DockingFrames 1.0.6 - Core

Benjamin Sigg

December 26, 2008

Contents

Abstract

`DockingFrames` is an open source Java Swing framework. This project allows to write applications with floating panels, meaning that the user can freely choose where to place the panels.

`DockingFrames` is divided into two projects, `Core` and `Common`. This document only covers `Core`, `Common` has its own guide.

The goal of this document is to provide any developer with a basic understanding of `DockingFrames`. One will not be able to rewrite the project after reading this document, but one will be able to start digging in the source.

1 Notation

This document uses various notations.

Any element that can be source code (e.g. a class name) and project names are written monospaced like this: `java.lang.String`. The package of classes and interfaces is rarely given since almost no name is used twice. The packages can be easily found with the help of the generated api documentation (JavaDoc).

Tipps and tricks are listed in boxes.

Important notes and warnings are listed in boxes like this one.

Implementation details, especially lists of class names, are written in boxes like this.

These boxes explain *why* some thing was designed the way it is. This might either contain some bit of history or an explanation why some akward design is as bad as it first looks.

2 Basics

The basic idea of `Core` is to have one object that controlls the framework, one object for each floating panel and one object for each area where a floating panel can be docked.

The controller is a `DockController`, the floating panels are `Dockables` and the dock-areas are `DockStations`.

2.1 Hello World

Let's start with a simple hello world. This application uses the three basic components, the example consists of valid code and can run:

```java
1  import javax.swing.JFrame;
2
3  import bibliothek.gui.DockController;
4  import bibliothek.gui.dock.DefaultDockable;
5  import bibliothek.gui.dock.SplitDockStation;
6  import bibliothek.gui.dock.station.split.SplitDockGrid;
7
8  public class HelloWorld {
9      public static void main( String[] args ) {
10         DockController controller = new DockController();
11
12         SplitDockStation station = new SplitDockStation();
13         controller.add( station );
14
15         SplitDockGrid grid = new SplitDockGrid();
16         grid.addDockable( 0, 0, 2, 1, new DefaultDockable( "N" ) );
17         grid.addDockable( 0, 1, 1, 1, new DefaultDockable( "SW" ) );
18         grid.addDockable( 1, 1, 1, 1, new DefaultDockable( "SE" ) );
19         station.dropTree( grid.toTree() );
20
21         JFrame frame = new JFrame();
22         frame.add( station.getComponent() );
23
24         frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
25         frame.setBounds( 20, 20, 400, 400 );
26         frame.setVisible( true );
27     }
28 }
```

What happens here? In line `10` a `DockController` is created. The controller will handle things like drag and drop. All elements will be in his realm. In line `12` a new `DockStation` is created and in line `13` this station is registered as root station at the `DockController`.

Then in line `15-19` a few children for `station` are generated. To set the layout of those children a `SplitDockGrid` is used. `SplitDockGrid` takes a few `Dockables` and their position and puts this information into a form that can be understood by `SplitDockStation` (line `19`). It would be possible to add the `Dockables` directly to the station, but this is the easy way.

In line `21` a new frame is created and in line `22` our `DockStation` is added to the frame.

More demonstration applications can be found in the big archive-file of `DockingFrames`. Each demonstration concentrates its attention on one feature of the framework.

2.2 Dockable

A `Dockable` represents a floating panel, it consists at least of some `JComponent` (the panel it represents), some `Icon` and some text for a title. Each `Dockable` can be dragged by the user and dropped over a `DockStation`.

Clients can implement the interface `Dockable`, but it is much less painful just to use `DefaultDockable`. A `DefaultDockable` behaves in many ways like the well known `JFrame`: title, icon and panel can be set and replaced at any time.

A small example:

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

If implementing `Dockable`, pay special attention to the api-doc. Some methods have a rather special behavior. It might be a good idea to subclass `AbstractDockable` or to copy as much as possible from it.

A careful analysis of `Dockable` reveals that there is no way for applications to store their own properties within a `Dockable` (unless using a subclass...). There are two reasons for this. First: if only using the default implementation, there is no need for clients to track these properties, store and load them or to delete them once they are no longer used. It is the responsibility of the framework to do so. Second: No special component within the framework or programming technique gets an unfair advantage over others, everything has to be designed in a way that it can work with any new, unknown, crazy other otherwise unexpected kind of `Dockable`.

2.3 DockStation

`Dockable`s can never fly around for themself, they need a `DockStation` as anchor point. The relationship between `DockStation` and `Dockable` can best be described as parent-child-relationship. A `DockStation` can have many children, but a `Dockable` only one parent.

There are some classes which are `DockStation` and `Dockable` at the same time. They allow to build a tree of `DockStation`s and `Dockable`s. The number of such trees is *not* limited to one.

There are different kind of `DockStation`s, each kind has its unique behavior and abilities.

StackDockStation The children are organized like on a `JTabbedPane`. Only one child is visible, but another can be made visible by clicking some button.

SplitDockStation The children are organized like in a tree of `JSplitPane`s. All children are visible and the user can change the (relative) size of the `Dockable`s.

FlapDockStation Much like `StackDockStation` but the one visible child pops up in its own window. This station can also show no `Dockable` at all.

ScreenDockStation Shows each child in its own window.

Clients can implement new `DockStation`s. But be warned that the interface contains many methods and a lot of them require a lot of code. Don't expect to write less than 1000 lines of code.

A small example that builds 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.

`DockStation`s are the most complex classes within the framework, they are also among the most important classes. It is very uncommon to subclass them or to write new ones. If you think you need to subclass a `DockStation`, be sure to have explored all other options.

2.4 DockController

A `DockController` manages almost all the interactions between `Dockable`s and `DockStation`s. A `DockController` seldomly does a task by himself, but it always knows how to find an object that can do the task.

There can be more than one `DockController` in an application. Each controller has its own realm and there is no interaction between controllers. Most applications will need only one `DockController`.

Clients need to register the root of their `DockStation`-`Dockable`-trees. They can use the method `add` of `DockController` to do that. All children of the root will automatically be registered as well. If a `DockStation` is not registered anywhere, it just does not work properly. For `Dockable`s one could say that registration equals visibility. A registered `Dockable` can be seen by the user, an unregistered not.

`DockController` uses other classes to handle tasks. Many of these classes can be observed by listeners. An incomplete list:
`DockRegister`: a list of all `Dockable`s and `DockStation`s.
`DockRelocator`: handles drag and drop operations, can create a `Remote` to play around without user interaction.
`DoubleClickController`: detects double clicks on `Dockable`s or on components which represent `Dockable`s.
`KeyBoardController`: detects `KeyEvent`s on `Dockable`s or on components which represent `Dockable`s.

Never forget to register the root-`DockStation`(s) at the `DockController` using the method `add`.

Why not just one `DockController` implemented as singleton? A singleton would make many interfaces simpler, eliminating all the code where the controller is handed over to even the smallest object. On the other hand there is absolutely no reason to limit oneself to only one object and there are applications which need more than one controller. In the end not using a singleton just gives more flexibility.

2.5 DockFrontend

`DockController` only implements the basic functionallity. While this allows developers to add new exciting shiny customized features, it certainly doesn't help those developers which just want to use the framework.

The class `DockFrontend` represents a layer before `DockController` and adds a set of helpful methods. Especially a "close"-button and the ability to store and load the layout are a great help. `DockFrontend` replaces `DockController`, clients should add the root-`DockStation`s directly to the frontend, not to the controller. They can use the method `addRoot` to do so.

`DockFrontend` adds a few nice features but not enough to write an application without even bothering to have a look at `DockingFrames`. Developers which can live with not having absolute control over the framework should use `Common`. `Common` adds all those features which make a docking-framework complete, e.g. a "minimize"-button

`DockFrontend` was written long after `DockController`. For the most part it just reuses code that already exists. It would be possible to write two applications with exact the same behavior once with and once without `DockFrontend`. The only thing that `DockFrontend` adds to the framework is a central hub where all the important features are accessible and a good set of default-values for various properties of the framework.

Use the methods called `setDefault...` to set default values for properties which will be used for `Dockables`. I.e. whether `Dockables` are hideable or not.

2.5.1 Close-Button

In order to show the close-button clients need first to register their `Dockables`. The method `addDockable` is used for that. Each `Dockable` needs a unique identifier that is used internally by `DockFrontend`. Later clients can call the method `setHideable` to show or to hide the close-button.

By calling the method `setShowHideAction` clients can make the buttons invisible for all `Dockables`, note however that the `Dockables` hideable-property is not affected by this method.

If clients want to control whether a `Dockable` can be closed, they should add a `VetoableDockFrontendListener` to the `DockFrontend`. This listener will be informed before a `Dockable` is made invisible and allows to cancel the operation.

> Why is the close-button not part of the very core of the framework? For one because the very core works on abstract levels and should not be made more complex with special cases like this button. There are also different implementations of this button and not all perform the same actions when pressed (this is especially true when using `Common`).

