DockingFrames 1.0.6 - Core

Benjamin Sigg

December 3, 2008

Contents

Abstract

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

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

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

1 Notation

This document uses various notations.

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

Tipps and tricks are listed in boxes.

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

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

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

2 Basics

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

The controller is a `DockController`, the floating panels are `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 }
```

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 `Dockable`s 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 `Dockable`s directly to the station, but this is the easy way.

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

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

2.2 Dockable

A `Dockable` represents a floating panel, it consists at least of some `JComponent` (the panel it represents), some `Icon` and some text for a title. Each `Dockable`

can be dragged by the user and dropped over a `DockStation`.

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

A small example:

```
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, there is no need for clients to track these properties, store and load them or to delete them once they are no longer used. It is the responsibility of the framework to do so. Second: No special component within the framework or programming technique gets an unfair advantage over others, everything has to be designed in a way that it can work with any new, unknown, crazy other otherwise unexpected kind of `Dockable`.

2.3 DockStation

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

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

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

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

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

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

ScreenDockStation Shows each child in its own window.

Clients can implement new `DockStations`. 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 `DefaultDockables` are put onto the station in lines `3,4`, the method `drop` is available in all `DockStations`.

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

2.4 DockController

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

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

Clients need to register the root of their `DockStation`-`Dockable`-trees. They can use the method `add` of `DockController` to do that. All children of the root will automatically be registered as well. If a `DockStation` is not registered anywhere, it just does not work properly. For `Dockables` 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 `Dockables` and `DockStations`.
`DockRelocator`: handles drag and drop operations, can create a `Remote` to play around without user interaction.
`DoubleClickController`: detects double clicks on `Dockables` or on components which represent `Dockables`.
`KeyBoardController`: detects `KeyEvents` on `Dockables` or on components which represent `Dockables`.

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

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

2.5 DockFrontend

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

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

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

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

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

2.5.1 Close-Button

In order to show the close-button clients need first to register their `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 means the position, size and relationships of all the `Dockable`s and `DockStation`s. To store this layout on a harddrive and later to load it again is a great help for the user, he does not need to setup the layout over and over again.

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

3.1 Local: DockableProperty

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

Some `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.

3.1.1 Creation

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

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

In line `1` we get some unknown `Dockable`. In line `3` the `DockStation` which is at the top of the tree of stations and `Dockable`s is searched. Then in line `4` the location of `dockable` in respect to `root` is determined.

There are five `DockableProperties` present in the framework.

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

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

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

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

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

3.1.2 Usage

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

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

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

`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.

3.1.3 Storage

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

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

3.2 Global: DockSituation

The layout of a whole set of `Dockable`s and `DockStation`s 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, i.e. from the dock-tree (built by stations and `Dockable`s) to an xml-file. A `DockSituation` uses various factories to transform one format into another.

3.2.1 Basic Algorithms

Global layout information comes in four formats:

dock-tree format The set of `Dockable`s and `DockStation`s as they are seen by the user.

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

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

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

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

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

A `DockSituation` uses different factories and strategies for these conversions:

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

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

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

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

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

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

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

3.2.2 Basic Usage

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

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

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

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

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

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

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

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

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

```
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.

3.2.3 Extended Algorithms

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

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

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

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

What happens if a `PredefinedDockSituation` finds layout information for an element, has all the necessary factories but not the element itself? The default behavior is to ignore the information. However it is possible to use

backup-DockFactories. These backup factories will create new elements if they were missing. They are also used when reading raw format and the original factory is missing. These backup factories are added through addBackup, they have to use a BackupFactoryData as intermediate format.

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

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

3.2.4 Extended Usage

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

An example can look like this:

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

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

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

3.3 DockFrontend

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

3.3.1 Local

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

3.3.2 Global

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

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

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

delete Deletes a layout from memory.

getSettings Gets a set of names for the different layouts.

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

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

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

3.3.3 Missing Dockables

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

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

binary format and convert this data later once an appropriate `DockFactory` becomes known.

"Empty entries" are best to be used if a client already knows the identifiers of all the `Dockable`s 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 `Dockable`s which are then automatically registered.

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

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

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

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

If a strategy allows to store anything and a client often uses different identifiers for their `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.

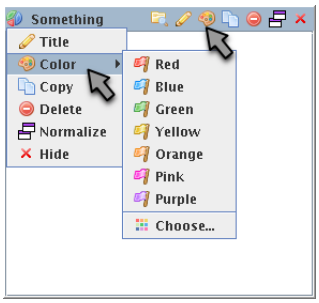

Figure 1: A `Dockable` with a few `DockAction`s in its title and on a popup menu. The action marked by an arrow is the same object just shown in different views.

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

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

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

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

4.1 Show Actions

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

4.1.1 List of Actions

`DockAction`s never travel alone in this framework. They always travel with other actions in a `DockActionSource`. Actions can be added or removed from `DockActionSource`s 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 `LocationHint`s.

`LocationHint`s 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 `Hint`s and `Origin`s can be written.

4.1.2 Source of Actions

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

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

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

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

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

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

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

- Modules which need a list of actions call `getGlobalActionOffers` from `Dockable`. This method is the ultimate piece of code which decides what to show. If need by this method can ignore anything else that has been said in this chapter and introduce its very own mechanism to collect actinos. Most `Dockable`s however will create a field holding a `HierarchyDockActionSource`. This special source observes the hierarchy of a `Dockable` and changes its content automatically. `Dockable`s 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 `DockActionOffer`s or `getGlobalActionOffer` methods which do not just collect actions. There are already mechanisms to introduce `DockAction`s and they should suffice for every possible situation.

4.2 Standard Actions

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

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

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

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

Menu A menu just contains a list of other `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`.

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.

4.3.1 Reuse existing view

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

In a real world example that will look like this:

```
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 in creating a custom action should be to write the new class (or interface) and declare the new type. The second step would be to call `createView`. The third step to implement the remaing methods. The result of these steps could look like this:

```
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

6 Themes

7 Drag and Drop

8 Preferences

9 Properties