DockingFrames 1.0.7 - Common

Benjamin Sigg

February 1, 2009

Contents

1 Introduction **3**

2 Notation **3**

3 Basics **3**
- 3.1 Concepts . . . 4
- 3.2 Hello World . . . 4
  - 3.2.1 Setup controller . . . 5
  - 3.2.2 Setup stations . . . 5
  - 3.2.3 Setup dockables . . . 6

4 Foundation **7**
- 4.1 Dockables . . . 7
  - 4.1.1 SingleCDockable . . . 7
  - 4.1.2 MultipleCDockable . . . 8
  - 4.1.3 Visibility . . . 9
  - 4.1.4 Mode . . . 9
- 4.2 Stations . . . 10
  - 4.2.1 All in one: CContentArea . . . 10
  - 4.2.2 Center area: CGridArea . . . 11
  - 4.2.3 Minimized: CMinimizeArea . . . 12
  - 4.2.4 Grouping Dockables: CWorkingArea . . . 12

5 Locations **13**
- 5.1 For a single dockable: CLocation . . . 13
- 5.2 For a group of dockables: CGrid . . . 13
- 5.3 For all dockables: layout . . . 15
  - 5.3.1 Persistant Storage . . . 15
  - 5.3.2 Dealing with lazy creation and missing dockables . . . 16

6 Actions **16**
- 6.1 CButton . . . 17
- 6.2 CCheckBox . . . 17
- 6.3 CRadioButton . . . 17
- 6.4 CMenu . . . 18
- 6.5 CDropDownButton . . . 18
- 6.6 CBlank . . . 19
- 6.7 System Actions . . . 19
- 6.8 Custom Actions . . . 20

7 Customizing: small but impressive **20**
- 7.1 Color . . . 20
- 7.2 Font . . . 21
- 7.3 Size . . . 21
- 7.4 Maximizing . . . 21

8 Menus **21**

1 Introduction

`DockingFrames` is an open source Java Swing framework. This framework allows to write applications with floating panels. A floating panel is a `Component` that can be moved around by the user.

`DockingFrames` consists of two libraries, `Core` and `Common`. `Common` provides advanced functionalities that are built on top of `Core`. `Common` is a wrapper around `Core` and requires `Core` to work.

This document covers only `Common`, `Core` has its own guide. Not all the details of `Common` are described, but this document gives a nice oversight of the more important aspects.

You can utilize `Common` without understanding `Core`. But knowing at least some basics about `Core` will make life much easier.

2 Notation

This document uses various notations.

Any element that can be source code (e.g. a class name) and project names are written mono-spaced 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`).

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.

3 Basics

`Common` is partitioned into three sub-projects.

For clients the sub-project `common` is the most interesting. This project is the main layer above `Core` and also the most advanced layer in the whole framework.

The sub-project `facile` builds a basis for the features of `common`. In theory `facile` could be used without `common` but in practice the classes and interfaces are clearly designed to be used by or together with `common`.

Finally `support` consists of classes that are just used by `facile` and `common`. These classes have few to none relations with `DockingFrames` and can be used alone as well.

3.1 Concepts

In the understanding of `Common` an application consists of one main-window and maybe several supportive frames and dialogs. The main-window is most times a `JFrame` and the application runs as long as this frame is visible. The main-window consists of several panels. Each panel shows some part of the data. E.g. the panels of a browser could be the "history", the "bookmarks" and the open websites.

`Common` is an additional layer between panels and main-frame. It separates them and allows the user to drag & drop panels.

To do so each panel gets wrapped into a `CDockable`. These `CDockable`s are then put onto `CStation`s. A controller (`CControl`) handles the movement of the dockables.

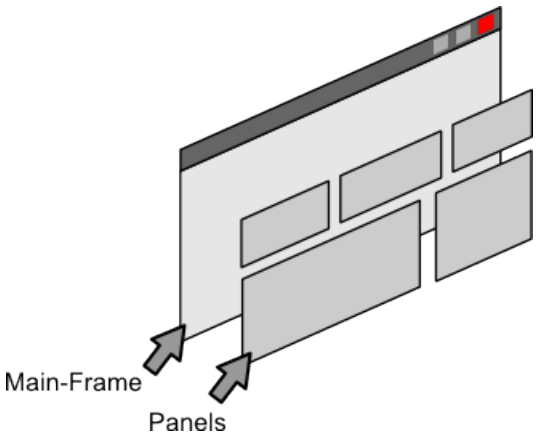

Figure 1: The standard application without `Common`. A main-frame and some panels that are put onto the main-frame.

3.2 Hello World

This chapter introduces the very basic vocabulary and shows a first example. In depth discussions of the concepts and implementations follow in the chapters afterwards.