2.5.2 Storing the layout

The methods `save`, `load`, `delete` and `getSettings` are an easy way to store and load the layout. This mechanism will be explained in detail in another chapter.

3 Load and Save layouts

The layout of an application means the position, size and relationships of all the `Dockables` and `DockStations`. To store this layout on a harddrive and later to load it again is a great help for the user, he does not need to setup the layout over and over again.

`DockingFrames` distinguishes between local and global layout information. Local information only describes the relationship between one `Dockable` and its parent, global information describes whole trees of elements. There are no algorithms which recreate a whole layout from a set containing local information, but there are also no algorithms which can place a `Dockable` in the tree using global information. So both kinds of data have their use.

3.1 Local: DockableProperty

Every `DockStation` can create a `DockableProperty`-object for one of its children. This `DockableProperty` contains the position and size of one child.

Some `DockStations` are also `Dockables`. Those stations are not only able to create `DockableProperties` for their children but their parents can create a property for them. These two properties can be strung together to form a chain describing the position of a grand-child on its grand-parent.

3.1.1 Creation

How to create a `DockableProperty`? One way is of course just to create new objects using `new XYProperty(...)`. The other way is to retreive them from some `DockStation`s and `Dockable`s:

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

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.

There are five `DockableProperties` present in the framework.

StackDockProperty for `StackDockStation`, contains just the index of the `Dockable` in the stack.

FlapDockProperty for `FlapDockStation`, contains index, size and whether the `Dockable` should hold its position when not focused.

ScreenDockProperty for `ScreenDockStation`, contains the boundaries of a `Dockable` on the screen.

SplitDockProperty for `SplitDockStation`. This deprecated property contains the boundaries of a `Dockable` on the station.

SplitDockPathProperty also for `SplitDockStation`. This new property contains the exact path leading to a `Dockable` in the tree that is used internally by the `SplitDockStation`.

3.1.2 Usage

How to apply a `DockableProperty`? Every `DockStation` has a method `drop` that takes a `Dockable` and its position. That might look like this:

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

In lines `1-3` some elements that were stored earlier are described. In line `5` we try to drop `dockable` on `root`, if that fails we just drop it somewhere (line `6`).

`DockablePropertys` are not safe to use. If the tree of stations and `Dockable`s changes, 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.

Always check the result of `drop`, if it is `false` then the operation was canceled by the station because the property is invalid.

3.1.3 Storage

`DockablePropertys` can be stored either as byte-stream or in xml-format by a `PropertyTransformer`. A set of `DockablePropertyFactories` is used by the transformer to store and load properties. The factories for the default properties are always installed. If a developer adds new properties then he should use the method `addFactory` to install new factories for them.

If using `DockFrontend` the method `registerFactory` can be used to add a new `DockablePropertyFactory`. This factory will then be used by global transformer of the frontent.

3.2 Global: DockSituation

The layout of a whole set of `Dockables` and `DockStations` can be stored with the help of a `DockSituation`. A `DockSituation` is a set of algorithms that transform the layout information from one format into another, e.g. from the dock-tree (built by stations and `Dockables`) to an xml-file. A `DockSituation` uses various factories to transform one format into another.

3.2.1 Basic Algorithms

Global layout information comes in four formats:

dock-tree format The set of `Dockables` and `DockStations` as they are seen by the user.

binary format A file containing binary data. This file is normally written by a `DataOuputStream` and read by a `DataInputStream`.

xml format A file containing xml. To write and read such a file the class `XIO` is used.

layout-composition format An intermediate format that consists of a set of `DockLayoutCompositions`. These objects are organized in a tree that has the same form as the dock-tree.

To convert one format into another a `DockSituation` is used. If converting from `a` to `b` then a `DockSituation` will always first convert `a` to `layout-composition` and then `layout-composition` to `b`.

`DockSituation` always creates new files or new objects. In its basic form it is not able to reuse existing elements.

A DockSituation uses different factories and strategies for these conversions:

DockFactory These factories are responsible to load or store the layout of a single Dockable or DockStation. Like DockSituation they need to support four formats. For one the dock-element they store or read, then binary- and xml-format and finally some object as intermediate formate. They are free to choose any kind of object as intermediate format.

AdjacentDockFactory They function the same way as DockFactories but can be used for arbitrary dock-elements. AdjacentDockFactories are used to store additional information about elements, that can, but does not have to be, layout information.

MissingDockFactory These are used when another factory is missing. The MissingDockFactory can try to read the xml-format or binary-format and convert it to the intermediate format.

DockSituationIgnore This strategy allows a DockSituation to ignore dock-elements when storing the layout. That can be helpful if e.g. an application has Dockables which show only temporary information that will be lost on shutdown anyway.

A DockSituation can handle missing factories when reading xml or binary format. It first tries to use a MissingDockFatory to read the data, if that fails it either throws away the data (for AdjacentDockFactories) or stores the data in the layout-composition as "bubble" in its raw format. These "bubbles" can be converted later when the missing factories are found.

A DockLayoutComposition contains a lot of information. First of all a list of children to build the tree. Then a list of DockLayouts which represent the information from AdjacentDockFactories. Each DockLayout contains a unique identifier for the factory and the data generated by the factory. Finally a DockLayoutComposition contains a DockLayoutInfo which represents the data of or for a DockFactory. A DockLayoutInfo either contains a DockLayout (the normal case) or some data in xml or binary format. The later case happens if a factory was missing while reading a file, the information gets stored until it can be read later.

The method fillMissing can be used to read "bubbles" in raw format. The method estimateLocations can be used to build DockableProperties for the elements. These are the positions were the elements would come to rest if the layout information were converted into a dock-tree.

3.2.2 Basic Usage

How is a DockSituation utilized in order to load or store the layout of an application?

Each `Dockable` and each `DockStation` have a method `getFactoryID`. This method returns an identifier that has to match the unique identifier that is returned by the method `getID` of `DockFactory`. So the first step in using a `DockSituation` will always be to make sure that for any identifier a matching `DockFactory` is available. Clients will call the method `add` of `DockSituation` to do so.

> Default factories are installed for `DefaultDockable`, `SplitDockStation`, `StackDockStation` and `FlapDockStation`.

> The `ScreenDockStationFactory` for `ScreenDockStation` is not installed per default. This factory requires a `WindowProvider` to create the station, and since this provider cannot be guessed by `DockSituation` the factory is missing. Clients have to add `ScreenDockStationFactory` manually.

Afterwards clients just have to call `write` or `writeXML` to write a set of `DockStations` and their children. Clients can later call `read` or `readXML` to read the same map of elements. Note that every call to `read` or `readXML` will create a new set of `Dockable`- and `DockStation`-objects.

Let't give an example how to write an xml file:

```
1  try{
2      JFrame frame = ...
3      DockStation root = ...
4
5      DockSituation situation = new DockSituation();
6      situation.add( new ScreenDockStationFactory( frame ) );
7      situation.add( new MySpecialFactory() );
8
9      Map<String, DockStation> map = new HashMap<String, DockStation>();
10     map.put( "root", root );
11
12     XElement xlayout = new XElement( "layout" );
13     situation.writeXML( map, xlayout );
14
15     FileOutputStream out = new FileOutputStream( "layout.xml" );
16     XIO.writeUTF( xlayout, out );
17     out.close();
18 }
19 catch( IOException ex ){
20     ex.printStackTrace();
21 }
```

On line `2` the main-frame of the application is given and on line `3` the applications root `DockStation`. The first step is to create a new `DockSituation` on line `5` and add the missing `ScreenDockStationFactory` on line `6`. Then other factories that are not part of `DockingFrames` but the application itself can be added like on line `7`. On lines `9, 10` a map with all the root-stations of the application is built up. Then on line `12` we prepare for writing in xml-format by creating a `XElement`. The situation converts the dock-tree to xml-format in line `13`. Finally on lines `15-17` the xml-tree is written into a file "layout.xml".

The next example shows how reading from binary format can look like:

```
try{
```

```
2      JFrame frame = ...
3
4      DockSituation situation = new DockSituation();
5      situation.add( new ScreenDockStationFactory( frame ) );
6      situation.add( new MySpecialFactory() );
7
8      FileInputStream fileStream = new FileInputStream( "layout" );
9      DataInputStream in = new DataInputStream( fileStream );
10
11     Map<String, DockStation> map = situation.read( in );
12
13     in.close();
14
15     SplitDockStation station = (SplitDockStation)map.get( "root" );
16     frame.add( station.getComponent() );
17 }
18 catch( IOException ex ){
19     ex.printStackTrace();
20 }
```

What happens here? In line `2` the main frame of the application is defined. In lines `4-6` a `DockSituation` is set up. In lines `8, 9` a file is opened. In line `11` that file gets read by the `DockSituation` and a map that was earlier given to `write` is returned. In line `15` the fact that `map` was earlier given to `write` is used to guess that there is a `SplitDockStation` with key "root" in the map. Finally in line `16` that station is put onto the main-frame which now shows the new elements.

