type of action is always a `TextAction`. So the next step is to define `TextAction`.

```
1   public class TextAction implements DockAction{
2       public void bind( Dockable dockable ) {
3           // ignore
4       }
5
6       public String getContent(){
7           return "text";
8       }
9
10      public <V> V createView( ViewTarget<V> target,
11              ActionViewConverter converter, Dockable dockable ) {
12
13          return converter.createView( TEXT_ACTION, this, target,
                dockable );
14      }
15
16      public boolean trigger( Dockable dockable ) {
17          // ignore
18          return false;
19      }
20
21      public void unbind( Dockable dockable ) {
22          // ignore
23      }
24  }
```

As can be seen in line 1, `TextAction` is an implemtation of `DockAction`. Since this action is rather stupid, we can ignore most input. Line 13 is the most important line, here an `ActionViewConverter` is used to create a view for the `TextAction`. Note that the action has to pass `TEXT_ACTION`, the type of action it is.

Since the `ActionViewConverter` does not know `TOOLBAR` or `TEXT_ACTION`, a `ViewGenerator` has to be defined. In fact there are several `ViewGenerator`s necessary, one for each combination of `ViewTarget` and `ActionType`. But in this example only one new generator is written.

```
1  public class ToolbarTextAction implements ViewGenerator<TextAction,
       JButton>{
2      public JButton create(
3          ActionViewConverter converter,
4          final TextAction action,
5          final Dockable dockable ) {
6
7          String content = action.getContent();
8          JButton button = new JButton( content );
9          button.addActionListener( new ActionListener(){
10             public void actionPerformed( ActionEvent e ) {
11                 action.trigger( dockable );
12             }
13         });
14         return button;
15     }
16 }
```

Note how the generator can make use of the knowledge, that it receives a `TextAction`. In line 7 it asks for the content of the action, a method only available for `TextAction`s. The generator also connects view and action, in this case by adding a `ActionListener` to `button`.

Finally the new generator has to be made public:

```
1  DockController controller = ...
2  ActionViewConverter converter = controller.getActionViewConverter();
3
4  converter.putDefault(
5      TEXT_ACTION,
6      TOOLBAR,
7      new ToolbarTextAction() );
```

There are several methods called `putX` in `ActionViewConverter`. `putDefault` should be used for new generators, `putTheme` is only used by `DockTheme`s, and `putClient` can be used by any client to override values that were set by `putDefault` or `putTheme`.

A module that needs a view for `TOOLBAR` would later call code that looks like this:

```
1  ActionViewConverter converter = ...
2  TextAction action = ...
3  Dockable dockable = ...
4
5  JButton button = converter.createView( action, TOOLBAR, dockable )
```

6 Titles

A `DockTitle` is a `Component` that shows the icon, title-text, actions and/or other information related to a `Dockable`. A drag and drop operation is most often initiated by the mouse grabbing a `DockTitle`.

6.1 New titles

There is not much help to offer for developers which want to write a new kind of title. However there are some classes which might help:

AbstractDockTitle offers all the features a `DockTitle` should have, subclasses can override `paintBackground` to add their own painting code.

BasicDockTitle paints some gradient as background. Clients can change these colors.

ButtonPanel a `Component` that can display a set of `DockAction`s. Clients just invoke `set(Dockable)` to show the actions of a particular `Dockable`. If there is not enough space for all `DockAction`s, then `ButtonPanel` can use an additional pop-up for the abundant actions.

6.2 Lifecycle

If a module wants to show a `DockTitle` for a `Dockable`, what has it to do? First a module needs to define what kind of `DockTitle` it wants to show. For that it needs the `DockTitleManager` which is available through a `DockController`. The module then calls `getVersion(String,DockTitleFactory)` to optain a `DockTitleVersion`. A `DockTitleVersion` describes the kind of a `DockTitle`.

Later when the module gets a `Dockable`, it invokes `Dockable.getDockTitle` with its `DockTitleVersion`. A `Dockable` can decide on its own how to create the title, but most `Dockable`s will simply call `DockTitleVersion.createDockable`.

If the module got a title (and not `null`), it binds the title to its `Dockable` calling `Dockable.bind(DockTitle)`. The `DockController` will handle any other binding operations that need to be done.

When a module no longer needs a `DockTitle`, it unbinds the title through `Dockable.unbind(DockTitle)`.

Clients can influence the `DockTitle` that is used for a `Dockable` in two ways:

- They override `Dockable.getDockTitle` and return any title they like.
- They install a new `DockTitleFactory` at the `DockTitleManager`. Clients can do this by invoking `registerClient(String,DockTitleFactory)`.