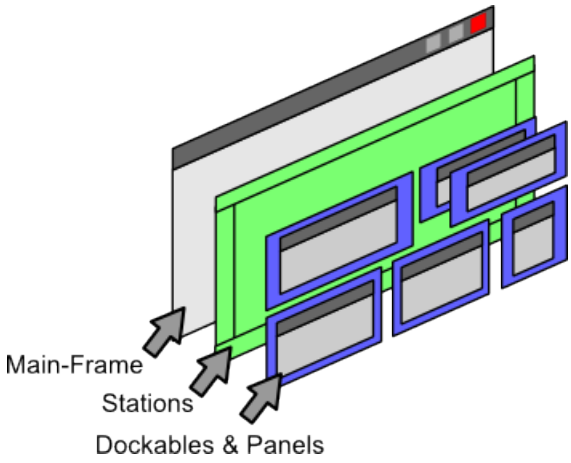

Figure 2: An application with `Common`. The panels are wrapped into dockables. The dockables are put onto stations which lay on the main-frame. Dockables can be moved to different stations.

3.2.1 Setup controller

The first step should be to create a `CControl`. `CControl` has more than one constructor, for this example it is sufficient to use the one constructor which requires only a `JFrame`. `CControl` monitors the state of this frame and ensures that the windows of `Common` are only visible when the frame is visible. Also dialogs created by the framework will have this frame as parent.

The code to create the controller looks like this:

```
1  public class Example{
2      public static void main( String[] args ){
3          JFrame frame = new JFrame();
4          frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
5
6          CControl control = new CControl( frame );
7
8          ...
```

3.2.2 Setup stations

The second step is to setup the layer between main-frame and dockables. There are different `CStation`s available. For example the `CMinimizeArea` shows minimized `CDockable`s. Instead of creating the stations manually one can also use a `CContentArea`. The `CContentArea` is a panel that consists of five stations. In the center a grid of dockables is shown, at the border four areas are reserved for minimized dockables.

There is always a default-`CContentArea` available, it can be accessed by calling `getContentArea` of `CControl`. If required additional `CContentArea`s can be created by the method `createContentArea` of `CControl`.

A `CContentArea` is a `JComponent`, so its usage is straight forward. Line `10` is the important new line in this code:

```
1  public class Example{
2      public static void main( String[] args ){
3          JFrame frame = new JFrame();
```

```
4
5           frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
6
7           CControl control = new CControl( frame );
8
9           frame.setLayout( new GridLayout( 1, 1 ) );
10          frame.add( control.getContentArea() );
11
12          ...
```

`CControl` always creates an additional station for handling externalized dockables.

3.2.3 Setup dockables

A `CDockable` is the thing that can be dragged and dropped by the user. A `CDockable` has a set of properties, e.g. what text to show as title, whether it can be maximized, what font to use when focused, and so on. `CDockable`s are divided into two categories: "single" and "multi" dockables. These categories are explained later, for this first example single dockables are the correct choice. Single dockables are represented by the interface `SingleCDockable`. The class `DefaultSingleCDockable` provides an easy to use implementation. In the code below new single dockables are created in lines `23-25` and `43-48`. They need to be registered at the `CControl` in lines `27-29`, otherwise they cannot be shown. Optionally the initial position can be set like in line `33` and `36`. There is no need to set the position of the first dockable `red`: since it is the first it gets per default all the space.

```
1   import java.awt.Color;
2   import java.awt.GridLayout;
3
4   import javax.swing.JFrame;
5   import javax.swing.JPanel;
6
7   import bibliothek.gui.dock.common.CControl;
8   import bibliothek.gui.dock.common.CLocation;
9   import bibliothek.gui.dock.common.DefaultSingleCDockable;
10  import bibliothek.gui.dock.common.SingleCDockable;
11
12  public class Example{
13      public static void main( String[] args ){
14          JFrame frame = new JFrame();
15
16          frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
17
18          CControl control = new CControl( frame );
19
20          frame.setLayout( new GridLayout( 1, 1 ) );
21          frame.add( control.getContentArea() );
22
23          SingleCDockable red = create( "Red", Color.RED );
24          SingleCDockable green = create( "Green", Color.GREEN );
25          SingleCDockable blue = create( "Blue", Color.BLUE );
26
27          control.add( red );
28          control.add( green );
29          control.add( blue );
30
31          red.setVisible( true );
32
33          green.setLocation( CLocation.base().normalSouth( 0.4 ));
34          green.setVisible( true );
```

```
35
36          blue.setLocation( CLocation.base().normalEast( 0.3 ) );
37          blue.setVisible( true );
38
39          frame.setBounds( 20, 20, 400, 400 );
40          frame.setVisible( true );
41      }
42
43      public static SingleCDockable create( String title, Color color ){
44          JPanel background = new JPanel();
45          background.setOpaque( true );
46          background.setBackground( color );
47
48          return new DefaultSingleCDockable( title, title, background );
49      }
50  }
```

4 Foundation

This chapter focuses on the foundation of `Common`: `CControl`, the stations and dockables.