3.2.3 Extended Algorithms

The major drawback of the basic algorithms is that they always create new `Dockable`s and `DockStation`s. As a result it is nearly impossible to just change the layout while an application is running, a layout can only be loaded on startup. `PredefinedDockSituation` builds upon `DockSituation` and extends the algorithms in a way that the can reuse existing dock-elements.

The extended version uses a special `DockFactory`, called `PreloadFactory`, that is wrapped around the factories providied by the client. Writing does not change much, the `PreloadFactory` delegates the work just to the original `DockFactory`. Reading is more interesting. The `PreloadFactory` forwards the dock-element to reuse to the the original `DockFactory` which then updates the layout of the element.

A side effect of this implementation is, that for the basic `DockSituation` no factories seem ever to be missing. In fact the issue of missing factories is just moved to the `PreloadFactory`. The `PreloadFactory` can however store data in its raw format if necessary.

> A `PreloadFactory` uses a `PreloadedLayout` as intermediate format. This `PreloadedLayout` contains the unique identifier of the original `DockFactory` and a `DockLayoutInfo`. The `DockLayoutInfo` contains either data in raw format or in the intermediate format of the original factory.

What happens if a `PredefinedDockSituation` finds layout information for an element, has all the necessary factories but not the element itself? The default behavior is to ignore the information. However it is possible to use backup-`DockFactories`. These backup factories will create new elements if they were missing. They are also used when reading raw format and the original factory is missing. These backup factories are added through `addBackup`, they have to use a `BackupFactoryData` as intermediate format.

Note that the `MissingDockFactory` of `DockSituation` is not used for elements that were predefined on writing, because for those elements the `PreloadFactory` - which is never missing - was used.

The existence of these two sets of algorithms, basic and extended, lays in the history of `DockingFrames`. First the basic algorithms were written. They did their job well for small applications. But when applications began to grow it became evident that their were not sufficient. Instead of rewriting them another layer was added. The division in two sets of algorithms has also the advantage of reduced complexity.
The recovery mechanisms for missing factories were introduced for version 1.0.7. They are not yet satisfying and it is likely that they will be changed again in future versions.

3.2.4 Extended Usage

`PredefinedDockSituation` is used in the same way as `DockSituation`. The only difference is the possibility to predefine elements. The method `put` can be used for that. This method expects a unique identifier for any new element.

An example can look like this:

```
1  DockStation rootStation = ...
2  Dockable fileTreeDockable = ...
3  Dockable contentDockable = ...
4
5  PredefinedDockSituation situation = new PredefinedDockSituation();
6
7  // setup situation {...}
8
9  situation.put( "root", rootStation );
10 situation.put( "file-tree", fileTreeDockable );
11 situation.put( "content", contentDockable );
12
13 // read or write {...}
```

In lines `1-3` some `DockStations` and `Dockables` are defined. These are the elements that are always present and need not to be recreated when loading a layout. In line `5` a new `PredefinedDockSituation` is created. Then the basic setup (adding factories, ...) is done in line `7`. In the lines `9-11` the predefenied elements are added to the situation. For each of them a unique identifier is choosen. Finally in line `13` we can either write or read the layout.

Any `String` can be used as unique identifier. Small identifiers with no special characters are however much less likely to attract any kind of trouble.

3.3 DockFrontend

`DockFrontend` offers storage for local and for global layout information. Clients need to register their `Dockable`s through `addDockable` if they want access to the full range of storage-features.

3.3.1 Local

Whenever `hide` is called for a registered `Dockable` its local position gets stored. If later `show` is called this position is reapplied and the element shows up at the same (or nearly the same) location it was earlier. This local information can also be stored in xml- or binary-format. The methods `write`, `writeXML`, `read` and `readXML` will take care of this.

3.3.2 Global

`DockFrontend` internally uses a `PredefinedDockSituation` to store the global layout. All root-`DockStation`s and all registered `Dockable`s are automatically added to this situation. The global layout can either be stored on disk using the methods `write`, `writeXML`, `read` and `readXML` or it can be stored in memory. It is possible to store more than just one layout in memory and allow the user to choose from different layouts. There are methods to interact with the layouts in memory:

save Saves the current layout in memory. Clients can provide a name for the layout or use the name of the last loaded layout.

load Loads a layout. The name of the layout is used as key.

delete Deletes a layout from memory.

getSettings Gets a set of names for the different layouts.

getCurrentSetting Gets the name of the layout that is currently loaded, can be `null`.

setCurrentSetting If there is a layout with the name given to this method than that layout is loaded. Otherwise the current layout gets saved with the new name.

The layouts that are in memory can be stored persistent as well.

3.3.3 Missing Dockables

The default behavior of `DockFrontend` is to through away information for missing `Dockable`s. It is however possible to change that behavior.

If data needs to be stored for a missing `Dockable` then `DockFrontend` uses an "empty entry". Clients can define new empty entries by invoking the method `addEmpty`. Existing entries can be removed with `removeEmpty`, with `listEmpty` all empty entries can be accessed. Once an entry has been marked as "empty" it can switch between filled and empty as many times as necessary without loosing its layout information. The `DockFrontend` can even store data in raw xml or binary format and convert this data later once an appropriate `DockFactory` becomes known.

> "Empty entries" are best to be used if a client already knows the identifiers of all the `Dockables` that can eventually be registered at the `DockFrontend`.

Another way is to register backup-`DockFactories` by calling the method `registerBackupFactory`. These factories will create new `Dockables` which are then automatically registered.

> A backup-factory is the strongest weapon against missing information. If there is a possibility to use them, use them.

And finally there is the `MissingDockableStrategy` which can be set using `setMissingDockableStrategy`. This strategy enables or disables to automatic processes.

- It allows to create "empty entries" automatically. There are two methods `shouldStoreShown` and `shouldStoreHidden` which have to check the identifiers and to return `true` to allow a new empty entry.
- It allows to use new `DockFactories` as soon as they become known. Normally `DockFrontend` does not change the layout without the explizit command from a client (by invoking `setSetting` directly or indirectly). If `shouldCreate` returns `true` however `DockFrontend` will update the layout as soon as enough information is available to do so.

> `MissingDockableStrategy` should be used when no information about what is missing is available. It allows to run a "do whatever is possible"-strategy.

> If a strategy allows to store anything and a client often uses different identifiers for their `Dockables`, then layouts will start to grow and never stop. Don't forget to delete outdated information.

> The interface `MissingDockableStragey` offers two default implementations: `DISCARD_ALL` and `STORE_ALL`. The first implementation is set as default and allows nothing, the second one allows everything.

4 Actions

All Dockables can be associated with some actions. An action normally appears as some kind of button in the title of a Dockable, they can however appear at other places as well. There are different types of actions, some may behave like a JButton others like a JCheckBox, clients can add new types.

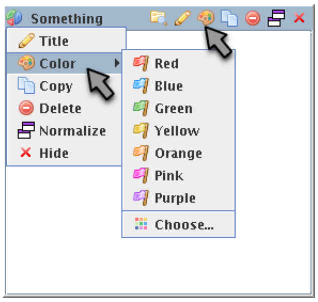

Figure 1: A Dockable with a few DockActions in its title and on a popup menu. The action marked by an arrow is the same object just shown in different views.

Actions are represented by the interface DockAction. Each Dockable has a list of them represented by a DockActionSource.

If some component wants to show some actions it firsts asks a Dockable for its global DockActionSource. It then asks each DockAction of that list to create a view that fits to the component. A title will ask for another kind of view than a menu. At any time actions can be added or removed from the DockActionSource and any component showing actions will react on these events.

The interface DockAction is quite simple. Two methods to install (bind) and to uninstall (unbind) the action. One method to create new views (createView) and one method to trigger an action programatically (trigger). More useful are the many subclasses and subinterfaces. StandardDockAction introduces icons, text and tooltip. Five subinterfaces for StandardDockAction exist and for all of them a default-view is provided.

There are three levels in the design of DockAction and its subclasses. First there is DockAction which allows almost any kind of Component to be used as view. Second there are subinterfaces for the standard tasks, the framework provides views for them. Third are real implementations of the second-level interfaces. Some interfaces are implemented in more than one action for different styles of aplication organization.

4.1 Show Actions

Assuming one has a DockAction (more about different kind of actions is in the next chapter) how can the framework be advised to show it?

4.1.1 List of Actions

DockActions never travel alone in this framework. They always travel with other actions in a DockActionSource. Actions can be added or removed from DockActionSources at any time and modules showing actions will react on this.

Most methods of DockActionSource can be understood without explanation. The method getLocationHint is an exception. It returns a LocationHint which is used to order several DockActionSources into a list (and treat them as one big DockActionSource). Clients which implement an ActionOffer can also introduce new kind of LocationHints.

