DockingFrames 1.1.3 - Core

Benjamin Sigg

May 5, 2019

Contents

1 Introduction

DockingFrames is an open source Java Swing framework published under the LGPL (lesser GNU public license). This means you are allowed to use DockingFrames in any way you like. However if you modify the framework you are required to distribute the modified source code together with your new library.

This document introduces you to the basic concepts of Core. Naturally it cannot cover all the details, you should also have a look into the API documentation http://dock.javaforge.com/doc.html, the tutorial project and the forum on http://forum.byte-welt.net/.

1.1 Core vs Common

DockingFrames consists of two projects, Core and Common (the libraries docking-frames-core.jar and docking-frames-common.jar). The important thing first: clients should use Common whenever possible.

Core provides the functionality, all the code that is required to show things on the screen and to interact with them. The content of Core is very generic, and while the API allows to implement many features, Core itself does not provide them.

Common provides an advanced default setup of Core. Features that are *possible* in Core, are *implemented* in Common. The API of Common is more restricted, but at the same time more easy to use.

If you are uncertain which API to take, consider these tipps:

- If using Common, the API of Core is available as well.
- If you want to implement new functionality that does not yet exist in either project, then you should start with Core.
- Everyone else who just wants to get an application running in a reasonable amount of time, and is not interested in over-the-top customization, will be much happier using Common.
- If you are still not certain: use Common.

Clients should make use of the Common project. In order to do so the libraries docking-frames-core.jar and docking-frames-common.jar have to be included in the classpath.

1.2 Use cases

What does the framework do? DockingFrames manages the layout of your graphical user interface. It allows the user to rearrange your user interface in the way he or she likes it. All you need to do is to group your controles in small panels (called Dockables).

For which application can it be used? In general one can say that bigger applications profit more than small ones. Also power-users will like the flexibility

to set up "their" user interface, the common user however might be overhelmed by all the buttons and options. A typical use-case would be an application which can present so many data to the user that one screen is not enough. With a modifiable user interface the user can easily filter the data and blend out the graphs, panels and buttons he does not need.

1.3 Other frameworks

There are at least 10 other docking frameworks for Java. As with any complex software it is impossible to say which of them is the best one. But there are some features which make sure `DockingFrames` is one of the better ones:

- The licence, you can use the framework without paying a fee nor are you required to open source your entire project.
- It is pure `Swing`, it does not have any dependencies to other libraries. It does not force you to use some special design pattern or set up some cryptic configuration files.
- It does support unsigned applets (does anyone use them anymore?) and webstart.
- Multiple instances can run independent from each other. Sounds trivial, but there are many libraries which cannot handle this case. This design has benefits, for example a preference dialog can easily show a preview. The preview is just another instance and any properties only affects this independent instance.
- Much control for the client. You can change almost anything to your likings. Unfortunatelly this is not always easy as some properties are hidden deep in the framework. On the bright side you are now reading the document which tells you how to modify some of the modules.

1.4 Notation

This document uses the following style-guide:

- "Technical things" like class names and project names are written mono-spaced like this: `java.lang.String`.
- Packages are not written. Almost all classes and interfaces have a unique name and with the help of the API documentation you should be able to find them easily.
- "The client" is the application using `DockingFrames`. "The developer" is you. "The user" is a sentient beeing using "the client", this might even be yourself.
- Additional information is given in boxes like the ones below.

Tips 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 awkward design is not as bad as it first looks.

Examples in the tutorial application are mentioned in these boxes.

1.5 Design principles

In order to understand `Core` it helps to know what the basic design is. These design principles are applied through the entire framework. Most modules follow this principles, altough there are a few exceptions in old code.

- The usage of `static` variables is discouraged. There are no global variables, all components must be built in a way that multiple instances can be run by the same classloader at the same time independently from each other.
- Communication through interfaces and usage of factories. Especialy newer code makes heavy use of factories and interfaces to keep classes independent from each other. This also means that the keyword `instanceof` is to be used rarely.
- Strong typesafty. For the client is should be impossible to smuggle an object of the wrong type into the framework, there should never be a `ClassCastExceptions`.
- Apply properties eagerly. This means that if the client changes some property it is applied before the client continous its work. This makes some parts of the framework more complex, but in the long run it adds a lot of flexibility.

1.6 Numbers

In `Core`, there are about 80'000 lines of code, distributed in over 1700 classes and interfaces. You don't need to know all of them to get your first application to run. Ordered by their semantics, the classes can be collected in groups:

Control group Long living objects which control the behavior of the user interface. For example the object handling drag & drop is created once and remains until the application shuts down.

Swing tree group Objects that are actually seen by the user because they are some kind of `java.awt.Component`. These objects build a tree, the objects from the `control group` can be seen as roots in this tree. Clients, or the framework itself, frequently reorganizes this tree.

Theme group Objects responsible for painting the user interface. Sometimes these classes are big and complex, but they never are important. They can always be replaced with some other painting code.

Support group Various small classes which do not fit into the other groups. These objects often have a short lifetime and can do exactly one task. A factory would be a good example.

Comparing the sizes (number of lines) of these groups the following numbers are seen:

Control group	10%
Swing tree group	30%
Theme group	20%
Support group	40%

2 Basics

The basic idea of `Core` is to have one object that controls 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 `Dockable`s and the dock-areas are `DockStation`s.

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:

```
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 }
```

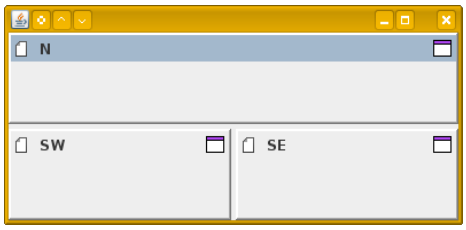


Figure 1: The `HelloWorld` application.

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 archive-file of DockingFrames. The demonstrations are stored in the project called “tutorial“. You can use the ”tutorial.sh“ or ”tutorial.bat“ file to start them.

Another ”hello world“ can be found in the tutorial application under ”Basics/Core/Hello World“.

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:

```
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() ) );
```

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, then clients do not have to worry about these properties. Storage of properties must and will be handled by the framework itself.
Second: Components of the framework cannot get any unfair advantage over custom components. Everything has to be designed in a way that it can work with new and unexpected implementations of `Dockable`.

2.3 DockStation

`Dockable`s can never fly around for themselves, 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. A controller can handle more than just one tree and `Dockable`s can switch from one tree to another.

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.

`Core` offers a collection four different stations. These are listed in the remainder of this section. Additional details can be found in chapter 7.

2.3.1 StackDockStation

This station is organized like a `JTabbedPane`. Only one child is visible, but another can be made visible by clicking some button. The framework will automatically create new `StackDockStation`s when a `Dockable` is dragged over

another. Also `StackDockStation`s with only one child get automatically replaced by this child.

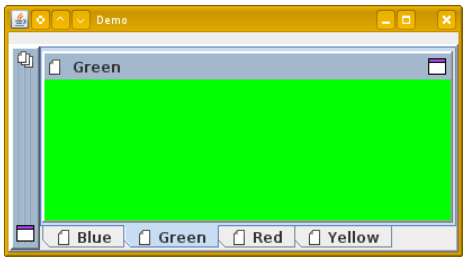
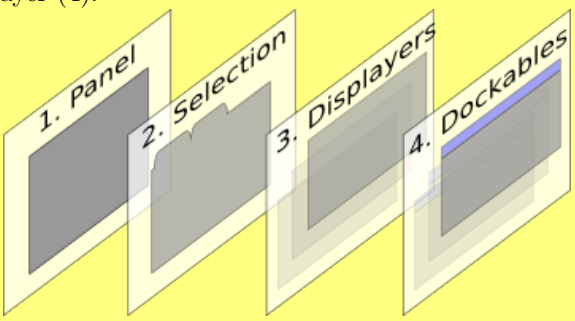


Figure 2: A `StackDockStation` with four children on a frame.

The station consists of four layers, as seen in the image below. There is a background panel (1) which just is some `Container` to put other things onto it. Then there is a selection layer (2), which is represented by an instance of `StackDockComponent`. Above that is a `DockableDisplayer` (3) for each `Dockable`. The displayers paint some decorations that depend on the `Dockables` in the topmost layer (4).



2.3.2 SplitDockStation

All the children of this station are visible. The user controls the children as if the station would consist of many `JSplitPane`s set into each other (hence the name). Internally the station is organized as tree, where a leaf is a `Dockable` and a node the gap between two sets of `Dockable`s. Furthermore this station offers a "fullscreen mode" where one of its children takes up the entire space and all other children are invisible.

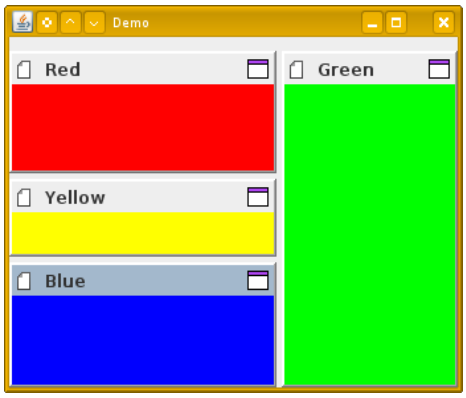
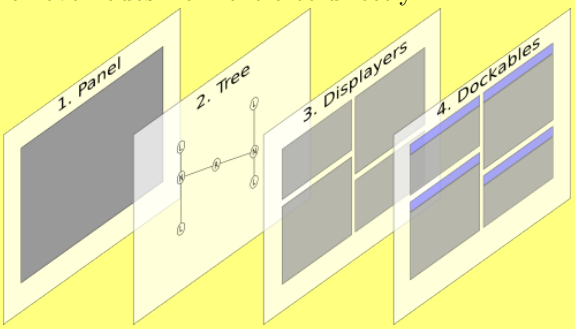


Figure 3: A `SplitDockStation` with four children on a frame.