4.1 Dockables

As mentioned in the previous chapter `CDockable`s fall in one of two categories: "single" or "multiple". Only one instance of a single dockable may exist during runtime, but many (or none) instance of a multiple dockable may exist. In many cases both kind of dockables have the same behavior, but there are some exceptions when it comes to the storage of their location.

Every `CDockable` needs to be registered at a `CControl`, the methods with name `add` can be used. They need to be made visible by calling `setVisible` of `CDockable`.

The interface `CDockable` has some awkward methods whose implementation is already described in the documentation. `CDockable` is not intended to be implemented by clients, but to be used by them. There is a subclass `AbstractCDockable` which provides the correct implementation for these awkward methods. Even in the framework itself no class (except `AbstractCDockable`) implements `CDockable` directly. The only reason for the existence of `CDockable` is to provide an abstraction from the implementation.

A `CDockable` is not a `Dockable`, but internally references a `Dockable`. This `Dockable` is always of type `CommonDockable`. It can be accessed through the method `intern` of `CDockable`. Clients should avoid modifying this `Dockable` directly.

4.1.1 SingleCDockable

The representation of a single dockable is `SingleCDockable`. A single dockable is created once, added to the control and made visible. It remains in memory

until explicitly removed from the `CControl` or the application terminates.

In order to store attributes (like the position) persistently each `SingleCDockable` requires a unique identifier.

Clients best use a `DefaultSingleCDockable`. A `DefaultSingleCDockable` can be used like a `JFrame`, for example it also has a content-pane, has methods to set the title-text, etc.

Examples for single dockables could be:

- A browser has one panel "history", the panel is shown on a single dockable.
- A view that is most of the time invisible. A single dockable is created lazily the first time when the view is shown.

4.1.2 MultipleCDockable

`MultipleCDockable` are used if the number of instances is not known prior to runtime. Each kind of `MultipleCDockable` is associated with a `MultipleCDockableFactory`. The framework can delete or create new instances of this kind of dockable whenever they are needed.

Clients are required to install the `MultipleCDockableFactory` before using any `MultipleCDockable`. There is a class `DefaultMultipleCDockable` which should provide all the features a client needs.

An example:

```
1  CControl control = ...
2
3  MultipleCDockableFactory<MyDockable, MyLayoutInformation> = new ...
4  control.addMultipleDockableFactory( "unique_id", factory );
5
6  MyDockable dockable = new ...
7  control.add( dockable );
```

Notice that in line `4` a unique identifier needs to be assigned to the factory.

Implementing a `MultipleCDockableFactory` is easy. There is a method to read and to write meta-information from or to a `MultipleCDockable`. Meta-information itself is a `MultipleCDockableLayout` which has methods to write or read its content to a stream (e.g. to file). There are no restrictions to what meta-information really is.

Examples for multiple dockables are:

- A text-editor can show many documents at the same time. Each document is shown in its own dockable.
- A 3D modeling software allows to see the modeled object from different angles. Each camera is a dockable.

In `Common` each `CDockable` requires to have a unique identifier. The framework will automatically create an identifier for `MultipleCDockables`.

Why the distinction between single and multiple dockables? The algorithms to store and load the layout (place and size of dockables) can either use existing objects or create new dockables. Using existing objects is preferred because the overhead of creation can be - at least for complex views - high. Single and multiple `CDockable`s represent this gap.

4.1.3 Visibility

A dockable is either visible or invisible. The user cannot interact with the dockable unless it is visible. There is more than one path to change visibility.

The direct approach is to call the method `setVisible` of `CDockable`. This method will show the dockable at its last known location.

A dockable is made visible implicitly if it is added to any station. This can happen if for example using a `CGrid` like explained in chapter 5.

Finally the user can make a dockable invisible by clicking on its close-button. Every subclass of `DefaultCDockable` has the method `setCloseable` to change whether the user can click away the element.

Visibility can be monitored with a `CDockableStateListener`. Either for a single dockable by adding the listener directly to the dockable, or globally by adding the listener to `CControl`. An example:

```
1  CDockable dockable = ...
2
3  dockable.addCDockableStateListener( new CDockableAdapter(){
4    @Override
5    public void visibilityChanged( CDockable dockable ){
6      System.out.println( "Visibility changed to '" +
7                           dockable.isVisible() + "'" );
8    }
9  });
```

The default behavior of the close-action is to call `setVisible` with `visible` set to `false`, so overriding this method is an easy way to introduce some additional code that is executed directly before the dockable closes.