> LocationHints consists of an Origin and a Hint. The hint tells the preferred location in respect to other elements, the origin are used if multiple hints collide. New Hints and Origins can be written.

4.1.2 Source of Actions

Actions haf different sources, each kind of source has a specific purpose.

- The **local action source** is part of every Dockable. This source is accessed through getLocalActionOffers. If AbstractDockable or a subclass like DefaultDockable is used then setLocalActionOffers allows to quickly set and exchange the actions. This source of actions should be used for actions that are closely linked with some Dockable.

- ActionGuards can add actions to every Dockable. An ActionGuard is added to a DockController through addActionGuard. Its method react will be called whenever the actions of a Dockable are searched. If react returns true them the method getSource is called which adds the actions. This source of actions is intended either for general purpose action or for actions which need a special position in the list of actions (like a close-action needs to be at the very right end).

- Every DockStation can add **direct** and **indirect action offers** to its children. For this DockStation has two methods getDirectActionOffers and getIndirectActionOffers. **Direct action offers** are used only for true children, indirect action offers can be applied to grandchildren as well. These sources of actions is intended for actions that are linked to a DockStation, like the maixmimze-action that can be seen on a SplitDockStation.

Two mechanisms are responsible for collecting all the actions form these different sources and o put them into one list. Clients can adjust these mechanisms even to a point where they no longer collect actions but introduce their own actions.

- Every `DockController` has at least one `ActionOffer`. An `ActionOffer` has two methods, `interested` tells whether the offer is interested in managing a certain `Dockable` and `getSource` collects the actions of an interesting `Dockable`. The primary function of an `ActionOffer` is to order the various sources. It is up to the offer to decide how to actually do the sorting. The default `ActionOffer` uses the `LocationHint` which is attached to every `DockActionSource`.

  Clients can use `addActionOffer` and `setDefaultActionOffer` to change the offers of a `DockController`. The public method `listOffers` then advises the controller to use its offers, clients however should not call this method directly. They should call `getGlobalActionOffers` of `Dockable`.

- Modules which need a list of actions call `getGlobalActionOffers` from `Dockable`. This method is the ultimate piece of code which decides what to show. If need by this method can ignore anything else that has been said in this chapter and introduce its very own mechanism to collect actinos. Most `Dockables` however will create a field holding a `HierarchyDockActionSource`. This special source observes the hierarchy of a `Dockable` and changes its content automatically. `Dockables` using `HierarchyDockActionSource` should `bind` the source. They need to call `update` if their own local action source is exchanged.

It is generally a bad idea to write `DockActionOffers` or `getGlobalActionOffer` methods which do not just collect actions. There are already mechanisms to introduce `DockActions` and they should suffice for every possible situation.

4.2 Standard Actions

There are a number of standard actions in the framework. Clients can either subclass them or instantiate and add listeners to them. A user would put the actions into six groups:

Button If the user clicks this action then always the same happens. The interface `ButtonDockAction` collects all the buttonlike actions.

Checkbox When triggered it changes some property from `true` to `false` or from `false` to `true`. All actions with this behavior implement the interface `SelectableDockAction`.

Radiobutton Like a group of checkboxes, but only one radiobutton can be selected within that group. Like checkboxes all these actions are represented by `SelectableDockAction`. Several radiobuttons can be linked together with the help of a `SelectableDockActionGroup`.

Menu A menu just contains a list of other `DockActions`. These other actions are normally hidden and only shown if the user wants to see them. Menus are implementing the interface `MenuDockAction`.

Drop-down-button Like a menu but the last triggered action can be triggered again without opening the menu. The interface `DropDownAction` represents these special menus.

Separator A separator just is a line, a graphical element to divide a set of actions into subsets. Separators are implemented through the class `SeparatorAction`.

4.2.1 Simple actions

Simple actions are a set of classes that implement the various action-interfaces. These simple actions do not have any advanced features and should be quite simple to use. An example might be the following code:

```
1  public class ExampleAction extends SimpleButtonAction{
2      public ExampleAction() {
3          setText( "Run..." );
4          setIcon( new ImageIcon( "example.png" ) );
5          setTooltip( "Run the example" );
6      }
7
8      @Override
9      public void action( Dockable dockable ) {
10         System.out.println( "kabum" );
11     }
12 }
```

Here the class `SimpleButtonAction` is used. The action is subclassed by `ExampleAction`. In lines `3-5` properties like the icon are set. The subclass overrides the method `action` (lines `9-11`) which is invoked every time when the user presses the button.

The available simple actions are:

- `SimpleButtonAction`: For creating buttons. Can either be subclassed (like in the example above) or just instanciated. Clients can add instances of the well known `ActionListener`s which will be invoked when the user presses the button. Exaclty like a `JButton`.
- `SimpleSelectableAction.Check` and `SimpleSelectableAction.Radio`: For creating checkboxes and radiobuttons. Clients can add instances of `SelectableDockActionListener` to be informed whenever the state of the action changes. A `SelectableDockActionGroup` can be used to make sure that only one action out of a set of actions is selected at any time.
- `SimpleMenuAction`: For creating menus. The method `setMenu` takes a `DockActionSource` and the content of this source will be shown.
- `SimpleDropDownAction`: For creating drop down menus. Has methods to get and set the selection, and methods to add or remove actions from the menu.

4.2.2 Group actions

Group actions are `DockAction`s that can be used for many `Dockable`s at once even with different properties for each `Dockable`. To be more precise, a `GroupKeyGenerator` will assign a key to each `Dockable`. If any view asks the action for a property (like the icon) this key will be used to search the property in a map. All the group actions extend the class `GroupedDockAction`.

Let's have a look at an example. The following action behaves like a checkbox. Its unique feature is the text that changes if the selected-state changes.

```
1  import bibliothek.gui.Dockable;
2  import bibliothek.gui.dock.action.actions.GroupKeyGenerator;
3  import bibliothek.gui.dock.action.actions.GroupedSelectableDockAction;
4
5  public class ExampleGroupAction extends
6              GroupedSelectableDockAction.Check<Boolean> {
7      public ExampleGroupAction(){
8          super( new GroupKeyGenerator<Boolean>(){
9              public Boolean generateKey( Dockable dockable ){
10                 return dockable.<getSomeProperty()>;
11             }
12         });
13         setRemoveEmptyGroups( false );
14
15         setSelected( Boolean.FALSE, false );
16         setSelected( Boolean.TRUE, true );
17
18         setText( Boolean.FALSE, "Unselected" );
19         setText( Boolean.TRUE, "Selected" );
20     }
21
22     @Override
23     public boolean trigger( Dockable dockable ) {
24         setSelected( dockable, !isSelected( dockable ) );
25         return true;
26     }
27
28     @Override
29     public void setSelected( Dockable dockable, boolean selected ){
30         dockable.<setSomeProperty( selected )>;
31         setGroup( selected, dockable );
32     }
33 }
```

The constructor (lines `7-20`) sets up the action. First the `GroupKeyGenerator` is set in lines `9-12`. The key is a `Boolean` which represents "some property" of a `Dockable`. The meaning of the property is not important. Through the keys `Dockables` get grouped. When `Dockables` get added and removed a group may become empty. Line `13` ensures that the action does not delete the properties of empty groups.

A `Boolean` only has two states, both states will be used as key. So there is a "true" and a "false" group. The selected-state of the action should match the key of the group. In other words: if "some property" is `true` then the action is selected, if "some property" is `false` then it is not. Lines `15, 16` are responsible for this setting. The same behavior is enforced for the text of the action in lines `18, 19`.

The standard behavior of a `SelectableDockAction` is to change its selected state as soon as the user triggers the action. If the action is used for many `Dockables` than this behavior would look rather odd. All the actions would change their state and most of them would do so wrongly. By overriding the method `trigger` this problem can be prevented (lines `23-26`). Instead of changing the selected state of the action, the group of the `Dockable` is changed by invoking `setSelected` in line `24`. Since the two groups have different selection states the user will think that the action changed the state.

By the way: the method `setSelected` in lines `29-32` needs to be overriden since the default behavior is to change the state of the action, not to change the group of a `Dockable`.

Be careful when using group actions: they are complex to handle. In many cases a simple action can replace a group action.

Group actions were introduced for `DockStations`. `DockStations` need to apply the same actions to many `Dockables`. Instead of setting up new actions all the time it was easier to have one action that holds many properties at the same time.

There are only three group actions implemented:

- `GroupedButtonDockAction`
- `GroupedSelectableDockAction.Check`
- `GroupedSelectableDockAction.Radio`

4.3 Custom actions

Clients are free to implement new actions.