Like the `StackDockStation`, this station consists of four layers. Layers 1, 3 and 4 are identical to the layers of the `StackDockStation`. A background panel (1), `DockableDisplayers` (3) to paint decorations and the children (4). Layer 2 is the logical tree which tells how to lay out the children. The nodes of this tree consist of `SplitNode`s and the root can be accessed through the method `getRoot`. Clients should never add or remove nodes from the tree directly.



2.3.3 FlapDockStation

This station is a list of buttons. If the user clicks on one of the buttons a window opens showing a child. Only one child can be shown at a time. This station can be used as sidebar to collect "minimized" `Dockables`.

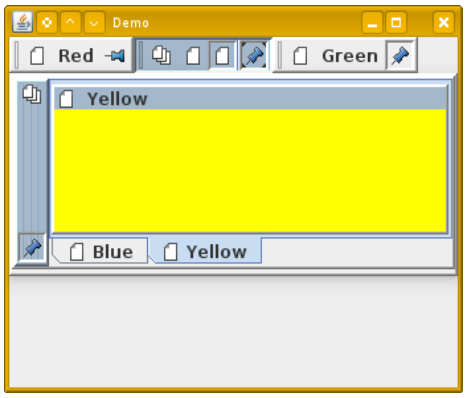
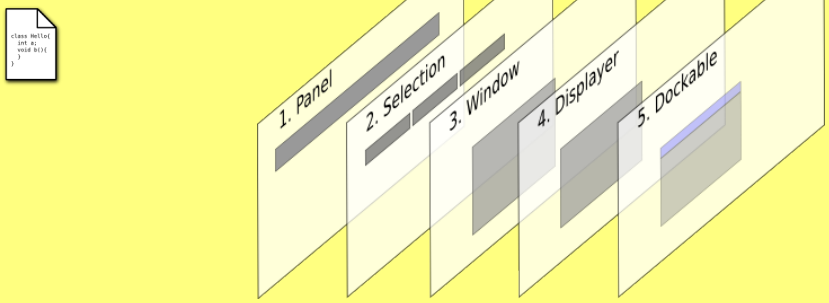


Figure 4: A `FlapDockStation` with three children on a frame. The selected child is a `StackDockStation` containing two more children.

`FlapDockStation` consists of 5 layers. A background panel (1) as `Container` for other components. A set of buttons (2), each button is a `DockTitle`. Then there is the window that shows the current selection (3), an instance of `FlapWindow`. A displayer (4) to paint decorations and the `Dockable` child (5) that is selected.



2.3.4 ScreenDockStation

The `ScreenDockStation` allows its children to float around freely on the screen. Each child is put onto its own window which is independent from any other window. This station also offers a "fullscreen mode" where a window is enlarged to fill the entire space of a screen.

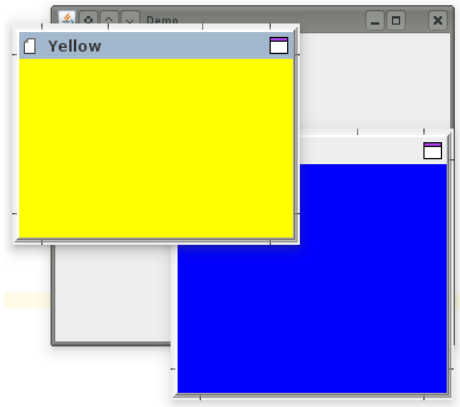
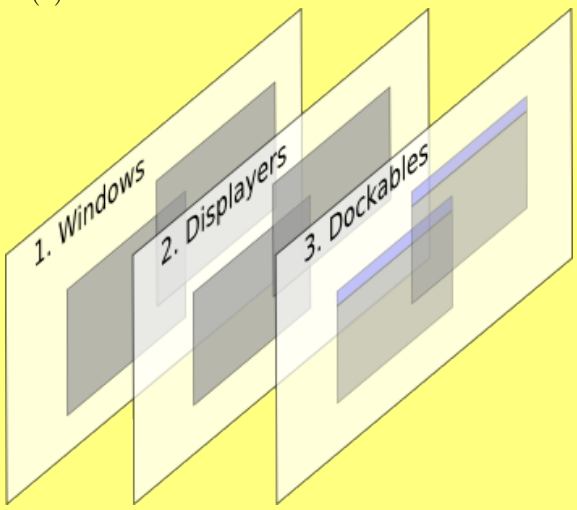


Figure 5: A `ScreenDockStation` with two children floating over a frame.

This station is pretty simple and consists of only 3 layers. Some windows (1), instances of `ScreenDockWindow`, provide a container to show the children. On each window there is a `DockableDisplayer` (2) to paint decorations, and on top is one `Dockable` child (3).



2.4 DockController

A `DockController` holds `Dockable`s, `DockStation`s and other supporting elements together. Most tasks are not handled by the `DockController` but by one of its sub-controllers, e.g. drag and drop is handled by the `DockRelocator`.

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

Clients need to register the roots 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. But there is absolutely no reason why only one controller should exist. A controller has no unique property that would justify a singleton. And not using a singleton 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 all `Dockable`s, e.g. whether `Dockable`s are hideable or not.

2.5.1 Close-Button

In order to show the close-button clients need first to register their `Dockable`s. 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 `Dockable`s, note however that the `Dockable`s 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 consists of the location, size and relationship of all the Dockables and DockStations. DockingFrames offers methods to store this layout persistently. Applications should use persistent layouts because the user certainly does not want to set up his preferred layout everytime when the application restarts.

DockingFrames distinguishes between local and global layout information:

1. Local information describes the relationship between one Dockable and its parent(s). Local information is represented by a chain of DockablePropertys, and each DockStation offers a method getDockableProperty to find the location of one of its children.
2. Global information describes the relationship of an entire tree of Dockables and DockStations. The class DockSituation offers methods to extract and to apply this data.

It should be notet that applications need to handle both local and global information in order to create a truly persistent layout. Local information is needed to store the location of Dockables which are invisible (not in the tree), global information is needed when stopping and starting the application. There are no algorithms implemented to create global information out of local information, and there are only basic algorithms which create local information out of global information. In any case, conversion between these two formats should be considered not to be possible.

For many applications the easiest solution to handle persistent layouts is to use a DockFrontend and completely ignore all the other sections of this chapter (see chapter 3.5).

3.1 Placeholders

Placeholders are an optional extension that allow clients to link global and local information. The idea behind placeholders is, that some Dockables can be assigned a unique identifier. If such a Dockable is removed from a DockStation, then a placeholder remains. At later time when the Dockable is added again to that station, the placeholder can be used to place the element at its former location. Placeholders are stored in the global and the local layout information, and thus build a link.

In Core this mechanism is normally disabled. Clients must implement a new PlaceholderStrategy and install the strategy using the property key PlaceholderStrategy.PLACEHOLDER_STRATEGY. The strategy should be set up before reading a layout, otherwise all placeholders will be marked as invalid and be deleted. The strategy should also be applied to any DockSituation that is created by clients.

An implementation of a PlaceholderStrategy can be found in the example “Persistent Layout: Placeholders”.

Placeholders were introduced in version 1.0.8. One of the reasons they were not used earlier is that they make data structures complex. Also detecting and removing invalid and outdated placeholders requires some work.

Most `DockStation`s use the `PlaceholderList` and the `PlacehoderMap` to manage their `Dockable`s and placeholders.

3.2 Local: DockableProperty

Every `DockStation` can create `DockableProperty`-objects for its children. Each of these `DockablePropertys` contains the position, size, placeholder and/or other data about one child.

Some `DockStation`s are also `Dockable`s. 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.

3.2.1 Creation

How to create a `DockableProperty`? One way is of course just to create new objects using `new XYProperty(...)`. The other way is to retrieve 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 seven `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`.

SplitDockPlaceholderProperty also for `SplitDockStation`. This property stores a placeholder, an identifier whose position is already known to the `SplitDockStation`. If the placeholder is not found, then a backup property can be applied.

SplitDockFullScreenProperty also for `SplitDockStation`. This property points to a child that is maximized.

3.2.2 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`).

`DockableProperty`s 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.

3.2.3 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 the global transformer of the frontent.

3.3 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 for these transformations.

An example featuring several aspects of global layouts is “Persistent Layout: Global”.

3.3.1 Basic Algorithms

Global layout information appears in five 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.

perspective format A lightweight version of the “dock-tree format”, for easy modification by clients. More about perspectives can be found in section 3.4.

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 different formats, but they are free to choose any 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 for example an application has `Dockables` which show only temporary information that will be lost on shutdown anyway.

PlaceholderStrategy This strategy filters placeholders, invalid placeholders are removed from the layout.

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 `DockablePropertys` for the elements. These are the positions were the elements would come to rest if the layout information were converted into a dock-tree.