The close-action can be replaced by calling `putAction` with the key `ACTION_KEY_CLOSE` of `CDockable`. The action can be replaced at any time. More about actions in chapter 6.

If the method `setLocation` of `AbstractCDockable` is called before the dockable is made visible, then the dockable is made visible at the supplied location. More about locations in chapter `sec:location`.

4.1.4 Mode

If a `CDockable` is visible then it always is in an extended-mode. The extended mode tells something about the behaviour of the dockable and where it is placed. There are four extended modes available:

normalized The normal state of a dockable. It is placed on the main-frame of the application, but only covers a fraction of the main-frame.

maximized A maximized dockable takes all the space it gets and often covers other dockables.

minimized A minimzed dockable is not directly visible. Only a button at one edge of the main-frame indicates the existance of the dockable. If the button is pressed then the dockable pops up. As soon as it loses focus it disapears again.

externalized The dockable receives its own window. Per default the window is an undecorated `JDialog` and child of the main-frame.

Users can change the extended mode either by dragging the dockable to a new area, or by clicking some buttons that are visible in the title of each dockable.

Clients can access and change the extended mode by calling `getExtendedMode` and `setExtendedMode` of `CControl`. A dockable has no extended mode if not visible. Furthermore clients can forbid a dockable to go into some extended modes. Methods like `setMaximizable` of `DefaultCDockable` allow that. Finally clients can exchange the button that must be pressed by the user by calling `putAction` of `AbstractCDockable`. Keys for `putAction` are declared as `String` constants in `CDockable` with names like `ACTION_KEY_MINIMIZE`.

4.2 Stations

Stations are needed to place and show `CDockable`s. A station provides the `Component`(s) (e.g. a `JPanel` or a dialog) that are the parents of the dockables. Stations are represented through the interface `CStation`.

`CStation`s delegate most of their work to some `DockStation` of `Core`. Like dockables a `CStation` requires a unique identifier. This identifier is used to persistently store and load layout information.

Currently only the existing `DockStation`s from `Core` are truly supported by `Common`. The `StateManager` makes a few assumptions what station is associated with what mode, e.g. a `FlapDockStation` is associated with mode “minimized”. Future versions of the framework might be designed more open, allowing developers to add new modes or other associations. Some improvements were already introduced in version 1.0.7.

4.2.1 All in one: CContentArea

The preferred way to create stations is to use a `CContentArea`. A `CContentArea` is not a single `CStation` but a panel containing many stations. Each content-area has a center area where dockables are layed out in a grid, and four small areas at the border where dockables show up when they are minimized.

There is a default-`CContentArea` present and can be accessed through `getContentArea` of `CControl`. A content-area can later be used like any other `Component`:

```
1  JFrame frame = ...
2  CControl control = ...
3
4  CContentArea area = control.getContentArea();
5  frame.add( area );
```

If more than one content-area is needed then clients can use `createContentArea` of `CControl` to create additional areas. These additional areas can later be removed through `removeContentArea`. The default content-area cannot be removed.

The default content-area is create lazily. There is no obligation to use or create it, clients can as well directly call `createContentArea` or not use them at all.

While `CContentArea` has a public constructor clients should prefer to use the factory method `createContentArea`. In future releases the constructor might be changed.

To place dockables onto a content-area a `CGrid` can be of help. With the method `deploy` the content of a whole `CGrid` can be put onto the center area. More about `CGrid` and other mechanisms to position elements are listed up in chapter 5.

4.2.2 Center area: CGridArea

A `CGridArea` is kind of a lightweight version of `CContentArea`. A grid-area contains normalized and maximized dockables. Other than a content-area it cannot show minimized dockables.

`CGridArea`s should be created through the factory method `createGridArea` of `CControl`. If it is no longer required it can be removed through the method `removeStation`.

Like `CContentArea` a `CGridArea` has the method `deploy` to add a whole set of dockables quickly to the area.

Usage of a grid-area could look like this:

```
1  JFrame frame = ...
2  CControl control = ...
3
4  CGridArea center = control.createGridArea( "center" );
5  frame.add( center.getComponent() );
```

Notice that in line `5` the method `getComponent` has to be called. This method returns the `Component` on which the station lies.

Some more things that might be interesting:

- A grid-area implements `SingleCDockable`, hence it can be a child of another area. Remember that the area must be manually added to the `CControl` as dockable.
- The method `setMaximizingArea` influences of what happens when a child of the area gets maximized. If `true` was given to the method then the child gets maximized within the boundaries of the grid-area. Otherwise the child might cover the area or even be transfered to another area.