4.3.1 Reuse existing view

Whenever possible an existing view should be reused. There are 6 kind of views defined in the framework. Each kind of view is represented through an instance of `ActionType`, each of them is stored as constant in `ActionType` itself. `ActionType` has one generic parameter. The view can force an action to implement some interface through that parameter. For example, the kind `ActionType.BUTTON` forces an action to implement `ButtonDockAction`. Actions can use an `ActionType` as key for a factory that is stored in the `ActionViewConverter`.

In a real world example that will look like this:

```java
public class ExampleButtonAction implements ButtonDockAction{

    public <V> V createView( ViewTarget<V> target,
            ActionViewConverter converter, Dockable dockable ){

        return converter.createView( ActionType.BUTTON, this,
            target, dockable );
    }

    public void action( Dockable dockable ){
        [...]
    }

    public Icon getIcon( Dockable dockable ){
        return [...];
    }

    [...]
}
```

Really important are the lines 3-8: these lines are all that is necessary to create different button-views for different environments (menu, title). The `ActionViewConverter` does all the work, it just has to be called with the correct parameters.

The interface `ButtonDockAction` declares other methods like `getIcon` (lines 14-16) which will not be a challenge to implement.

4.3.2 Custom view

Writing a custom action with custom view is possible, but will require a lot of work. Some good news: it is only necessary to implement the interface `DockAction` and the raw interface `DockAction` has only very few methods. The greatest challenge will be to write the method `createView`. This method can be called any time and receives a `ViewTarget`, a `ActionViewConverter` and the `Dockable` for which the view will be used. It has to return either `null` or the type of object that is specified as the generic parameter of `ViewTarget`. The framework will always use the same three instances of `ViewTarget`, all of them are stored as constants in `ViewTarget` itself. So in theory a `createView` could check which of the three `ViewTarget`s it received and create one of three different views. In practice it is much better to use the `ActionViewConverter` for this task.

You might remember that the `ActionViewConverter` can instanciate new views if an `ActionType` is given to its `createView` method. So the first step in creating a custom action should be to write the new class (or interface) and declare the new type. The second step would be to call `createView`. The third step to implement the remaing methods. The result of these steps could look like this:

```java
import bibliothek.gui.Dockable;
import bibliothek.gui.dock.action.ActionType;
import bibliothek.gui.dock.action.DockAction;
import bibliothek.gui.dock.action.view.ActionViewConverter;
import bibliothek.gui.dock.action.view.ViewTarget;

public class CustomAction implements DockAction{
    public static final ActionType<CustomAction> CUSTOM =
        new ActionType<CustomAction>( "custom" );

    public <V> V createView( ViewTarget<V> target,
            ActionViewConverter converter, Dockable dockable ){
        return converter.createView( CUSTOM, this,
                target, dockable );
    }

    @Override
    public void bind( Dockable dockable ){
        // ignore
    }

    @Override
    public void unbind( Dockable dockable ){
        // ignore
    }

    public boolean trigger( Dockable dockable ){
        return false;
    }
}
```

Now the `ActionViewConverter` needs to be instructed of what to do with the `ActionType CUSTOM`. This should be done on startup, before the first

CustomAction is even created. The ActionViewConverter is accessible through the DockController. A client can call putDefault to set the default view factory for some type and target:

```java
1 DockController controller = ...;
2 ActionViewConverter converter = controller.getActionViewConverter();
3
4 ViewGenerator<CustomAction, BasicTitleViewItem<JComponent>> generator =
5     new CustomButtonGenerator();
6
7 converter.putDefault( CustomAction.CUSTOM, ViewTarget.TITLE,
8     generator );
```

In this code the converter is accessed in line 2. Some new factory is created in lines 4, 5 and this new factory is registered at the converter in lines 7, 8. The CustomButtonGenerator is just a class that implements ViewGenerator:

```java
public class CustomButtonGenerator implements
        ViewGenerator<CustomAction, BasicTitleViewItem<JComponent>>{
    public BasicTitleViewItem<JComponent> create(
            ActionViewConverter converter, CustomAction action,
            Dockable dockable ){

        return [...]
    }
}
```

Set a ViewGenerator for ViewTarget.TITLE, ViewTarget.MENU and for ViewTarget.DROP_DOWN. Even if these generators do not create views but just return null, not installing them would lead to an error.

5 Titles

A DockTitle is a Component that may show an icon, a text, some DockActions or other information about a Dockable. Users often grap a DockTitle when they want to initialize a drag & drop operation.

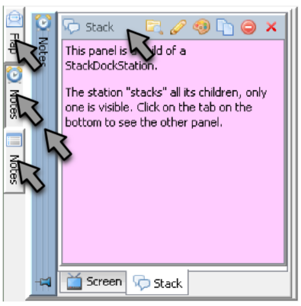

Figure 2: Some DockTitles.

5.1 Lifecycle

This chapter will explain the mechanism of creating and managing DockTitles.

A module that wants to show a `DockTitle` first has to create a unique identifier. The unique identifier is represented by an instance of `DockTitleVersion`. A `DockTitleVersion` can be created by the `DockTitleManager`. When a module asks for a `DockTitleVersion` is has to give a `DockTitleFactory` to the manager, this factory becomes the default factory and will be used to create titles (unless someone replaces the default factory).

If the module receives a `Dockable` it can call `getDockTitle`. This method requires the unique identifier `DockTitleVersion` which was created earlier. Most `Dockables` would then access the default factory to create the title, but some `Dockables` might create their customized titles.

Assuming `getDockTitle` does not return `null`, the module calls the method `bind( DockTitle )` of `Dockable`, this tells the `Dockable` that is has a new title. If the module no longer needs the title it calls `unbind( DockTitle )`.

Do not call the method `bind` or `unbind` of `DockTitle`, these methods are called automatically by the `DockController`.

`Dockables` provide some information about their titles:

- The method `listBoundTitles` returns an list of all `DockTitles` which are currently in use for the `Dockable`.
- A `DockableListener` has several methods that will be invoked if titles get added, removed, updated or exchanged.

5.2 Custom titles

5.2.1 Write the title

It is possible to replace all the titles in the framework. The interface `DockTitle` is rather open and the title is responsible to collect the information it wants to show.

Most titles will have a constructor that has a `Dockable` as argument. They will add a `DockableListener` to their `Dockable` once `bind` is called and remove the listener once `unbind` is called.

Modules that show a title can communicate with the title through the method `changed`. This method takes a `DockTitleEvent` with further information.

A module does not need to know what title it shows. It just delivers the `DockTitleEvent` to the title. The module can use a subclass of `DockTitleEvent` to transfer more information than `DockTitleEvent` alone could carry. This design allows to use any implementation of `DockTitle` at any place while some titles still can use additional information from their environment. An example is the `EclipseDockTitleEvent` which is used by tabs. This event also tells the titles at which location they are and whether their tab is focused or not.

There are some classes that can help implementing a custom title:

- `AbstractDockTitle` provides standard implementations for most of the features a title requires. Subclasses only need to override the method `paintBackground` to have their custom painting code used.
- `BasicDockTitle` paints some gradients as background. Clients can change the color of these gradients. This title is also a good reference of how things can be done.
- `ButtonPanel` is a `Component` able to display a set of `DockActions`. `ButtonPanel` is able to show a popup-menu if there is not enough space for all actions.

In order to use the popup menu of `ButtonPanel` some special code has to be written. First: the argument `menu` of the constructor of `ButtonPanel` has to be set to `true`. Second: the method `getPreferredSize` of `ButtonPanel` cannot be used, any standard `LayoutManager` will fail. Instead the method `doLayout` of the `Container` which shows the panel can be overriden. In this `doLayout` method the container should call `getPreferredSizes` to obtain a list of possible sizes of the panel. The n'th dimension in this array tells how big the `ButtonPanel` would be if it would show n actions. The container should choose the biggest possible n and call `setVisibleActions`.

5.2.2 Apply the title

There are three ways to introduce a custom title into the framework.

To override or implement `getDockTitle` of `Dockable` is the simplest way. The method just creates a new instance of the custom title when called.

The `DockTheme` can be used as well. Either override the method `getTitleFactory` or call `setTitleFactory` when using a `BasicTheme`. With a few exceptions all the modules use the factory of the theme, hence replacing this factory will have a big effect.

Or use the `DockTitleManager` to make some better tuned settings. The `DockTitleManager` can be accessed by calling `getDockTitleManager` of `DockController`. Search the unique string identifier of the module that uses a title and call `getVersion` to access the associated `DockTitleVersion`. Then with the help of `setFactory` a new factory can be introduced. In code this could look like this:

```
DockController controller = ...

DockTitleManager manager = controller.getDockTitleManager();
DockTitleVersion version =
  manager.getVersion( SplitDockStation.TITLE_ID, null );
version.setFactory( new CustomDockTitleFactory(), Priority.CLIENT );
```

6 Themes

A `DockTheme` relates to `DockingFrames` like a `LookAndFeel` to `Java Swing`. At any given time a `DockController` is associated with exactly one theme. The theme defines various graphical elements like icons, painting code and also some behavior code. The current `DockTheme` can be changed through the method `setTheme`:

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

6.1 Existing Themes

Several `DockTheme`s are already included in the framework. A list of theme-factories can be accessed through the method `getThemes` of `DockUI`. This sub-chapter will list up the existing themes and mention some of their specialities.