3.3.2 Basic Usage

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

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

```
1  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.

3.3.3 Reuse existing Dockables

The major drawback of the basic algorithms is that they always create new `Dockables` and `DockStations`. 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 they can reuse existing dock-elements.

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

A side effect of this implementation is that for the basic algorithms no factory seems 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 the originals are 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.

`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.

3.3.4 Exctract local information

It is possible to exctract `DockablePropertys` from a global layout with the help of a `DockSituation`. First the layout data is required in its intermediate format. This data can only be accessed if the client uses its own format to store layout data. As an example, storing the layout of one `DockStation` using XML:

```
1  public void write( StackDockStation station, DockSituation situation,
       XElement out ){
2      DockLayoutComposition intermediate = situation.convert(station);
3      situation.writeCompositionXML(intermediate, out.addElement("layout")
           );
4  }
```

Once the client has acquired the data in its intermediate format it can use `estimateLocations` to assign locations to each node in the tree of compositions. An example using XML:

```
1   public void read( DockSituation situation, XElement in ){
2      // acquire intermediate data
3      DockLayoutComposition intermediate = situation.readCompositionXML(in
           .getElement("layout"));
4
5      // guess locations
6      situation.estimateLocations(intermediate);
7
8      // get the location of the root (which will be null, because the
           root has no parent)
9      DockableProperty location = intermediate.getLayout().getLocation();
10 }
```

It is up to the client to find out which `DockLayoutComposition` represents which `Dockable`. A custom `DockFactory` can help by storing some keys in the layout which can later be identified by the client.

A `DockFrontend` will estimate locations of those missing `Dockables` for which `addEmpty` was called.

If using a `PredefinedDockSituation`, the method `listEstimatedLocations` is of interest as it returns a map of identifier-location pairs. The identifiers are the identifiers of the `Dockables` which were added by the client to the situation.

3.4 Perspectives

Layout information appears in different formats, perspectives is one of these formats. Perspectives offer clients a way to read, modify or build layout information using lightweight objects and keeping typesafety.

In order to work with perspectives clients need access to a `Perspective` object:

- Any `DockSitutation` offers a method `createPerspective` which sets up a new `Perspective` using the current settings of the `DockSituation`.
- `DockFrontend` offers a method `getPerspective`. Clients can provide a `FrontendPerspectiveCache` which basically converts `Dockables` and

DockStations to their counterparts in the perspective API. This is required for clients that introduce their own DockFactorys.

The FrontendPerspectiveCache allows clients to use their own, specialized classes to describe Dockables and DockStations. This may not be necessary for all clients, these clients can make use of the DefaultFrontendPerspectiveCache.

Once a Perspective object is aquired it can be used to directly read and write the xml, binary or the intermediate format. Clients using a DockSituation should use the convert methods to create or apply the intermediate format. Clients using a DockFrontend should use the get/setSetting methods in order to access and apply layouts through the intermediate format.

Perspective creates objects of type PerspectiveElement. There are various subtypes of this interface, in fact for each type of Dockable or DockStation of the framework there is a subtype representing exactly that item (e.g. SplitDockPerspective represents SplitDockStation). Clients are free to move around elements in any way they wish. However, the perspective API does not enforce the correctness of the layout, it is the clients responsibility to build a layout that actually makes sense.

An example showing how to use perspectives to build the layout is “Persistent Layout: Perspectives”.

3.5 DockFrontend

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

Layout information can be stored in xml- or binary-format. The methods write, writeXML, read and readXML will take care of this.

3.5.1 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.

3.5.2 Global

DockFrontend internally uses a PredefinedDockSituation to store the global layout. All root-DockStations and all registered Dockables are automatically added to this situation. The global layout can either be stored on disk 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.

3.5.3 Missing Dockables

The default behavior of `DockFrontend` is to throw away information for missing `Dockables`. 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`:

- 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 `Dockable`s, 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 `Dockable`s 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.

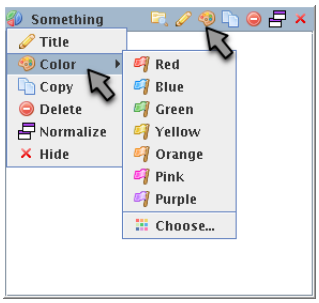


Figure 6: A `Dockable` with a few `DockAction`s in its title and on a popup menu. The action marked by an arrow is the same object just shown in different views.

The example "Actions" shows how to set up some actions.

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. There are 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. Several 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`, 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 have 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` then the method `getSource` is called. This source of actions is intended for general purpose actions and for actions which need a special position in the list of actions (e.g. a close-action needs to be at the very 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 are 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 from these different sources and to put them into a 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 one of its offers.

- Modules which need a list of actions call `getGlobalActionOffers` from `Dockable`. This method is the ultimate piece of code which decides what to show. Usually the method is implemented by returning an instance of `HierarchyDockActionSource`. However, this method can ignore anything that has been said in this chapter and introduce its very own mechanism to collect actions.

Most `Dockables` will utilize `HierarchyDockActionSource` instead of implementing `getGlobalActionOffers`. 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 thing 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 `DockAction`s. 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 `Dockable`s get grouped. When `Dockable`s 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`.

Another example is “Actions: GroupAction” showing an action with more than two keys.

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 `Dockable`s 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 `DockStation`s. `DockStation`s need to apply the same actions to many `Dockable`s. 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 with new views.

4.3.1 Reuse existing view

Whenever possible an existing view should be reused. There are six 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`.

An example for an action that uses an `ActionType` to create its view:

```
1  public class ExampleButtonAction implements ButtonDockAction{
2
3      public <V> V createView( ViewTarget<V> target,
4              ActionViewConverter converter, Dockable dockable ){
5
6          return converter.createView( ActionType.BUTTON, this,
7              target, dockable );
8      }
9
10     public void action( Dockable dockable ){
11         [...]
12     }
13
14     public Icon getIcon( Dockable dockable ){
15         return [...];
16     }
17
18     [...]
19 }
```

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 should be to introduce a new `ActionType`. Only the second step is to write the new action-class. This could result in something like this:

```
1  import bibliothek.gui.Dockable;
2  import bibliothek.gui.dock.action.ActionType;
3  import bibliothek.gui.dock.action.DockAction;
4  import bibliothek.gui.dock.action.view.ActionViewConverter;
5  import bibliothek.gui.dock.action.view.ViewTarget;
6
7  public class CustomAction implements DockAction{
8      public static final ActionType<CustomAction> CUSTOM =
9          new ActionType<CustomAction>( "custom" );
10
```

```
11      public <V> V createView( ViewTarget<V> target,
12              ActionViewConverter converter, Dockable dockable ){
13          return converter.createView( CUSTOM, this,
14                  target, dockable );
15      }
16
17      @Override
18      public void bind( Dockable dockable ){
19          // ignore
20      }
21
22      @Override
23      public void unbind( Dockable dockable ){
24          // ignore
25      }
26
27      public boolean trigger( Dockable dockable ){
28          return false;
29      }
30  }
```

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:

```
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`:

```
1  public class CustomButtonGenerator implements
2          ViewGenerator<CustomAction, BasicTitleViewItem<JComponent>>{
3      public BasicTitleViewItem<JComponent> create(
4              ActionViewConverter converter, CustomAction action,
5              Dockable dockable ){
6
7          return [...]
8      }
9  }
```

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 grab a `DockTitle` when they want to start a drag & drop operation.

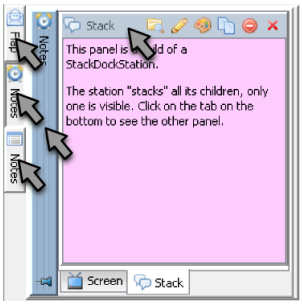


Figure 7: Some `DockTitles`.

5.1 Lifecycle

Any client that wants to show a `DockTitle` needs to specify what *kind* of title it shows and needs to *request* a title.

The *kind* of a title is specified by a `DockTitleVersion`. New `DockTitleVersions` are obtained through the `DockTitleManager` (there is one per `DockController`). Creating a new `DockTitleVersion` requires the calling client to provide a default `DockTitleFactory`.

The *request* for a title is handled by a `DockTitleRequest`. Once a `DockTitleRequest` is created its method `request` can be called to execute the request. Clients should call `install` before using the request and `uninstall` once the request is no longer in use. This way the `DockTitleRequest` will automatically be executed again if the underlying `DockTitleFactory` is exchanged.

Once a `DockTitle` is acquired it must be connected with its `Dockable`. Clients must call the method `bind( DockTitle )` of `Dockable`, this tells the `Dockable` that is has a new title. If the client no longer shows the title it must call `unbind( DockTitle )`.

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

`Dockable`s provide some information about their titles:

- The method `listBoundTitles` returns an list of all `DockTitle`s 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 Implementing a new title

It is possible to replace all the titles in the framework. While the interface `DockTitle` is rather open, a title is responsible to collect all the information it wants to show by itself.

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.

There is only one connection between a module that shows a title and the title itself: the method `changed`. Modules use this method to send `DockTitleEvent`s to the title.

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 `DockAction`s. `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`.

An example showing a custom title is “DockTitle: Custom title”.

5.2.2 Apply the title

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

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

Overriding or implementing `requestChildDockTitle` of `DockStation` allows to exchange the title of all children.

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:

```
1 DockController controller = ...
2
3 DockTitleManager manager = controller.getDockTitleManager();
4 DockTitleVersion version =
5   manager.getVersion( SplitDockStation.TITLE_ID, null );
6 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. The current `DockTheme` can be changed through the method `setTheme`:

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

6.1 Existing Themes

Several `DockThemes` 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 `DockThemes` 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 `StackDockStations` 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:

```
1 DockTheme original = ...
2 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.

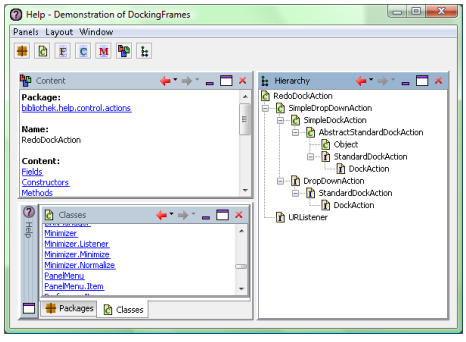


Figure 8: BasicTheme

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.

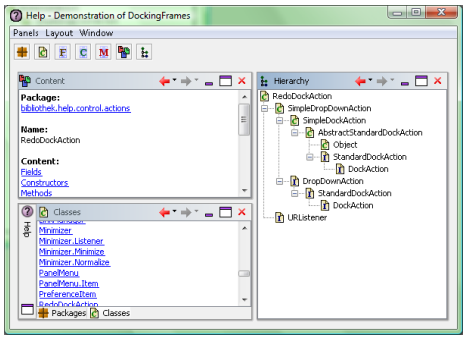


Figure 9: FlatTheme

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.