4.2.3 Minimized: CMinimizeArea

Most things that were said for `CGridArea` hold true for `CMinimizeArea` as well. A minimize-area should be created through `createMinimizeArea` of `CControl`.

4.2.4 Grouping Dockables: CWorkingArea

The `CWorkingArea` is a subclass of `CGridArea`. The difference between them is, that the property `working-area` is `false` for a grid-area, but `true` for a `CWorkingArea`.

Having this property set to `true` places some constraints on the station:

- Children of this station cannot be put moved to another station if that other station shows dockables in normalized mode. For a user this means that children can only be minimized, maximized or externalized, but not dragged away.
- The user cannot drag dockables away from the station unless they are already children of the station.
- If the station has no children then it appears as grey, empty space which does not go away.
- Children of a working-area are not stored for temporary layout. For the user this means that applying a layout does neither affect the station, or dockables that can be put onto the station.

`CWorkingArea`s can be used to display a set documents. For example in an IDE (like `Eclipse` or `Netbeans`) each source file would get its own `CDockable` which then is put onto the working-area.

The children of a `CWorkingArea` are often good candidates for being `MultipleCDockable`s.

5 Locations

Location means position and size of a dockable. A location can be relative to some parent of a dockable or it can be fix.

5.1 For a single dockable: CLocation

The location of a single dockable is represented by a `CLocation`. The method `getBaseLocation` of `CDockable` gets the current location and the method `setLocation` changes the current location.

Most subclasses of `CLocation` offer one or more methods to optain new locations. For example a `CGridAreaLocation` offers `north`. While `CGridAreaLocation` represents just some `CGridArea`, the location optained through `north` represents the upper half of the grid-area. Clients can chain together method calls to create locations:

```
1  CGridAreaLocation root = ...
2  CDockable dockable = ...
3
4  CLocation location = root.north( 0.5 ).west( 0.5 ).stack( 2 );
5  dockable.setLocation( location );
```

The chain of calls in line `4` creates a location pointing to the upper left quarter of some grid-area. Assuming there is a stack of dockables in that quarter, the location points to the third entry of that stack. In line `5` the location of `dockable` is set, the framework will try to set `dockable` at the exact location but cannot make any guarantees (e.g. if there is no stack in the upper left quarter, then framework cannot magically invent one).

To create a root-location clients can call one of the static factory methods of `CLocation` or directly instantiate the location. Calling the factory methods of `CLocation` is preferred.

Setting the location of a dockable `a` to the location of another dockable `b` will move away `a` from its position. As an example:

```
1  CDockable a = ...
2  CDockable b = ...
3
4  CLocation location = a.getBaseLocation();
5  b.setLocation( location );
```

If `a` should remain at its place then the method `aside` of `CLocation` can create a location that is near to `a`, but not exactly `a`'s position:

```
5  b.setLocation( location.aside() );
```

`CLocation` is a wrapper around `DockableProperty`. While each `DockableProperty` has its own API and concepts, `CLocation`s unify usage by providing the chain-concept. The chain-concept allows some typesafety and should reduce the amount of wrongly put together locations.

5.2 For a group of dockables: CGrid

Sometimes it is necessary to set the position of several dockables at once. For example when the application starts up a default layout could be created. If

dockables are minimized or externalized the position can simply be set with `CLocations`. If dockables are shown normalized on a grid-area, a working-area, or the center of a `CContentArea` then things get more complex. Using `CLocation` would require a precise order in which to add the dockables, and some awkward coordinates to make sure they are shifted at the right place when more dockables become visible.

`CGrid` is a class that collects dockables and their boundaries. All this information can then be put onto a grid-like areas in one command. Furthermore a `CGrid` can also automatically register dockables at a `CControl`. An example:

```
1  CControl control = ...
2
3  SingleCDockable single = new ...
4  MultipleCDockable multi = new ...
5
6  CGrid grid = new CGrid( control );
7
8  grid.add( 0, 0, 1, 1, single );
9  grid.add( 0, 1, 1, 2, multi );
10
11 CContentArea content = control.getContentArea();
12 content.deploy( grid );
```

The `CControl` created in line `6` will call the `add`-methods of `control` (line `1`) with any dockable that is given to it. In lines `8,9` two dockables are put onto the grid. The numbers are the boundaries of the dockables. In line `12` the contents of the grid are put onto `content`. The dockables `single` and `multi` will be arranged such that `multi` has twice the size of `single`.

Boundaries are relative to each other, there is no minimal or maximal value for a coordinate or size. `CGrid` is able to handle gaps and overlaps, but such defections might yield awkward layouts.