Keep in mind that `DockTheme`s do not have to follow a specific path for setting up their views. All the current themes are derived from `BasicTheme` and thus share a lot of concepts. Future or custom themes however might be implemented in different ways.

6.1.1 NoStackTheme

This theme is a wrapper around other themes. It prevents `StackDockStation`s from having a `DockTitle` and makes sure that the user cannot drag or create a `StackDockStation` into another `StackDockStation`. The code for creating a `NoStackTheme` looks like this:

```
DockTheme original = ...
DockTheme theme = new NoStackTheme( original );
```

6.1.2 BasicTheme

The `BasicTheme` is a simple but working theme. All the other themes of the framework build upon `BasicTheme`. This theme shows content whenever possible. It tries to use all features and thus is quite good for debugging, to check whether all features are supported.

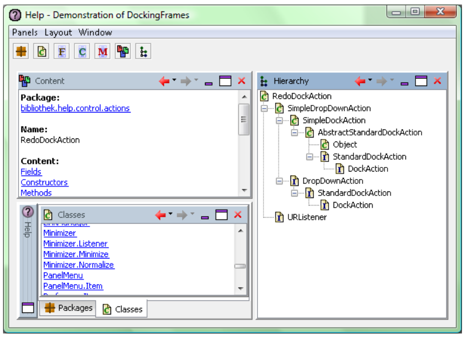

Figure 3: BasicTheme

6.1.3 SmoothTheme

`SmoothTheme` is basically the same as `BasicTheme`. The only difference is a replaced default-`DockTitleFactory`. As a result new `DockTitle`s are used by most elements, these new titles smoothly change their color when the "active" state of their `Dockable`s changes.

6.1.4 FlatTheme

`FlatTheme` is a variation of `BasicTheme` that tries to minimze the number of borders. Among other things it uses new `DockTitle`s and new views for `DockAction`s. It is the ideal theme for developers that want to learn how to customize an existing theme.

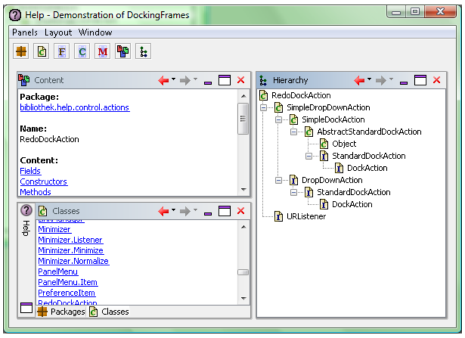

Figure 4: FlatTheme

6.1.5 BubbleTheme

A more experimental theme. `BubbleTheme` often uses animations and other graphical gimmicks. It has a few performance issues, but it is a good theme to demonstrate the potential of the theme-mechanisms.

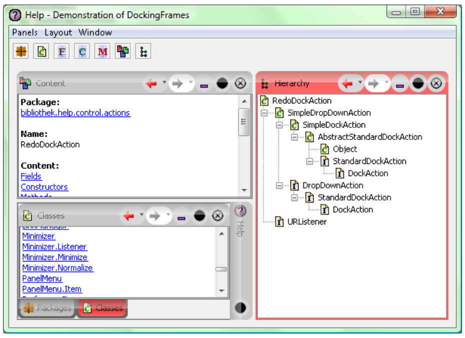

Figure 5: BubbleTheme

6.1.6 EclipseTheme

`EclipseTheme` tries to mimmic the behavior and look of the well known IDE Eclipse. All the `Dockables` are shown on tabbed-components and often `DockTitles` are replaced by the tabs. The theme does not use the default theme-mechanisms as often as other themes and it might be a bit tricky to customize the theme. On the other hand it certainly looks good.

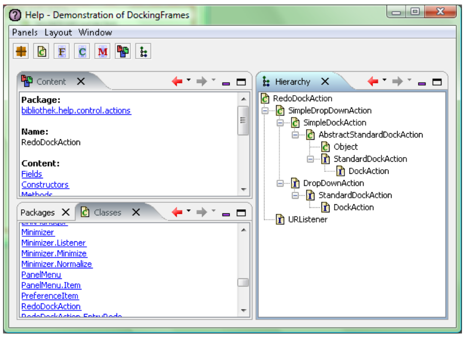

Figure 6: EclipseTheme

`EclipseTheme` offers some keys the map of properties that is stored in `DockProperties`. The keys are:

PAINT_ICONS_WHEN_DESELECTED A `Boolean` that tells whether icons on tabs should be painted if the tab is not selected. In every tabbed-component one tab has to be selected and its associated `Dockable` is the only visible element on the component.

THEME_CONNECTOR An `EclipseThemeConnector`. The connected tells whether a `DockAction` belongs onto a tab, or in a separate list of "unimportant" actions. The connector also tells what kind of title to use for a `Dockable`.

TAB_PAINTER A `TabPainter`. This class is a factory that creates the tab-components and sets up other settings that are related with tabs.

The `DefaultEclipseThemeConnector` puts every `DockAction` which is annotated with `EclipseTabDockAction` onto tabs.

The settings for titles and borders that are given by an `EclipseThemeConnector` are not respected if the element is on a `StackDockStations`. A `StackDockStation` always uses some tabbed-component.

6.2 Customize DockThemes

More than 50% of the frameworks source code is only used for painting stuff. No `DockTheme` uses particular complex code, just the mass can lead to some loss of direction. This sub-chapter will give only an overview of the basic classes, interfaces and concepts.

Many of the mechanisms used by `DockThemes` can be used by clients as well.

6.2.1 UI-Properties

UI-properties is a concept to distribute properties to components. A property could be a `Color` and a component some `DockTitle` which uses that color to paints it background. The basic idea is to use a map. The keys are `Strings`, the values are the properties. A `DockTheme` or a client can modify or put new key-value pairs into the map and components can read those values which are interesting for them.

Unfortunatelly a simple map is not enough. There needs to be a way to specify values that are used only by a subset of components. Or to remain with the map: the components must become part of the key as well.

The UI-properties provide the necessary features. The mechanism includes these classes, interfaces and generic parameters:

- `UIProperties`: the base map.
- `V`: the generic values that have to be distributed, e.g. the class `Color`.
- `UIValue`: A wrapper around `V`. Each component creates one `UIValue` for each query it will ask the `UIProperties`. In example a `DockTitle` would have to create and store exactly one `UIValue` to represent its background color. `UIValues` also act as observer and the `UIProperties` notify an `UIValue` if its wrapped `V` gets changed.
- `UIBridge`: An `UIBridge` is set between a set of `UIValues` and the `UIProperties`. `V` properties will not be handed directly from `UIProperties` to `UIValue` if there is a bridge between. The bridge can

modify the `V` property in any way it likes, since the `UIBridge` knows the destination of a `V` it can also use information derived from the `UIValue`.

The implementation gets more complex:

- For each key several `V` properties can be put into the base map. Each value gets assigned another priority ("default", "theme" or "client") and only the one with the. highest priority is used.
- Each `UIValue` is associated with a `Path`. The `Path` tells what an `UIValue` will do with the `V` property. The `Path` also tells what kind of type the `UIValue` has.
- `UIBridge`s are also associated with a `Path`. An `UIBridge` is responsible to handle all those `UIValues` that are associated either with the same `Path` or a `Path` that has the bridges `Path` as prefix.

This scheme allows a flexible handling of resources. On one hand the number of keys is limited and one method call is enough to change a lot things in the user interface (e.g. all background colors of titles). On the other hand clients can implement sophisticated strategies to change some properties without the need to know in detail how the property will be used.
Originally this mechanism was invented to handle `Colors`. Then it became evident that the same mechanism could be used for other resources as well. The current implementation requires to implement several classes for each type of resource. While this might be annoying for the first use it ensures type safety. In a system where cause (writing in the map) and effect (reading from the map) can be separated by dozens of classes and an unknown amount of time one does not want to care about types as well.

6.2.2 Colors

In order to understand this chapter 6.2.1 should be read first.

All the colors used in the framework are handled by the `ColorManager`. The `ColorManager` is an `UIProperties` and can be accessed through the `DockController`. It's use could look like this:

```
DockController controller = ...
ColorManager colors = controller.getColors();
colors.put( Priority.CLIENT, "title.active.left", Color.GREEN );
```

In this snippet the value for the key "title.active.left" is changed to green. The priority `CLIENT` is highest possible priority. It is never overridden by the framework.