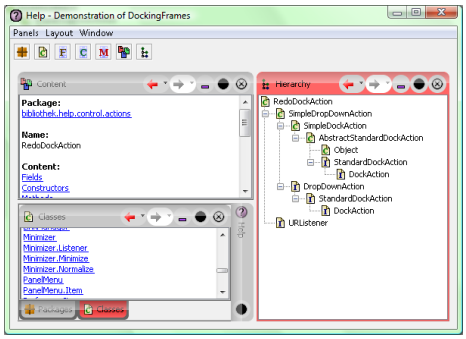


Figure 10: BubbleTheme

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.

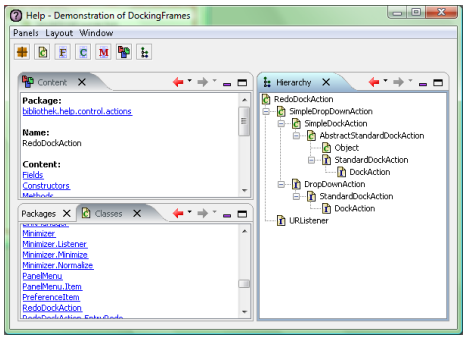


Figure 11: EclipseTheme

`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 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.

6.3 Customizing

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.

6.3.1 UI-Properties

The `UIProperties` distribute properties like colors, texts or fonts in the framework. 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.

While `UIProperties` build upon a map, they can do more than an ordinary map. They report changes in the map through an observer mechanism represented by `UIValues`. Further more they can filter their content through `UIBridges`.

The full list of classes and interfaces building the base for the UI-properties consists of:

- `UIProperties`: The map that connects properties, observers and filters.
- `V`: The type of the properties, e.g. the class `Color`.
- `UIValue`: An observer that is attached to `UIProperties` and receives an event if a property changes. The `UIValue` has to provide information to

the filters, that means an `UIValue` represents the component that is using the property.

- `UIBridge`: An `UIBridge` is a filter between the `UIProperties` and the `UIValue`s. An `UIBridge` can decide to inform an `UIValue` about a changed property at any time. Depending on the target `UIValue` an `UIBridge` may filter the property in different ways.
- UIScheme: a set of default properties and default `UIBridges`.

The implementation gets more complex:

- For each key several `V` properties can be put into the 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 type the `UIValue` has.
- `UIBridge`s are also associated with a `Path`. An `UIBridge` is responsible to handle all those `UIValue`s 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 `Color`s. 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.

There are several subclasses of `UIProperties`, each of these classes handles another kind of property:

- `ColorManager` handles `Color`s.
- `FontManager` handles `Font`s. Rather than distributing `Font`s directly, this class distributes `FontModifiers`. A `FontModifier` can use the default font of a component slightly modify it (e.g. make it italic), or just replace the font.
- `IconManager` handles `Icon`s.
- `TextManager` handles language dependent text.

- `ThemeManager` is not directly a subclass, but offers a similar interface. It is responsible for distributing factories and strategies used all over the framework.

6.3.2 Colors

In order to understand this chapter 6.3.1 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:

```
1 DockController controller = ...
2 ColorManager colors = controller.getColors();
3 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`:

```
1  DockController controller = ...
2  ColorManager colors = controller.getColors();
3  colors.publish( Priority.CLIENT, TitleColor.KIND_TITLE_COLOR, new
       ColorBridge(){
4      public void add( String id, DockColor uiValue ){
5          // ignore
6      }
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 `UIValue`s 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 `DockColor`s 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.

An example showing the same things as the snippets is "UI Properties: Color"

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 various implementations of `ColorScheme` can be used to find keys.

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`.

6.3.3 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 `FontModifier`s. 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 `JComponent`s with the capability to modify their font.

6.3.4 Icons

`Icon`s can be modified through the `IconManager`. The `IconManager` can be accessed through the method `getIcons` of `DockController`. It is an `UIProperties` and offers all the methods that are known from colors and fonts.

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.

6.3.5 Text

Language dependend text is distributed by the `TextManager`, it can be accessed through `DockController.getTexts()`. The `TextManager` is an `UIProperties` and offers all the methods that are known from colors and fonts.

The default text for different languages is stored in several `*.properties` files. These files can be loaded by `ResourceBundle`s. Clients can make use of the class `DefaultTextScheme` to load additional languages into the framework.

6.3.6 Actions

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

6.3.7 Titles

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

6.3.8 Border

Any `Border` can be modified or replaced by a `BorderModifier`.

`BorderModifier`s can be set by the `ThemeManager`. The `BorderModifier` interface works in the same way as the `FontModifier` interface.

6.3.9 Background

The usual background of a `Component` is either grey or transparent. Clients can set a painting algorithm for the background with help of the interface `BackgroundPaint`. Instances of `BackgroundPaint` are applied through the `ThemeManager`.

The method `paint` of `BackgroundPaint` is called every time when a component has to be repainted. The method receives a `PaintableComponent` which offers the standard algorithms to paint border, children and other stuff. The `BackgroundPaint` can freely decide what to paint and in which order to paint.

6.3.10 Drag and drop decorations

During drag and drop operations `DockStation`s use a `StationPaint` to paint decorations. The `StationPaint` can be set through the `ThemeManager`.

6.3.11 Displayers

A `DockableDisplayer` is a wrapper around a `Dockable` painting some decorations like a title or some border. All `DockStation`s make use of `DockableDisplayer`s to paint their children. `DockableDisplayer`s are created by `DisplayerFactory`s which are accessible through the `ThemeManager`.

Once a `DockableDisplayer` is created it cannot be replaced until either the theme is exchanged or the displayer marks itself as invalid. In the later case the displayer needs to call the `discard` method of any `DockableDisplayerListener` that was added to it.

Clients are free to implement new displayers or extend existing displayers. Any new displayer should be a focus-cycle-root, assuming the displayer uses `Swing`-components the code below can be used to setup the correct focus management:

```
1  // the new displayer
2  DockableDisplayer displayer = ...
3
4  JComponent root = (JComponent)displayer.getComponent();
5
6  root.setFocusable( true );
7  root.setFocusCycleRoot( true );
8  root.setFocusTraversalPolicy(
9      new DockFocusTraversalPolicy(
10         new DisplayerFocusTraversalPolicy( displayer ), true ));
```

7 Stations in depth

`DockStations` are the most complex classes, or modules, of the framework. Some of the stations offer fine tuning that could be interesting for the more ambitious projects, the goal of this chapter is to show some of the these advanced configuration options. In no way can this chapter replace studying the API documentation and the source code itself. Before reading this chapter you should read about the Basics (page 9), it offers a nice overview of the stations.

7.1 ScreenDockStation

This station packs its children into free floating panels. These panels are called windows, and can be moved and resized by the user.

7.1.1 Window type

The windows (of type `ScreenDockWindow`) are usually `JDialog`s. The reason for this is, that a `JDialog` is guaranteed to float over its parent frame. In some applications however a `JDialog` is not the correct tool, in these cases a client can implement custom `ScreenDockWindow`s or reuse and configure some of the existing windows. For this to happen a client has to implement a `ScreenDockWindowFactory` and use the property key `ScreenDockStation.WINDOW_FACTORY` to set it. The factory may be an instance of `DefaultScreenDockWindowFactory` with non-default settings.

- `ScreenDockDialog` is the default window.
- `ScreenDockFrame` is exactly the same as the dialog, just using a `JFrame` instead.
- `InternalDockDialog` is a dialog that can appear on a `JDesktopPane`. If using this window, the client should also install an `InternalFullscreenStrategy`.
- `DefaultScreenDockWindowFactory` offers several methods to configure the dialog, including an option to show the OS dependent controls.

New implementations of `ScreenDockWindow` should be subclasses of `AbstractScreenDockWindow`. This class offers everything a window needs except the window-container itself.

Should a client implement a completely new `ScreenDockWindow`, then it will be required to implement a matching `ScreenDockFullscreenStrategy`. Accessing the `MagnetController` and use its `start` method will allow the new window to feature attraction and stickiness.

7.1.2 Window configuration

The look and behavior of each default `ScreenDockWindow` can be configured, some of the available options are:

- Whether the window is transparent
- Whether the window can be resized by the user
- To move if the title of a `Dockable` is dragged
- What kind of border to paint

All these options are set by the `ScreenDockWindowConfiguration`, which really is just a factory creating new instances of `WindowConfiguration`. The factory can be set using the property key `ScreenDockStation.WINDOW_CONFIGURATION`.

Do not subclass `WindowConfiguration`. Use the `set`-methods to change its properties.

7.1.3 Stickiness and attraction

What happens when a window is dragged near another window? Or if two windows touch each other and one of them is dragged away? The framework offers some special behavior in these cases:

- A window dragged near another window can be *attracted* to the fixed window. The dragged window will move itself a little bit such that the sides of the windows touch each other.
- A window dragged away from a neighbour can be *sticking* to the neighbour. If one window is dragged, the neighbours are automatically dragged as well.

The exact behavior of each window is defined by the `AttractorStrategy`. Clients can set up their own strategy by using the property key `ScreenDockStation.ATTRACTOR_STRATEGY`.

The actual implementation of attracting and sticking windows is provided by the `MagnetStrategy`, which can be replaced using the property key `ScreenDockStation.MAGNET_STRATEGY`. Clients providing their own `MagnetStrategy` may be interested in using the `StickMagnetGraph`, a class that analyzes the layout of the windows and their dependencies.

7.1.4 Fullscreen

Since a standard `JDialog` can not be maximized, and the `ScreenDockWindows` usually are `JDialogs`, the framework must find out on its own when a window is maximized. This property, also called “fullscreen”, is defined by the interface

`ScreenDockFullscreenStrategy`. Replacing it usually makes little sense, but it can be done using the property key `ScreenDockStation.FULL_SCREEN_STRATEGY`.