Make sure not to add a dockable twice to a `CControl`. If using a `CGrid` the `add` method of `CControl` must not be called.
Also note that there is a second constructor for `CGrid` that does not have any argument. If that second constructor is used, then the `CGrid` will not add dockables to any `CControl`.

Dockables can also be grouped in a stack by `CGrid`. Any two dockables with the same boundaries are grouped. The `add` method uses a vararg-argument, more than just one dockable can be placed with the same boundaries this way.

Internally `CGrid` uses a `SplitDockGrid`. `SplitDockGrid` contains an algorithm that creates a `SplitDockTree`. This tree has dockables as leafs and relations between dockables are modeled as nodes. A `SplitDockTree` can be used by a `SplitDockStation` to build up its layout.

5.3 For all dockables: layout

The "layout" is the set of all locations, even including invisible dockables. `CControl` supports the storage and replacement of layouts automatically. Clients only need to provide some factories for their custom dockables. A layout does not have direct references to any dockable, it is completely independent of gui-components.

There are four important methods in `CControl` used to interact with layouts:

- `save` - stores the current layout. The method requires a `String` argument that is used as key for the layout. If a key is alread used then the old layout gets replaced with the new one.
- `load` - is the counterpart to `save`. It loads a layout that was stored earlier.
- `delete` - deletes a layout.
- `layouts` - returns all the keys that are in use for layouts.

The class `CLayoutChoiceMenuPiece` can build some `JMenuItems` that allow the user to save, load and delete layouts at any time. More about `MenuPieces` can be found in chapter 8.

Layouts are divided into two subsets: "entry" and "full" layouts. An entry-layout does not store the location of any dockable that is associated with a working-area. A full-layout stores all locations. The method `save` always uses entry-layouts and a full-layout is only used when the applications properties are stored persistantly in a file.
Working-areas are intended to show some documents that are only temporarely available. Assuming that each dockable on a working-area represents one such document it makes perfectly sense not to replace them just because the user chooses another layout. Changing them would mean to close some documents and load other documents, and that is certainly not the behaviour the user would expect.

The client is responsible to store the contents of any single-dockable.

5.3.1 Persistant Storage

`Common` uses a class called `ApplicationResourceManager` to store its properties. Among other things all layout information is stored in this resource-manager. Normally any information in the resource-manager gots lost once the application shuts down. But clients can tell the resource-manager to write its contents into

a file. Either they call `getResources` of `CControl` and then one of the many methods that start with "write" or they use directly `CControl`. An example:

```
1  File file = new File( "layout.data" );
2
3  // write properties
4  control.write( file );
5
6  // read properties
7  control.read( file );
```

5.3.2 Dealing with lazy creation and missing dockables

While `MultipleCDockable`s are created only when they are needed, `Common` assumes that `SingleCDockable`s are always present. However this assumption would require to create components that might never be shown. In order to solve the problem `SingleCDockableBackupFactory` was introduced. If a missing single-dockable is required the factories method `createBackup` is called. Assuming the factory returns not `null` then the new dockable is properly added to `CControl` and made visible.

`SingleCDockableBackupFactory`s need to be registered at the `CControl` using the method `addSingleBackupFactory`. They can also be removed using the method `removeSingleBackupFactory`.

If a dockable is removed from a `CControl` then normally all its associated location information is deleted. If however a backup-factory with the same id as the dockables id is registered, then the location information remains. If another dockable with the same id is later registered, then this new dockable inherits all settings from the old one.

`CControl`s behavior for missing dockables can be fine tuned with a `MissingCDockableStrategy`.

6 Actions

Actions are small graphical components associated with a dockable. They can show up a different locations, e.g. as buttons in the title. An action is an instance of `CAction`. `Common` provides several subclasses of `CAction`. `CAction`s can be added to any `DefaultCDockable` through the method `addAction`. An example:

```
1  DefaultCDockable dockable = ...
2  CAction action = new ...
3
4  dockable.addAction( action )
```

To separate a group actions from another group a separator is needed. The method `addSeparator` of `DefaultCDockable` adds such a separator. The separator will behave like a `CAction`.

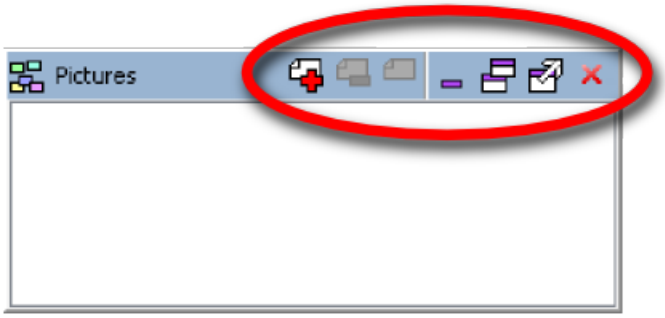

Figure 3: A set of actions on a dockable. The actions are the icons within the red oval.