Or a more sophisticated use could involve a `ColorBridge`:

```
DockController controller = ...
ColorManager colors = controller.getColors();
colors.publish( Priority.CLIENT, TitleColor.KIND_TITLE_COLOR, new
    ColorBridge(){
    public void add( String id, DockColor uiValue ){
        // ignore
    }
```

```
7       public void remove( String id, DockColor uiValue ){
8           // ignore
9       }
10      public void set( String id, Color value, DockColor uiValue ){
11          TitleColor title = (TitleColor)uiValue;
12          if( title.getTitle().getDockable() == <somevalue> )
13              title.set( Color.GREEN );
14          else
15              title.set( value );
16      }
17  });
```

Here a `ColorBridge` for the `Path` `KIND_TITLE_COLOR` is installed in line `3`. This path is only used by `UIValues` that implement `TitleColor`. Hence the unchecked cast from `DockColor` to `TitleColor` in line `11` is safe. The methods `add` (line `4-6`) and `remove` (line `7-9`) are called by `UIProperties` when a `UIValue` gets added or removed to it. These methods can be ignored as long as the bridge does not change the color on its own. Otherwise the `DockColors` could be stored in some list and their method `set` could be called whenever the color needs to be exchanged.

This bridge searches for a specific `Dockable` called "somevalue" (line `12`). The bridge returns `GREEN` for all colors used by any title of this `Dockable`. There is no distinction between the colors for background, foreground or other usages.

There is no global list of keys and every `DockTheme` uses different keys. All the modules that need colors are annotated with `ColorCodes` and expose their own list of keys to the API-documentation. Also the method `updateColors` of `BasicTheme` or subclasses can help: in this method all the colors that will ever be used by the theme are written into the `ColorManager`.

All the standard themes use a `ColorScheme` as their initial set of colors. All the standard themes provide a key for the `DockProperties` to change that initial scheme. For example the key provided by `BasicTheme` is stored as constant `BASIC_COLOR_SCHEME`. There are several subclasses of `ColorScheme` for the different themes.

By the way: some themes use colors that are read from the current `LookAndFeel`. Clients can call the method `registerColors` of `DockUI`. This method takes a `LookAndFeelColors` which is responsible in reading the colors from the `LookAndFeel`.

6.2.3 Fonts

Fonts use the same mechanism as Colors. A `FontManager` can be accessed through the methods `getFonts` of `DockController`. Unlike colors a set of standard keys are defined as constants in `DockFont`.

The `FontManager` does not distribute `Font`-objects but `FontModifiers`. A `FontModifier` has one method that receives the original `Font` and can return any `Font` it likes. In example a `FontModifier` could inverse the bold-property of a `Font`. There are two `FontModifiers` ready to use:

- `ConstantFontModifier` does not modify anything but always return the same `Font`
- `GenericFontModifier` can modify the italic-, bold- and size-property of a font.

Clients that want to use a `FontModifier` might be interested in the classes `DLabel` and `DPanel` which already modify their font. Also the class `FontUpdater` can be used to create new `JComponents` with the capability to modify their font.

6.2.4 Icons

`Icons` can be modified through the `IconManager`. The `IconManager` is just a map with the capability to inform observers if some of its value changed. The `IconManager` can be accessed through the method `getIcons` of `DockController`.

There is no global list of keys in the source code. However the file "icons.ini" contains a list of keys and paths of all the default icons.

6.2.5 Actions

The views for `DockActions` are changed through the `ActionViewConverter`. Please read chapter 4 for more information.

6.2.6 Titles

`DockTitles` are managed by the `DockTitleManager`. Please read chapter 5 for more information.

6.3 Custom Theme

With the exception of the classes that are directly related to a `DockTheme` no code in the framework depends on a special undocummented behavior of a theme. Clients can reimplement the interface `DockTheme` without fear to break things.

A better approach then full reimplementation might be to extend the class `BasicTheme`. This class provides some default values which can easily be changed by the appropriate `setXZY` method.

`DockTheme` has a method `install`, this method can be used to exchange some values that are not stored in the `DockTheme` itself. For example to exchange icons in the `IconManager`.

A theme dives deep into the framework. Implementing a new theme requires a lot of time and a good understanding of the framework. This document might help to understand the basics, but some stuff can only be found out by looking directly at the source code.

7 Drag and Drop

To drag a `Dockable` to a new location and drop it there is the most important feature of any docking framework. Surprisingly the implementation of this part is very small.

7.1 Relocator

The sourcecode that detects drag gestures, searches for the target station and makes sure that the user has some visual feedback is located in the `DefaultDockRelocator`. `DefaultDockRelocator` itself extends from `DockRelocator` which just allows to register some listeners and set some useful properties. Clients seldomly need to implement a new `DockRelocator`. If they do then they, then they have to implement a new `DockControllerFactory`. The code will look like this:

```java
public class MyDockControllerFactory extends
    DefaultDockControllerFactory{
  @Override
  public DockRelocator createRelocator( DockController controller ) {
    return new MyDockReloactor();
  }
}
```

This factory has then to be given to the constructor of a `DockController`. For the remainder of this chapter it is assumed, that the default relocator is in use.

The `DockRelocator` that is in use can be accessed through the method `getRelocator` of `DockController`.

7.2 Sources

The relocator needs to know where and when the user presses and moves the mouse. There is more than one solution for this problem.

7.2.1 DockElementRepresentative

A `DockElementRepresentative` is a `Component` which represents a `Dockable`. Anyone can add `MouseInputListeners` to a representative and hence be informed about anything the mouse does on top of such a `Component`.

`DockTitle` and `Dockable` are two implementations of `DockElementRepresentative`. Their registration is handled automatically. If clients implement a new representative then they should call the methods `addRepresentative` and `removeRepresentative` of `DockController` to install or uninstall the representative.

> `DockElementRepresentative` was added late to the framework. It carries some legacy code: the method `isUsedAsTitle`. This method introduces a distinction between those representations for which all features are activated (e.g. popup menus) and those for which only a selected subset is available. Normally clients implement representatives that are used as title and can return `true` here.

The behavior for representations of `Dockables` that are not registered is unspecified. Clients should not add a `DockElementRepresentative` if its `Dockable` is unknown to the `DockController`.

7.2.2 Remote control

Sometimes it is not possible to implement a `DockElementRepresentative`. Remote control of a relocator is an alternative for these cases. Remote control is realized by the classes `RemoteRelocator` and `DirectRemoteRelocator`.

A `RemoteRelocator` can be optained by calling `createRemote` of `DockRelocator`. `RemoteRelocator` should be used in combination with a `MouseListener` and a `MouseMotionListener`:

- `MouseListener.mousePressed` →`RemoteRelocator.init`
- `MouseMotionListener.mouseDragged` →`RemoteRelocator.drag`
- `MouseListener.mouseReleased` →`RemoteRelocator.drop`

The methods `init`, `drag` and `drop` return a `Reaction`. The reaction tells the caller what to do next:

- `CONTINUE`: the operation continues, the event was ignored.
- `CONTINUE_CONSUMED`: the operation continues, the event was consumed. The caller should invoke `MouseEvent.consume`.
- `BREAK`: the operation was canceled, the event was ignored.
- `BREAK_CONSUMED`: the operation was canceled, the event was consumed. The caller should invoke `MouseEvent.consume`.

A `DirectRemoteRelocator` can be optained by calling `createDirectRemote` of `DockRelocator`. A `DirectRemoteRelocator` is basically the same as a `RemoteRelocator` but always assumes that the user pressed the correct button on the mouse. Its methods do not return a `Reaction` because it would always be the same.

Clients can use several remote controls at the same time, they will cancel each other out if necessary. A `RemoteRelocator` can be used several times.

7.3 Destinations

A relocator needs to find the one `DockStation` on which the `Dockable` is dropped.

7.3.1 Search

The `DefaultDockRelocator` searches the destination anew whenever the mouse is moved. The search includes these steps:

1. An ordered list of all potential destinations is built. A `DockStation` is a potential destination if it is visible (`isStationVisible` of `DockStation`), not the dragged `Dockable` nor one of its children, and its boundaries contain the location of the mouse (`getStationBounds` of `DockStation`). The order depends on parent-child relations between the stations, between the `Windows` on which the stations are, and on custom conditions that every station can offer (`canCompare` and `compare` of `DockStation`).

2. Then the method `prepareMove` or `prepareDrop` of `DockStation` is called. These methods check whether the station really is a good destination. They return `true` if so, `false` if not. The first station that returns `true` is the destination.

3. The method `draw` of the new destination is called, the method `forget` on the old destination. The new destination will paint some markings to give a visual feedback to the user, the old destination will delete all the information about any drag and drop operation.

> There is more information about the exact semantics in the API-documentation for `DockStation`.

7.3.2 Drop

The moment a user releases the mouse and drops a `Dockable` the method `move` or `drop` of `DockStation` is called. These methods can either put the `Dockable` somewhere onto the station or merge the `Dockable` with an existing child of the station (sometimes referred as “put” and “merge” action). The results of the first reaction depend on the kind of station. The results of the second reaction are independent of the kind of station.