It seems like there can be only one definition of "fullscreen". But because the framework does nowhere enforce that a `ScreenDockWindow` indeed is a real window, it does neither define what a "screen" could be. And it is really hard to globally define "fullscreen" when there is no definition of "screen".

7.1.5 Drop size

What happens when a `Dockable` is dropped onto a `ScreenDockStation`? It is put into a window, but how big should this window be? The default behavior of the framework is to look at the current size of the `Dockable`, and keep that size. Another solution could be to make sure the `Dockable` has its preferred size. A client can change the default behavior by implementing a `ScreenDropSizeStrategy`, and installing it using the property key `ScreenDockStation.DROP_SIZE_STRATEGY`.

7.2 SplitDockStation

The `SplitDockStation` organizes its children in a layout that might look at first glance like a grid, but in reality is a binary tree. Each `Dockable` is a leaf in that tree. Any node has an orientation (horizontal or vertical) telling how its children are aligned, and a `divider` property telling the relative size of the children. Users modify the tree through drag and drop, but clients can access and modify the tree programatically.

7.2.1 The tree

The layout-tree is represented by objects of type `SplitNode`. Each node can be seen as a rectangle on the screen, and the children of each node must be within that rectangle. To be more precise, there are four different types of nodes in the tree:

- The `Root` node really has no special meaning, it is just a wrapper around another node promoting that other node to be the true root.
- A `Node` has exactly two children. The node has an orientation that tells how the children are aligned, and it has the `divider` property, a `double` between `0` and `1.0`, telling the size of the first child in respect to the size of the node itself.
- A `Leaf` is a wrapper around a `Dockable`.
- Finally a `Placeholder` is not visible to the user. When a `Leaf` is removed from the tree a `Placeholder` may remain. This placeholder can later be converted back into the `Leaf` that was removed.

Clients can access the tree by calling `SplitDockStation.getRoot`.

Usually modifying the tree directly is a bad idea. When modifying the tree, be aware of:

- The tree does not validate itself, if a client creates an invalid tree the application will simply show a very strange layout or start throwing exceptions.
- Removing or adding branches to the tree does not automatically remove or add `Dockables`.

Instead of accessing the tree directly, and perhaps causing a lot of damage, clients can make use of the class `DockableSplitDockTree`. A client can create a new `SplitDockTree` and call `SplitDockStation.dropTree` to replace the current layout of the station.

`SplitDockTree` may look very hard to use, many methods need to be called just to build a simple tree. But there are some advantages that should be considered:

- This class is very safe to use, it is nearly impossible to create an invalid tree with it.
- The tree is built from bottom to top, it is an ideal tool to have different methods build and deliver different branches of the tree.
- A `SplitDockGrid` can be converted into a `SplitDockTree`

7.2.2 Divider

Between each two `Dockables`, there is a little gap. The user can grab this gap with the mouse and move it around. For the very unlikely case that a client needs to modify this behavior, there exists the interface `SplitDividerStrategy`. The interface itself really does not do much, it gets a `Component` and there are some suggestions in the API documentation of what the interface should do with that `Component`. The divider strategy is changed by using the property key `SplitDockStation.DIVIDER_STRATEGY`.

7.2.3 LayoutManager

There are many actions a user can perform: making a `Dockable` "fullscreen" (the station hides all other children), drop a new `Dockable`, adjust the sizes of the children, or adjust the size of the entire station. In older versions code that reacted to or implemented these actions was either part of the `SplitDockStation` itself, or of the `SplitNodes`. New developments showed, that it was nearly impossible to modify the behavior. To solve the issue `SplitLayoutManager` was introduced, now these user actions are forwarded to one of the methods

of SplitLayoutManager, and the manager may decide either to call the old code, or to chose a custom solution. It is unlikely that a client ever needs to change the SplitLayoutManager, but it can be done with the property key SplitDockStation.LAYOUT_MANAGER.

7.3 StackDockStation

The StackDockStation acts like a JTabbedPane, only one of its children is visible at any time. The look of this station highly depends on the current DockTheme. Each theme defines a StackDockComponent, and this object is responsible for creating, painting and layouting the tabs. While the BasicTheme makes use of a JTabbedPane, all the other themes make use of a class called CombinedStackDockComponent. This means that tabs usually have the same behavior, even if they look differently.

Much of what is written in this chapter does not apply to the BasicTheme, because the abilities of the JTabbedPane are very limited compared to the abilities of CombinedStackDockComponent.

7.3.1 TabPane

The class CombinedStackDockComponent is shared by the EclipseTheme, BubbleTheme and FlatTheme. While the class is responsible for painting and layouting tabs, the real magic happens in its superclass AbstractTabPane, which implements the interface TabPane. TabPane is completely independent from StackDockStation, and only CombinedStackDockComponent brings the two modules together.

TabPane does not need to know about StackDockStation, it just has to offer some methods to add and remove tabs. This allows to make a separation: TabPane is responsible for painting tabs, StackDockStation is responsible for deciding which tabs exist.

TabPane was designed to show Dockables, and as a result it has much more features than a JTabbedPane. To understand the next chapters, it is certainly a good idea to have an overwiev of the different parts of a TabPane.

Figure 12 shows some tabs how they could appear in any application. The items to the left are called *tabs*, while the button on the right side are part of the *info-component*.

Figure 13 shows what happens if there is not enough space to show all tabs. An additional component shows up, a menu called *tab-menu* allows the user to select the tabs that are not visible.

Finally figure 14 shows a single tab. Each tab can contain an icon, some text, and perhaps some buttons.

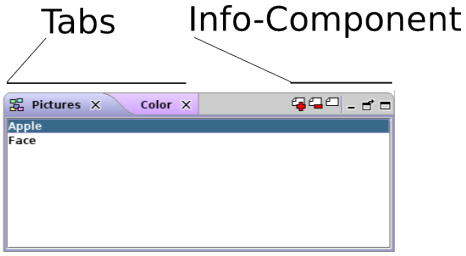


Figure 12: A `TabPane` with enough space shows some tabs, and some `DockActions` that are associated with the currently selected `Dockable`.

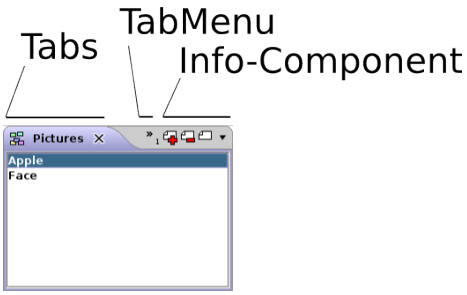


Figure 13: A `TabPane` that has not enough space can show a *tab-menu*, this menu allows the user to select `Dockables` that are otherwise not accessible.

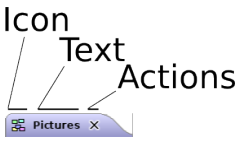


Figure 14: A single tab shows the title information of the `Dockable` it is associated with, including some of its `DockActions`.

A `TabPane` is nearly the same as a `JTabbedPane`. The tabs are represented by the interface `Tab`, the menu showing inaccessible `Dockables` are of type `TabMenu`. An additional info-component of type `LonelyTabPaneComponent` shows `DockActions`.
Clients that want to implement a new `TabPane` should subclass either `AbstractTabPane` or `CombinedStackDockComponent`.

7.3.2 Tab content

What information should a tab show? Well, the icon and the title-text of the `Dockable` seems like a good idea. But since many tabs have to share limited space, some developers may decide it would be a good idea to limit the length of the text, or not to show any icons. The `TabContentFilter` allows clients to override the default icon, text and tooltip of a tab. A new `TabContentFilter` can be installed by using the property key `StackDockStation.TAB_CONTENT_FILTER`.

7.3.3 Tab configuration

When a tab runs out of space some drastic actions have to be performed. For example the tab could stop painting its icon, this gives at least 20 free pixels. Or it could stop painting its title text, to make sure the icon remains visible. What exactly happens in such a situation depends on the `TabConfiguration`, and this configuration is created by the factory `TabConfigurations`.

Clients can change the configuration using the property key `StackDockStation.TAB_CONFIGURATION`.

7.3.4 Header layout

Like a `java.awt.Container` using a `LayoutManager`, a `TabPane` makes use of a `TabLayoutManager`. The `TabLayoutManager` defines the size and location of all tabs, menu and info-component. The `TabLayoutManager` receives a `TabPane` and by calling methods like `putOnTab` it can tell the `TabPane` how to present the `Dockable`s. Clients can set a custom layout manager using the property key `TabPane.LAYOUT_MANAGER`.

The default implementation of `TabLayoutManager` is `MenuLineLayout`. This class uses a factory called `MenuLineLayoutFactory` to configure some of the details of the layout. The method `createOrder` is of special interest, it returns a `MenuLineLayoutOrder` which tells order and weight of tabs, menu and info-component.

7.4 FlapDockStation

The `FlapDockStation` shows a line of "buttons", if clicking one a window opens showing a `Dockable`. At any time, only one `Dockable` can be shown.

7.4.1 Button content

The buttons consists of different parts, as can be seen in figure 15. These parts have different jobs:

Knob The knob provides an empty area where the user can grab the button. It ensures that drag and drop is always possible.

Icon That is just the icon of the `Dockable`.

Text The title-text of the `Dockable`.

Children If the button represents a `DockStation`, then the button can show actions to quickly select one of the children of the `DockStation`.

Actions `DockAction`s associated with a `Dockable` can be shown on the button as well.