An action is not a `Component`, it can appear at the same time at different locations with different views. For example an action can be seen as button in a title and at the same time as menu-item in a popup-menu.

6.1 CButton

`CButton`s are actions that can be triggered many times by the user and will always behave the same way. `CButton`s need to be subclassed, its abstract method `action` will be called whenever the button is triggered. An example:

```
1  public class SomeAction extends CButton{
2      public SomeAction(){
3          setText( "Something" );
4      }
5
6      protected void action(){
7          ...
8      }
9  }
```

6.2 CCheckBox

This action has a state, it is either selected or not selected (`true` or `false`). Whenever the user triggers the action the state changes. Like `CButton` is must be subclassed. The method `changed` will be called when the state changes. An example:

```
1  public class SomeAction extends CCheckBox{
2      public SomeAction(){
3          setText( "Something" );
4      }
5
6      protected void action(){
7          boolean selected = isSelected();
8          ...
9      }
10 }
```

6.3 CRadioButton

In most aspects the `CRadioButton` behaves like a `CCheckBox`. `CRadioButton`s are grouped together, the user can always select on of the buttons. A group of

is realized with the help of the class `CRadioGroup`:

```
1  CRadioButton buttonA = ...
2  CRadioButton buttonB = ...
3
4  CRadioGroup group = new CRadioGroup();
5
6  group.add( buttonA );
7  group.add( buttonB );
```

6.4 CMenu

A `CMenu` is a list of `CActions`. The user can open the `CMenu` and it will show a popup-menu with its actions. Clients can add and remove actions from a `CMenu` through methods like `add`, `insert`, or `remove`.

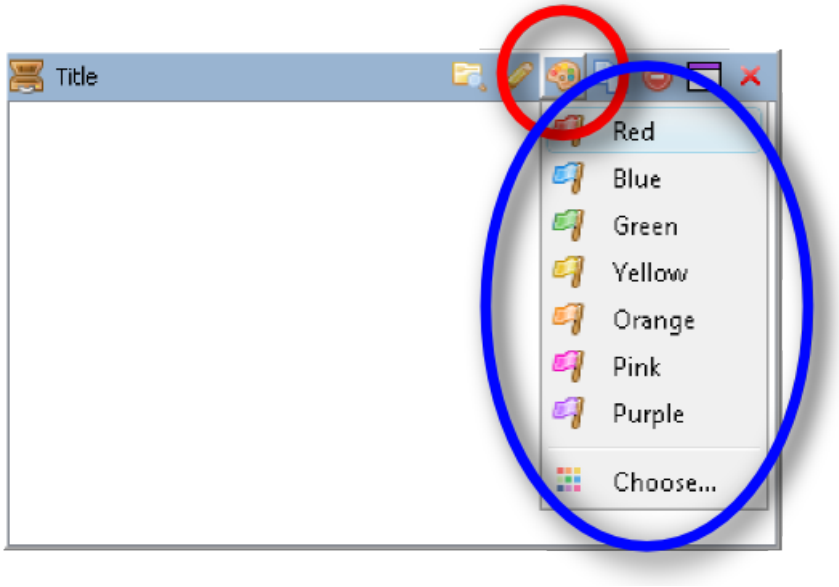

Figure 4: An open `CMenu`. The action itself is at the top within the red circle. Its menu consists of `CButtons` and a separator, the menu is within the blue oval.

6.5 CDropDownButton

A `CDropDownButton` consists of two buttons. One of them opens a menu, the other one triggers the last selected item of that menu again.

The behavior of `CDropDownButton` can be influenced through its items. This requires that the items are subclasses of `CDropDownItem`. `CButton`, `CCheckBox` and `CRadioButton` fulfill this requirement. There are three properties to set:

- `dropDownSelectable` - whether the action can be selected at all. If not, then clicking onto the item might trigger it, but the drop-down-buttons icon and text will remain unchanged.

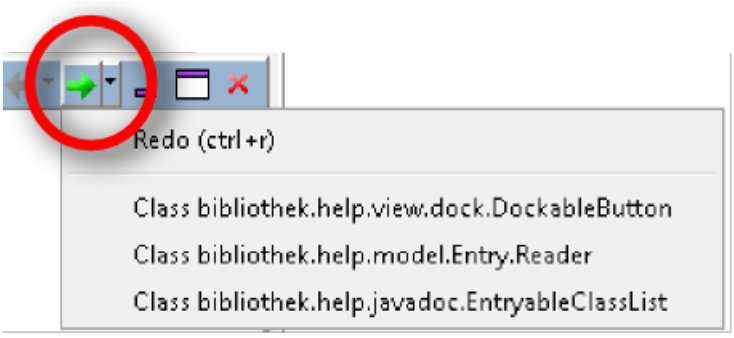

Figure 5: A `CDropDownButton` within a red circle.