Merging normally results in creating a new `StackDockStation`. The existing child and the dropped `Dockable` are put onto that new station. Then the `StackDockStation` is put at the place where the existing child was. Creation of “merged `Dockables`” is handled by a `Combiner`, per default by the `BasicCombiner`. Many `DockStations` 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`.

> Exchanging a `Combiner` does not affect any existing `Dockable` or `DockStation`, it will only affect the creation of new elements.

7.4 Influences

There are a number of factors that can influence the search for a new destination. Some of them are customizable.

7.4.1 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.

> The keys that have to be pressed to activate `SCREEN_ONLY` or `NO_COMBINATION` are the properties `SCREEN_MASK` and `NO_COMBINATION_MASK`. The can be changed by accessing the `DockProperties`.

7.4.2 Restrictions

The set of possible destinations for a `Dockable` can be restricted. There are several reasons why a client or the framework itself would do that:

- Some `Dockable` must always be visible.
- Some `DockStation`s represent a special area that can only be used by a subset of `Dockable`s.
- Some `Dockable`s can only be presented on a certain kind of `DockStation`.

This restrictions are implemented through acceptance tests. An acceptance test either checks one "put" or one "merge" action. Tests can be stored at various locations:

- Every `Dockable` has two methods called `accept`.
- Each `DockStation` has a method `accept`. This method tells whether some `Dockable` can become a child of the `DockStation`. This method checks "put" and "merge" actions at the same time.

- 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.

> Acceptance tests are very powerful. They have to be implemented carefully or the drag and drop mechanism might become crippled.

************************ was writing here **********************

8 Preferences

The preference system was introduced with version 1.0.6 of the core library.

If a setting is meaningful for the ordinary user, and the user would like to be able to change the setting, then this setting should be made accessible through a preference. For example the shortcut to maximize a Dockable (ctrl+m) is a good candidate for a preference.

So what is a preference? A preference is a representation of some kind of property in a unified way. It is a mediator between the system in which the property is stored, and the graphical user interface and storage mechanisms written for preferences.

Each preference consists of some properties:

Value The thing which the user would like to change.

ValueInfo Information about the value, for example the maximum value for an Integer-value. The exact meaning of this property depends on the TypePath.

TypePath The type tells how to work with the value, how to present it to the user or how to write it as xml. The type is represented by a Path-object. It is a Path and not a Class object because many preference-types may use the same objects as value. For example an unbounded Integer versus an Integer which must be in the range 1 to 10.

Path A unique path to this preference. Used as an identifier if preferences have to be stored in some kind of map.

8.1 Organization

The basic module of the preference system is the PreferenceModel.

Most methods of the PreferenceModel are simple to understand and do not need a discussion. Those which are part of a greater scheme however will receive some attention in this chapter.

In the preference system a PreferenceModel is just a layer above some kind of storage mechanism for the real properties. It is most often used as a mediator and a buffer between that storage mechanism and the algorithms that want to use the preferences (for example a user interface). The methods `read` and `write` are used to access the covered storage mechanism. The method `read` will read values from the storage mechanism into the model. The method `write` will write the values back into that storage mechanism.

8.2 Models

There are some implementations of `PreferenceModel` already in the core library.

8.2.1 DefaultPreferenceModel

This model uses a list of `Preference`-objects to represent the preferences. All the properties needed for a preference are stored in such a `Preference`-object. The API-documentation reveals that there are many `Preference`s representing different aspects of the core library. For example there is a `Preference` which represents the keystroke for maximizing a `Dockable`.

There are also subclasses of `DefaultPreferenceModel`. These subclasses are collections of preferences which belong together, for example the `EclipseThemePreferenceModel` which contains preferences that are related to the `EclipseTheme`.

8.2.2 MergedPreferenceModel

This model is a list of other models. It just takes the preferences from these other models and presents them as its own preferences. It offers a quick and simple way to create a combination of two ore more models.

8.2.3 PreferenceTreeModel

This model is a `PreferenceModel` and a `javax.swing.TreeModel`. If seen as `PreferenceModel`, then it behaves like a `MergedPreferenceModel`. If seen as `TreeModel`, then it contains `PrefereceTreeModel.Node`-objects. A node can either be just a name, or another `PreferenceModel`. This model is intended to be used in a `JTree` where the user can select one aspect of the whole set of preferences to show.

The subclass `DockingFramesPreferenceModel` is the set of preferences which includes all the aspects of the core-library.

8.3 Lifecycle

This section describes the best way how to use a `PreferenceModel`. Not everything used in this section is explained yet, so you might want to read this section a second time when you finished this whole chapter.

The correct lifecycle of a `PreferenceModel` includes normally these steps:

1. Create the model. Set up all the preferences that are used by the model.
2. Call `load` on a `StoragePreference`.
3. Call `write` on the model to synchronize the model with the underlying system.

4. (work with the underlying system)
5. To work with the model: call first `read`, then make the changes in the model, then call `write`.
6. (work with the underlying system)
7. Call `read` on the model to synchronize the model with the underlying system.
8. Store the model using `store` of a `PreferenceStorage`.

If the `PreferenceStorage` used in step 2 is empty because its `read` or `readXML` method failed, then calling `read` of `PreferenceModel` would at least load some default settings.

Steps `4, 5, 6` can be cycled as many times as needed.

An additional step `0` and `9` would be to read and write the `PreferenceStorage` when starting up or shuting down the application.

8.4 User Interface

If a `PreferenceModel` should be displayed, the `PreferenceTable` can be used. This table shows a label and an editor for each preference. For `PreferenceTreeModels` a `PreferenceTreePanel` should be used, it shows a `PreferenceTable` and a `JTree` for the nodes of the `PreferenceTreeModel`.

Clients can also use a `PreferenceDialog` or a `PreferenceTreeDialog` to show a dialog with the well known buttons "ok" and "cancel".

8.4.1 Editors

Since there are different types of preferences, different editors are needed. The kind of editor for one preference is determined by the type-path (`getTypePath` in a model). Clients can add new editors to a `PreferenceTable` through the method `setEditorFactory`.

An editor is always of type `PreferenceEditor`. Each editor gets a `PreferenceEditorCallback` with which it can interact with the table. Whenever the user changes the editors value, the editor should call the method `set` of `PreferenceEditorCallback` to make sure the new value gets stored.

8.4.2 Operators

There are some operations which should be available for almost any preference. For example *set a default value* or *delete the current value*. The preference system introduces the `PreferenceOperation` to handle this kind of actions.

A `PreferenceOperation` is nothing more than a label and an icon. The logic for an operation is either in an editor or in a model.

Editor: Editors with operations must call the method `setOperation` of `PreferenceEditorCallback` for each operation they offer. By calling `setOperation` more than once, the editor can change the enabled state of the operation. If the user triggers an operation of the editor, the method `doOperation` of `PreferenceEditor` is called. It is then the editors responsibility to handle the operation.

Preference: Preferences can have operations as well. The method `getOperations` of `PreferenceModel` will be called once to get all the available operations for one preference. The method `isEnabled` will be invoked to find out whether an operation is enabled or not. Models can change the enabled state by calling `preferenceChanged` of `PreferenceModelListener`. If the user triggers an operation, `doOperation` of `PreferenceModel` will be invoked.

If an editor and a preference share the same operations, then per definition the operations belong to the editor. All settings from the model will just be ignored.

8.5 Storage

The `PreferenceStorage` can be used to store `PreferenceModel`s in memory or persistent either as byte-stream or as XML.

The normal way to write a model from memory to the disk looks like this:

```
1  // the stream we want to write into
2  DataOutputStream out = ...
3
4  // the model we want to store
5  PreferenceModel model = ...
6
7  // And now store the model
8  PreferenceStorage storage = new PreferenceStorage();
9  storage.store( model );
10 storage.write( out );
```

Note that there are two phases in writing `model`. First the model gets `stored` (line `9`) into `storage`. It is possible to store more than just one model in a `PreferenceStorage`. Second `storage` gets written onto the disk in line `10`.

The standard way to read a model are to apply the same steps in reverse:

```
1  // the source of any new data
2  DataInputStream in = ...
3
4  // the model we want to load
5  PreferenceModel model = ...
6
7  // And now load the model
8  PreferenceStorage storage = new PreferenceStorage();
9  storage.read( in );
10 storage.load( model, false );
```

Like writing this operation has two phases. In line `9` `storage` gets filled with information, in line `10` the information gets transfered to `model`. The argument `false` is a hint what to do with missing preferences. In this case missing preferences are just ignored. A value of `true` would force them to become `null`.

There are some preferences which do not need to be stored by the `PreferenceStorage` because they are already stored by the underlying systen. These preferences are called *natural*, while the others are called *artificial*. The method `isNatural` of `PreferenceModel` can be used to distinguish them.

9 Properties