Which of these elements show up depends on the class `ButtonContent`. It offers some conditions with which the framework can decide whether to

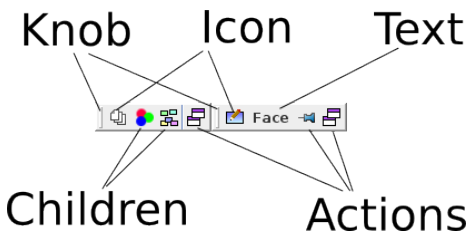


Figure 15: Two flap-buttons. The left button shows a stack of `Dockable`s, while the right button shows a single `Dockable`.

show an element or not. A `ButtonContent` can be set using the property key `FlapDockStation.BUTTON_CONTENT`.

There are several default setups for `ButtonContent` available as constants in `ButtonContent` itself. The conditions are modeled by `ButtonContentCondition`, an interface that can be implemented by clients.

The buttons are `DockTitle`s, the default `DockTheme`s all use `BasicButtonDockTitle` as button. Clients can install custom buttons by accessing the `DockTitleManager` and using the key `FlapDockStation.BUTTON_TITLE_ID`.

7.4.2 Button actions

The flap-button can show `DockAction`s that are associated with the `Dockable`. But since space is a limited resource, usually not all actions are shown. The default behavior is to show only actions which are annotated with `ButtonContentAction`. Clients may change the behavior by implementing the interface `ButtonContentFilter` and installing it using the property key `FlapDockStation.BUTTON_CONTENT_FILTER`.

8 Drag and Drop

Naturally, draging and droping of `Dockables` is a key feature of the framework. Funny enough, the code actually involved in DnD is rather small compared to other modules of the framework.

8.1 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 their own `DockRelocator`. If they do, they have to implement a new `DockControllerFactory` in order to install their customized class. The method `createRelocator` is responsible for creating the new object.
This factory has then to be given to the constructor of a `DockController`.

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

8.2 Deciding what element to drag

The `Relocator` needs to know where and when the user presses and moves the mouse. There are two solutions to this problem: either let the `Relocator` know what `Components` are shown, or remotely control the `Relocator`. The first solution is achieved with `DockElementRepresentatives`, the second solution is achieved with the `RemoteRelocator`.

8.2.1 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`.

While the internal implementations of `DockElementRepresentative` are handled automatically by the framework, clients introducing new representatives will have to call the methods `addRepresentative` and `remoteRepresentative` of `DockController` to install or uninstall the item.

`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.

8.2.2 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 RemoteRelocator.

A RemoteRelocator can be obtained 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.

There is a second interface called DirectRemoteRelocator. Instances can be obtained 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 the event would always be consumed anyway.

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

8.3 Deciding where to drop an element

A relocator needs to find the one DockStation on which the Dockable should be dropped. There is a default search algorithm which just orders all DockStations by importance and visits them, and there are some interfaces which can influence the search.

8.3.1 Search

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

1. An ordered list of all potential destinations is built. A `DockStation` is a potential destination if it is visible (`isStationShowing` of `DockStation`), not the dragged `Dockable` nor one of its children. Each station offers a set of `DockStationDropLayers` (`getLayers`), the layers decide whether the mouse is over a station or not. The order of the destinations depends on the priority of the layers, the parent-child relations between the stations and between the `Windows` on which the stations are.
2. Then the method `prepareDrop` of `DockStation` is called. This method checks whether the station really is a good destination, if so it returns a `StationDropOperation`. The first station returning an operation is the destination.
3. The method `draw` of the new operation is called, the method `destroy` on the old operation. The new operation will paint some markings to give a visual feedback to the user.

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

Most of the work for drag and drop is done by the `DockStations` themselfs, the `DockRelocator` just connects them. In order to complete the task the following methods and interfaces should be used:

- `DockStation.accept` and `Dockable.accept` tells the station whether a child-parent relation is possible.
- `DockController.getAcceptance` allows access to the global `DockAcceptance`, an additional restriction that should be checked before allowing a drag and drop operation.
- To paint on the station, a `StationPaint` should be used. A `StationPaint` can be accessed through the `ThemeManager`.

8.3.2 Drop

Once the user releases the mouse, `Dockable` is dropped. The framework will call the method `execute` of `StationDropOperation`.

- The `Dockable` may just be dropped aside of all the other children of the station. All that happens is that the `DockStation` gets a new child.

- The `Dockable` may be dropped over another child of the station. In this case the station may decide to combine the two children. The future parent `DockStation` will access a `Combiner` which defines how exactly two `Dockable`s can be merged into one, usually the answer is by creating a new `StackDockStation`. Clients can replace the current `Combiner` through the `ThemeManager`.
- If the dragged `Dockable` is a `DockStation` itself, it may be feasible to merge the parent and the new child `DockStation` into one station. The interface `Merger` is responsible for that. Clients can replace the default `Merger` by calling `DockRelocator.setMerger`.

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

8.4 Restrictions

Not every possible `DockStation` is a good or valid target for a dragged `Dockable`. The framework applies a set of restrictions to drag and drop operations, these restrictions are implemented by "acceptance tests". Each acceptance test can veto against some child parent relations. The usual reasons why clients would want to implement their own tests consist of:

- 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`.

Acceptance tests are performed during the drag and drop operation, but also if one of `DockStation.drop` methods is called. The acceptance tests are implemented by these methods:

- Every `Dockable` has two methods called `accept`. One method checks whether the `Dockable` can be put directly onto some new parent, the other method checks whether the `Dockable` can be combined with an already existing child.
- Each `DockStation` has a method `accept`. This method tells whether some `Dockable` can become a child of the `DockStation`.
- And then there are `DockAcceptance`s. A `DockAcceptance` has `accept`-methods too. These methods get a `DockStation` and some `Dockable`s, 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.

Acceptance tests are performed by the potential destination `DockStation`. The `DockStation` is the first module that knows where a `Dockable` will land. Handling acceptance tests allows the station to cut down the amount of work it does, and to try alternative actions (e.g. a "put" instead of a "merge" action) if some future configuration does not pass the tests.
The drawback is, that a `DockStation` can break the mechanism by just not performing the tests.

8.5 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`.

8.6 Animations

During drag and drop, the framework may show some animations to help the user understand what effects dropping the `Dockable` would have. The animations usually involve moving or resizing the `Dockable`s that are not dragged. These animations are implemented with help of the `Span` interface. Each `Span` object represents some gap in the layout, a `Span` basically is a self mutating integer, to be understood as the size of a gap in pixels. Each `DockStation`

may use several Spans at the same time, and an animation may involve multiple Spans changing their value simultaneously.

There are two sides involved in the animations:

- The DockStations define *where* and *when* the animations appear. For example a FlapDockStation can trigger an animation to insert empty space between each of its buttons.
- The Spans define *how* an animation looks like. For example a Span could be implemented such that an animation starts slowly and increases its speed over time.

Clients cannot tell a DockStation where the animations take place, but they can influence how the animations look like. To do that, clients need to implement both the Span and the SpanFactory interface. The DockStation will configure the Span, by associating different sizes (number of pixels) to different SpanModes, and later by telling the Span which SpanMode currently is required. In return the Span will call the resized method of the SpanCallback whenever the size of the gap changes.

To install a new SpanFactory clients can:

- Use the property key DockTheme.SPAN_FACTORY to globally change the factory.
- Use ThemeManager.setSpanFactory to change the factory only for one class of DockStations.
- Calling setSpanFactory of BasicTheme *before* the theme is installed.

Some themes, like the EclipseTheme, deliberately disable the animations by installing the NoSpanFactory.

9 Preferences

The preference system allows the user to change settings which are otherwise not accessible. An example would be the shortcut for maximizing a `Dockable` (`ctrl+m`). The preference system makes a sharp distinction between model and view, clients are free to integrate the model in their own view, or to create a new model and using the standard view. Figure 16 shows the simple version of the standard view with some random preferences.

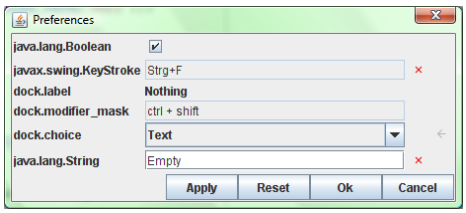


Figure 16: The `PreferenceDialog` showing some random preferences.

Additionaly the prerefence API offers mechanism to persistently store preferences.

9.1 Model

The model is an adapter to the view and presents some properties as a list of modifiable items. Whether the model represents properties of the framework or custom properties is unimportant for the view or the persistent storage mechanism to work.

9.1.1 Preference

A preference is an abstract concept. One preference represents some property of the framework (or of the client). A preference is a set of meta-informations about a property:

Path A unique identifier, is used by the persistent storage to identify a property.

Value The current value of the property.

TypePath Tells how to work with `Value`. For example how to present the value to the user (as text, as image...) or how to store the value. An object of type `Path` is used to represent the `TypePath`.

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

`Value` is some `Object` and `TypePath` tells the view how to cast `Value` in order to use it. If `TypePath` were a `Class` then there would never be doubt whether the correct cast is performed. But `TypePath` is a `Path` and hence an additional indirection is introduced.
The reason for this is that the same `Object` might need different treatment in different situations. E.g. an `Integer` could just be an int, it could be a natural number or it could be an int from the range 1 to 100.

There is an interface `Preference` and a class `DefaultPreference` which bring this preference-abstraction to code. It is not necessary to use them, they are just here to simplify things.

9.1.2 PreferenceModel

The `PreferenceModel` is the basic module of the preference system. A `PreferenceModel` is a list of preferences (the abstraction, not the interface). It often acts as mediator between some unspecified storage mechanism for properties and the user interface. The methods `read` and `write` are used to access that covered storage mechanism. To transfer values into the model `read` is called, to transver values to the storage mechanism `write` is called.

`DefaultPreferenceModel` is the standard implementation of `PreferenceModel`. Its entries are objects of type `Preference`.
Several models can be combined using a `MergedPreferenceModel`.

There are several subclasses of `DefaultPreferenceModel` for various settings that can be made. For example `EclipseThemeModel` handles properties of `EclipseTheme`.
There are also many implementations of `Preference` for various properties of the framework. The API-documentation reveals more.