- `dropDownTriggerableNotSelected` - if not set, then this item cannot be triggered if not selected. As a consequence the item must be clicked twice until it reacts.
- `dropDownTriggerableSelected` - if not set, then this item cannot be triggered if selected. It still can be triggered by opening the menu and then clicking onto the item.

If a `CDropDownButton` cannot trigger its selected item, then it just opens its menu.

6.6 CBlank

This action is not visible and does nothing. It can be used as placeholder where a `null` reference would cause problems, e.g. because `null` is sometimes replaced by some default value.

6.7 System Actions

`Common` adds a number of actions to any `CDockable`, e.g.: the close-button. These actions are deeply hidden within the system and cannot be accessed. There is however a mechanism to replace them with custom actions. Each `CDockable` has a method `getAction` which is called before a system action is put in place. If this method does return anything else than `null` then the system action gets replaced. `AbstractCDockable` offers the method `putAction` to set these replacements. An example:

```
1  SingleCDockable dockable = ...
2  CAction replacement = ...
3
4  dockable.putAction( CDockable.ACTION_KEY_MAXIMIZE, replacement );
```

In this example whenever the maximize-action of `dockable` should be visible, `replacement` is shown. This feature should of course be treated with respect, changing the behavior of an action can confuse the user a lot.

The class `CCloseAction` is an action that closes any dockable on which it is shown. The subclasses of `CExtendedModeAction` change the extended-mode of their dockables.

6.8 Custom Actions

Clients are free to write their custom actions. They need to implement a new `DockAction` and a subclass of `CAction`. The subclass can give its super-class an instance of the custom `DockAction` or call `init` to set the action. Please refere to the guide for `Core` to find out how to implement a `DockAction`.

7 Customizing: small but impressive

`Common` allows to customize some behavior and components. Understanding these features is not necessary to work with `Common`, but impressive effects can be built with them. This chapter will, without any specific order, introduce some of these features.

7.1 Color

Every dockable has a `ColorMap`. This map contains colors that are used in the graphical user interface. Normally the map is empty and some default colors are used. If a client puts some colors into the `ColorMap`, then the user interface is immediatelly updated using the new colors. `ColorMap` itself contains a set of keys that can be used, as an example:

```
1  CDockable dockable = ...
2  ColorMap map = dockable.getColors();
3  map.setColor( ColorMap.COLOR_KEY_TAB_BACKGROUND, Color.RED );
```

Some keys are specialications of other keys. For example `COLOR_KEY_TAB_BACKGROUND` changes the background of tabs, while `COLOR_KEY_TAB_BACKGROUND_FOCUSED` changes the background of focused tabs only. A specialized key overrides the value provided by a general key.

Colors require the support of a `DockTheme` that applies them. Only themes of `Common` do that, the original themes of `Core` will render the `ColorMap` useless. In `Common` clients should interact with themes only through the `ThemeMap`, this map will make sure that only themes are used that support colors.
Also note that some `Components`, like the `JTabbedPane`, and some `LookAndFeel`s do not support custom colors.

7.2 Font

Exactly like the color, fonts of dockables can be exchanged. Each dockable has a `FontMap` which contains `FontModifiers`. `FontModifiers` can change some property of a font, an example:

```
1  CDockable dockable;
2  FontMap fonts = dockable.getFonts();
3
4  GenericFontModifier italic = new GenericFontModifier();
5  italic.setItalic( GenericFontModifier.Modify.ON );
6  fonts.setFont( FontMap.FONT_KEY_TAB, italic );
```

The `FontModifier` `italic` will change the italic flag of the original font to `true` (line `5`).

Some `Components`, like the `JTabbedPane`, and some `LookAndFeels` do not support custom fonts. In this case the settings are just ignored.

7.3 Size

Every dockable has a width and a height. Some dockables are flexibel in their size, others would be better of with a constant size. There is a feature to lock the size and a feature to set a specific size.

7.3.1 Lock the size

Every `AbstractCDockable` has the method `setResizeLocked`. If the size is locked through this method than any station will try not to change the size of the dockable. There are also methods to lock only width or height (`setResizeLockedHorizontally` and `setResizeLockedVertically`).

Locking the size does not prevent the user from manually resizing the dockable. And sometimes a station needs to violate the locking as well, e.g.: when a grid-area has only one child the size cannot be choosen freely.

7.3.2 Request a size

7.4 Maximizing

7.5 Preferences

7.6 Menus