9.1.3 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.

9.2 View

A `PreferenceModel` is best displayed in a `PreferenceTable`. This table will show a label, an editor and operations for each preference.

A `PreferenceTreeModel` can be displayed in a `PreferenceTreePanel`. It will show not only a `PreferenceTable` but also a `JTree` where the user can select which sub-model to edit.

Further more the `PreferenceDialog` and the `PreferenceTreeDialog` are available. These dialogs offer the options to apply the settings, to cancel editing and to reset all preferences to their default value.

9.2.1 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.

9.2.2 Operations

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 these 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.

Operations might be confusing at first, but they can be really useful. The strength of operations is that they are handled automatically, and that they need not much code.

9.3 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 system. These preferences are called *natural*, while the others are called *artificial*. The method `isNatural` of `PreferenceModel` can be used to distinguish them.

The distinction between natural and artificial preferences might seem strange. But this allows to use different types of storage mechanisms at the same time.

9.4 Lifecycle

This section describes the best way how to use a `PreferenceModel`.

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.

10 Extensions

Extensions allow libraries to add new code to the framework, this code will be treated as if it were always part of the framework. Basically it is a plug-in mechanism. Currently there are not many points where an extension can be inserted, new extension-points will be added when needed. Developers which are interested in using the extension mechanism should contact the developers directly at `http://forum.byte-welt.net/forumdisplay.php?f=69`.

`Extension`s are collected by the `ExtensionManager`. Any module can call `load` to load extensions that match some `ExtensionName`.

Extensions were introduced in 1.0.8 to allow the usage of the glass-components. The glass-components could not be added directly to the framework due to licencing issues.
In version 1.1.1 a number of new extension points were added in order to support the new Toolbar Extension.

10.1 Extension Points

A number of extensions exists. The following list only includes the extensions of the `Core` library.

New choices in the preferences
: *`ChoiceExtension.CHOICE_EXTENSION`*: Allows to add additional entries to a `Choice`. A `Choice` is a preference allowing the user to pick one of many items.

Extending a `DockTheme`
: *`DockThemeExtension.DOCK_THEME_EXTENSION`*: Allows to modify a `DockTheme` during the installation process.

Additional colors
: *`ColorScheme.EXTENSION_NAME`*: Allows to extend or override the contents of a `ColorScheme`.

New `DockableProperty`
: *`PropertyTransformer.FACTORY_EXTENSION`*: Adds new factories to the default list of `DockablePropertyFactory`.

Merging `Dockables`
: *`DefaultDockRelocator.MERGE_EXTENSION`*: Adds new `Mergers` to the `DockRelocator`, these `Mergers` are executed after the default `Mergers`.

Modify drag and drop
: *`DefaultDockRelocator.INSERTER_EXTENSION`*: Allows an extension to completely override the default drag and drop behavior.

Attraction of floating `Dockables`
: *`ScreenDockStation.ATTRACTOR_STRATEGY_EXTENSION`*: Installs additional rules to find out whether to windows of a `ScreenDockStation` attract each other or stick together.

Persistent storage
DockSituation.DOCK_FACTORY_EXTENSION: Adds new factories for persistently storing the layout of `DockStation`s and `Dockable`s.

Modify titles
DockTitleVersion.DOCK_TITLE_VERSION_EXTENSION: Extends the mechanism that creates new `DockTitle`s, the extension can inject new types of titles.

Modify displayers
DisplayerFactory.DISPLAYER_EXTENSION: Injects additional factories for creating `DockableDisplayer`s, these factories have a higher priority than the default factories.

Additional text
TextManager.TEXT_EXTENSION: Adds more language files to the `TextManager`.

Modify the window configuration
DefaultScreenDockWindowConfiguration.CONFIGURATION_EXTENSION: Modifies the default configuration of the windows of a `ScreenDockStation`.

Images during drag and drop
DefaultDockableMovingImageFactory.FACTORY_EXTENSION: Modifies the image that is shown beneath the mouse during a drag and drop operation.

Modifying `ScreenDockStation`
ScreenDockStation.STATION_EXTENSION: Installs an algorithm that modifies the behavior of a `ScreenDockStation` when dropping a `Dockable`.

Handling new types in perspectives
DefaultFrontendPerspectiveCache.CACHE_EXTENSION: The extension adds new types of `PerspctiveElement` to the factory which converts normal `DockElement`s to `PerspectiveElement`s.

Extending a `DockFrontend`
DockFrontend.FRONTEND_EXTENSION: This extensions knows which `DockFrontend`s exist and can modify them.

10.2 Glass Extension

The glass-extension adds new icons and a new way to paint tabs to the `EclipseTheme`. Clients only need to ensure that the libraries `docking-frames-ext-glass.jar` and `glasslib.jar` are part of the classpath. The `ExtensionManager` will then automatically load this extension.

The Glass Extension is licensed by a modified version of the LGPL. You are prohibited to use the library if your application provides “pornography, racialistics, violence, or the like material”.

10.3 Toolbar Extension

The Toolbar-extension adds toolbars, a set of new `DockStation`s and `Dockable`s, to the framework. Clients need to ensure that the library `docking-frames-ext-toolbar.jar` is part of the classpath, the `ExtensionManager` will then load and install the extension automatically.

There are several examples included in the tutorial, these examples are stored in folders called “Toolbar”.

11 Properties

There are a number of interesting settings whose effects are deeply hidden within the framework. Properties are an easy way to access these settings and change them. Properties are represented by the class `DockProperties` which can be accessed through `getProperties` of `DockController`.

`DockProperties` is nothing else than a map. Instances of `PropertyKey` are used as keys. The type of the value depends on the key and the map is typesafe. With the help of a `DockPropertyListener` any object can be informed immediately when a value changes.

There are a number of keys and the remainder of this chapter will list all of the keys that are present in version 1.1.1. If not explicitly said otherwise, then any change in the properties will have an immediate effect. This list is only an overview, please have a look at the API documentation or the source code to find out about types and default values.

Some of these properties are accessed and changed by `DockThemes`. It is still possible to override these properties, but clients should be careful and ensure not to break the theme.

11.1 Themes

These properties either are only used by some `DockThemes`, or are changed by `DockThemes`.

Colors of the `BasicTheme`
: *`BasicTheme.BASIC_COLOR_SCHEME`*: Sets a strategy (acting like a map) that tells which `Colors` to use when the `BasicTheme` is selected.

Colors of the `BubbleTheme`
: *`BubbleTheme.BUBBLE_COLOR_SCHEME`*: Sets a strategy (acting like a map) that tells which `Colors` to use when the `BubbleTheme` is selected.

Actions of the `BubbleTheme`
: *`BubbleTheme.ACTION_DISTRIBUTOR`*: Tells where a `DockAction` should appear: on a tab, on a info-component or on the titel of a `Dockable`.

Stacking `Dockables`
: *`DockTheme.COMBINER`*: The default strategy for merging two `Dockables` into one, for example by putting them together on a `StackDockStation`.

`Dockable` decorations
: *`DockTheme.DISPLAYER_FACTORY`*: The displayer is a `Component` between a `Dockable` and its parent, the displayer adds some decorations, for example a border, to the `Dockable`.

Drag indicator
: *`DockTheme.DOCKABLE_MOVING_IMAGE_FACTORY`*: Tells what image to show when the user drags a `Dockable` around.

`Dockable` **selection**
: *DockTheme.DOCKABLE_SELECTION*: A `Component` which allows the user to select the focused `Dockable`.

Start `Dockable` **selection**
: *DockableSelector.INIT_SELECTION*: If the user hits this `KeyStroke` a window pops up, the user can select the new focused `Dockable` on that window.

Background
: *DockTheme.BACKGROUND_PAINT*: This strategy paints the background of various components, it may also make some component transparent.

Borders
: *DockTheme.BORDER_MODIFIER*: An adapter that receives a `Border`, the adapter may replace the original border with a custom border.

Animations during drag and drop
: *DockTheme.SPAN_FACTORY*: During drag and drop, the `Span`s are used for an animation where empty space seem to appear beneath the `Dockable`.

Painting during drag and drop
: *DockTheme.STATION_PAINT*: A strategy used to paint on `DockStation`s during a drag and drop operation.

Text rotation on titles
: *DockTitle.ORIENTATION_STRATEGY*: This strategy knows whether the orientation of a `DockTitle` is horizontal or vertical, it then tells how to rotate the text on the title.

Colors of the `EclipseTheme`
: *EclipseTheme.ECLIPSE_COLOR_SCHEME*: Sets a strategy acting like a map that tells which `Color`s to use when the `EclipseTheme` is selected.

`EclipseTheme`**: when to paint icons**
: *EclipseTheme.PAINT_ICONS_WHEN_DESELECTED*: Tells whether icons should be painted on tabs when they are not selected.

`EclipseTheme`**: how to paint tabs**
: *EclipseTheme.TAB_PAINTER*: A factory and strategy that defines the look of the tabs used by the `EclipseTheme`.

`EclipseTheme`**: tab configuration**
: *EclipseTheme.THEME_CONNECTOR*: Tells where to paint tabs, and which `DockAction`s to show on these tabs.

Colors of the `FlatTheme`
: *FlatTheme.FLAT_COLOR_SCHEME*: Sets a strategy (acting like a map) that tells which `Color`s to use when the `FlatTheme` is selected.

Actions of the `FlatTheme`
: *FlatTheme.ACTION_DISTRIBUTOR*: Tells where a `DockAction` should appear: on a tab, on a info-component or on the titel of a `Dockable`.

Size of icons
IconManager.MINIMUM_ICON_SIZE: Defines the minimum size of icons, any icon smaller than this size will be treated as if it would be bigger.

11.2 Stations

Properties related to `DockStation`s.

`FlapDockStation`**: button content**
FlapDockStation.BUTTON_CONTENT: Decides what content to show on the buttons, e.g. whether to show an icon or some text.

`FlapDockStation`**: button actions**
FlapDockStation.BUTTON_CONTENT_FILTER: Filters the `DockAction`s that are shown on the button.

`FlapDockStation`**: persistent layout**
FlapDockStation.LAYOUT_MANAGER: Strategy to store and load properties, like the size, of `Dockable`s that are not necessarily known to the `FlapDockStation`.

`FlapDockStation`**: minimum size**
FlapDockStation.MINIMUM_SIZE: The minimum size of the station itself, this is specially important when the station does not have any children.

`FlapDockStation`**: windows**
FlapDockStation.WINDOW_FACTORY: A factory creating a `FlapWindow`s, this window is used to show one of the children of a `FlapDockStation`.

`ScreenDockStation`**: Attraction and stickiness**
ScreenDockStation.ATTRACTOR_STRATEGY: Defines which two windows are attracted or sticked together.

`ScreenDockStation`**: Where the screen ends**
ScreenDockStation.BOUNDARY_RESTRICTION: A definition of the boundaries of the screen, and how windows behave when they are pushed against the boundaries.

`ScreenDockStation`**: Size of new windows**
ScreenDockStation.DROP_SIZE_STRATEGY: Tells how big a window is when it is created.

`ScreenDockStation`**: Fullscreen on mouse click**
ScreenDockStation.EXPAND_ON_DOUBLE_CLICK: Tells whether double clicking with the mouse can command a window to switch into the fullscreen mode.

`ScreenDockStation`**: Definition of "fullscreen"**
ScreenDockStation.FULL_SCREEN_STRATEGY: Strategy deciding whether a window is in fullscreen mode or not.

`ScreenDockStation`**: Implementation of stickiness**
ScreenDockStation.MAGNET_STRATEGY: An algorithm that implements

magnetization, the algorithm is responsible for finding out which two windows are attract each other or stick together, and how to react when one of the windows is moved aorund.

`ScreenDockStation`**: Issues with focus**
`ScreenDockStation.PREVENT_FOCUS_STEALING_DELAY`: A short delay in which a window cannot steal the focus if the owner window of the `ScreenDockStation` changed.

`ScreenDockStation`**: Configuration of windows**
`ScreenDockStation.WINDOW_CONFIGURATION`: A factory creating configurations for the windows, for example whether the window is resizeable.

`ScreenDockStation`**: Implementation of windows**
`ScreenDockStation.WINDOW_FACTORY`: A factory creating new windows.

`SplitDockStation`**: Moving the gaps**
`SplitDockStation.DIVIDER_STRATEGY`: This strategy is responsible for changing the location of the gaps when the user grabs them with the mouse.

`SplitDockStation`**: Handling the layout**
`SplitDockStation.LAYOUT_MANAGER`: Decides about size and location of the children, about what happens if the size of the `SplitDockStation` changes, and which drop operations are possible.

`SplitDockStation`**: Maximize a child**
`SplitDockStation.MAXIMIZE_ACCELERATOR`: Tells which keys the user has to hit to maximize a child.

`StackDockStation`**: How the tabs look like**
`StackDockStation.COMPONENT_FACTORY`: A factory creating a `StackDockComponent`, this component is responsible for painting all the tabs.

`StackDockStation`**: Reaction on dropping a** `Dockable`
`StackDockStation.IMMUTABLE_SELECTION_INDEX`: Whether dropping a `Dockable` changes the selected `Dockable` or not.

`StackDockStation`**: The contents of the tabs**
`StackDockStation.TAB_CONTENT_FILTER`: An adapter telling what icon and text to show on the tabs.

`StackDockStation`**: How a small tab looks like**
`StackDockStation.TAB_CONFIGURATIONS`: A configuration telling how the tabs behave if space is running out.

`StackDockStation`**: Where the tabs show up**
`StackDockStation.TAB_PLACEMENT`: Tells on which side (left, top, right, bottom) the tabs appear.

11.3 Miscellaneous

Some properties that do not fit in any other category.

Applets and webstart

DockController.RESTRICTED_ENVIRONMENT: A Java application has limited rights when executed as applet or from webstart. The framework however needs some special rights, for example to monitor the position of the mouse. If these rights are not available, the framework activates some workarounds (which are not very efficient). In such cases the framework is called to *be running in a restricted environment*. The property is set automatically, and usually clients need only read access. They can change the property, with the danger that the application no longer works afterwards.

Show text on buttons

DockAction.BUTTON_CONTENT_FILTER: `DockAction`s can be shown on buttons. Usually the button contains only the icon of the action, but this strategy allows to show the text (usually used in menus) of the actions as well.

Importance of actions

DockActionImportanceOrder.ORDER: Tells the order of importance of a set of `DockAction`s. In a situation where there is not enough space to show all actions, the least imporant actions will disappear first. Clients can also use the annotation `DockActionImportance` to mark the importance of actions.

Layout of tabs

TabPane.LAYOUT_MANAGER: This strategy is used by `TabPane` to decide where to show tabs, menus or info-component.

Always show tabs

SingleTabDecider.SINGLE_TAB_DECIDER: Usually tabs only appear when some `Dockable`s are stacked. This strategy tells whether `Dockable`s that are not stacked should still feature a tab.

Drop down menu on stacks

CombinedMenuContent.MENU_CONTENT: If there is not enough space to show all tabs on a `StackDockStation`, a menu appears where the user can see the missing tabs. How exactly this menu looks like and how it is implemented is defined by this property.

Keep track of `Dockables`

PlaceholderStrategy.PLACEHOLDER_STRATEGY: Tells the *placeholder* of a `Dockable`. The placeholder is left behind if a `Dockable` is removed from a station, this way the framework still knows the old place of the item. Clients using `DockFrontend` or the Common project should not change this property.

Disable items

DisablingStrategy.STRATEGY: This strategy tells which items (titles, tabs, actions, `Dockable`s) are disabled. Items that are disabled have a different color and do not react to user input.

No stacking during drag and drop
DockRelocatorMode.NO_COMBINATION_MASK: If this `KeyStroke` is pressed during a drag and drop operation, the framework will not combine `Dockable`s. For example the framework will not create a new `StackDockStation`.

Forced floating during drag and drop
DockRelocatorMode.SCREEN_MASK: If this `KeyStroke` is pressed during a drag and drop operation, the only valid target of the operation is a `ScreenDockStation`.

Close `Dockables`
DockFrontend.HIDE_ACCELERATOR: If this `KeyStroke` is hit, the currently focused `Dockable` is closed - assuming the `Dockable` can be closed in the first place.

Dealing with AWT components
AWTComponentCaptureStrategy.STRATEGY: Allows clients to implement code to take images from AWT components. There are some default strategies available, going from "nice" to "ugly workaround".

Tooltips in applets and on webstart
GlassedPane.TOOLTIP_STRATEGY: In a restricted environment the framework will use an invisible `Component` to catch all `MouseEvent`s. This component is also responsible for showing tooltips. This strategy allows clients to modify the tooltip behavior: how they are created, and what they show.

11.4 Gimmicks

These properties are not really necessary, they might be interesting for applications with a lot customization.

`DockStation`: default icon
PropertyKey.DOCK_STATION_ICON: This icon is shown by a `DockStation` unless some other icon is set.

`DockStation`: default title
PropertyKey.DOCK_STATION_TITLE: This text is shown by a `DockStation` unless some other text is set.

`DockStation`: default tooltip
PropertyKey.DOCK_STATION_TOOLTIP: This tooltip is shown by a `DockStation` unless some other tooltip is set.

`Dockable`: default icon
PropertyKey.DOCKABLE_ICON: This icon is shown by a `Dockable` unless some other icon is set.

`Dockable`: default title
PropertyKey.DOCKABLE_TITLE: This text is shown by a `Dockable` unless some other text is set.

`Dockable`: default tooltip
PropertyKey.DOCKABLE_TOOLTIP: This tooltip is shown by a `Dockable` unless some other tooltip is set.

11.5 Glass Extension

The Glass Extension provides some additional properties.

Detailed configuration
EclipseThemeExtension.GLASS_FACTORY: This factory creates the "glass effect".

Size of tabs
CGlassExtension.SMALL_TAB_SIZE: Allows to make tabs a little bit smaller.

11.6 Toolbar Extension

The Toolbar Extension provides some additional properties.

Shrinking, Expanding, Stretching
ExpandableToolbarItemStrategy.STRATEGY: Each toolbar-item can appear in three different sizes, this strategy tells which sizes are available for which items. Clients usually have no need to implement this interface, instead the `Dockable`s should implement `ExpandableToolbarItem`.

`ToolbarGroupDockStation`**: Painting between** `Dockable`**s.**
ToolbarGroupDockStation.DIVIDER_STRATEGY_FACTORY: This strategy allows the `ToolbarGroupDockStation` to paint some borders between its children.

`ToolbarGroupDockStation`**: header component**
ToolbarGroupDockStation.HEADER_FACTORY: With this factory a client can add a `Component` at the top end of a `ToolbarGroupDockStation`.

`ToolbarGroupDockStation`**: scrollbars**
ToolbarGroupDockStation.SCROLLBAR_FACTORY: This factory creates scrollbars that are shown on a `ToolbarGroupDockStation`.

`ToolbarDockStation`**: gap between children**
ToolbarDockStation.GAP: An integer telling how much space should be between the children of a `ToolbarDockStation`.

`ToolbarDockStation`**: gap between border and children**
ToolbarDockStation.SIDE_GAP: An integer telling how much space should be between the border and the children of a `ToolbarDockStation`.

Toolbar behavior
ToolbarStrategy.STRATEGY: This strategy tells how different parts of the Toolbar Extension fit together. For example it can tell whether a `Dockable` can be a child of a toolbar-station, or not.