DockingFrames 1.0.8 - Common

Benjamin Sigg

August 8, 2010

Contents

1 Introduction **4**

2 Notation **5**

3 Basics **6**

3.1 Concepts 6
3.2 Hello World 6
3.2.1 Setup controller 7
3.2.2 Setup stations 7
3.2.3 Setup dockables 8

4 Foundation **10**

4.1 Dockables 10
4.1.1 SingleCDockable 11
4.1.2 MultipleCDockable 11
4.1.3 Visibility 12
4.1.4 Mode 13
4.2 Stations 13
4.2.1 All in one: CContentArea 14
4.2.2 Center area: CGridArea 15
4.2.3 Minimized: CMinimizeArea 15
4.2.4 Grouping Dockables: CWorkingArea 15

5 Locations **17**

5.1 For a single dockable: CLocation 17
5.2 For a group of dockables: CGrid 17
5.3 For all dockables: layout 19
5.3.1 Persistant Storage 19
5.3.2 Dealing with lazy creation and missing dockables 20

6 Actions **21**

6.1 CButton 21
6.2 CCheckBox 21
6.3 CRadioButton 22
6.4 CMenu 22
6.5 CDropDownButton 22
6.6 CPanelPopup 23
6.7 CBlank 23
6.8 System Actions 23
6.9 Custom Actions 24

7 Other Effects **25**

7.1 Color 25
7.2 Font 25
7.3 Size 26
7.3.1 Lock the size 26
7.3.2 Request a size 26
7.4 Maximizing 27
7.5 Preferences 27

7.6 Themes 28
7.7 LookAndFeel 29
7.8 Menus 29
7.8.1 Themes 31
7.8.2 LookAndFeel 31
7.8.3 Layout 31
7.8.4 List of Dockables 31
7.8.5 Preferences 32

8 Suggestions **33**
8.1 Of people using the library 33
8.2 Of the developers 33

A Properties **35**

1 Introduction

DockingFrames is an open source Java Swing framework. This framework allows to write applications with floating panels: Components 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, it is a wrapper around Core and requires Core to work.

This guide does not claim to be complete nor that all of its parts are relevant. It is intended as a starting point to explain basic concepts and to find out which classes, interfaces and properties are important for developers. This document only covers Common, Core has its own guide.

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

While `Common` is a layer atop of `Core`, `Common` itself consists of three more layers: `common`, `facile` and `support` (in their respective packages). The `facile` layer mostly contains stand-alone abstractions of classes/interfaces of `Core`, the `common` layer brings these abstractions together. The `support` layer contains exactly what it's name suggest: small, generic classes and methods that do not fit anywhere but that are really helpful in building up the other layers.

Clients almost exclusively have to make use of the `common` layer. They can use the other layers, but it seldomly makes sense to do so.

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 showing some part of the data. E.g. the panels of a web-browser could be the "history", the "bookmarks" and the open websites.

`Common` adds an additional layer between panels and main-frame, it separates them and allows the user to drag & drop panels. For this to happen the client needs to wrap each panel into a `CDockable`. These `CDockable`s are put onto a set of `CStation`s, a controller (of type `CControl`) manages the look, position, behavior etc. of all these elements.

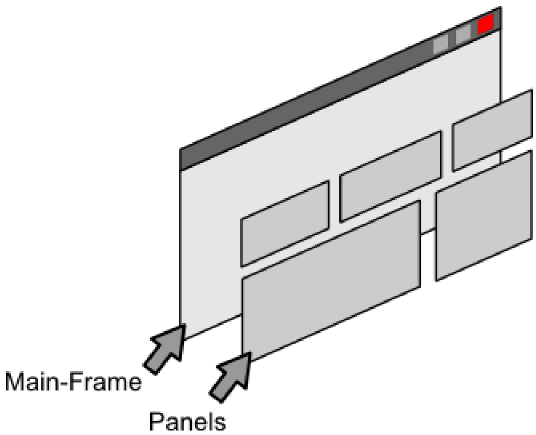

Figure 1: The standard application without `Common`. A main-frame and some panels that are put onto the main-frame.

3.2 Hello World

A first example containing only three colored panels will introduce the very basic vocabulary. In depth discussions of the concepts and implementations follow in the chapters afterwards.

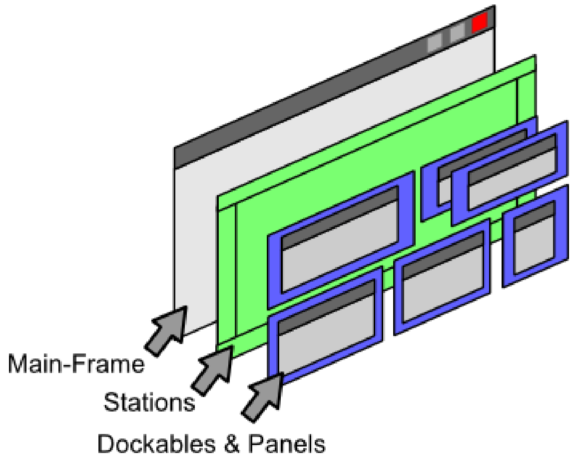

Figure 2: An application with `Common`. The panels are wrapped into dockables. The dockables are put onto stations which lay on the main-frame. Dockables can be moved to different stations.

3.2.1 Setup controller

The first step should be to create a `CControl`. This central controller wires all the objects of the framework together. A `CControl` needs to know the root window of the application, it is used as parent for any dialog that may be opened (e.g. during a drag & drop operation a dialog may be used to paint the dragged element). Most applications will be able to just forward their root window to one of the constructors.

The code to create the controller looks like this:

```
public class Example{
    public static void main( String[] args ){
        JFrame frame = new JFrame();
        frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

        CControl control = new CControl( frame );

        ...
```

3.2.2 Setup stations

The second step is to setup the layer between main-frame and dockables. There are different `CStations` available, for example the `CMinimizeArea` shows minimized `CDockables`. But most applications will always use the same layout: some station in the center of the frame shows a grid of `CDockables` and on the four edges minimized `CDockables` are listed. The class `CContentArea` is a combination of several `CStations` offers exactly that layout.

There is always a default-`CContentArea` available, it can be accessed by calling `getContentArea` of `CControl`. If required additional `CContentAreas` 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 ){
```

```
        JFrame frame = new JFrame();

        frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

        CControl control = new CControl( frame );

        frame.setLayout( new GridLayout( 1, 1 ) );
        frame.add( control.getContentArea() );

        ...
```

> `CControl` always creates an additional station for handling free floating `CDockable`s.

3.2.3 Setup dockables

The last step is to set up some `CDockable`s. `CDockable`s are the things 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` is just an interface and clients should always use one of the two subclasses `DefaultSingleCDockable` or `DefaultMultipleCDockable`. Without going into details: `single-dockable`s exist exactly once, while `multi-dockable`s can be created and destroyed by the framework anytime.

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 location can be set like in line `33` and `36`. The initial location is applied in the moment when the `dockable` gets visible, it will not have any influence afterwards. So there is no point in setting the location of the first `dockable`, since there are no other `dockable`s it gets all the space anyway and the initial location does not matter afterwards. With other words: the order in which `dockable`s are made visible is important.

> There is a class `CGrid` which allows to build an initial layout more easily, more about locations can be found in chapter 5

```
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 );
```

```java

        frame.setLayout( new GridLayout( 1, 1 ) );
        frame.add( control.getContentArea() );

        SingleCDockable red = create( "Red", Color.RED );
        SingleCDockable green = create( "Green", Color.GREEN );
        SingleCDockable blue = create( "Blue", Color.BLUE );

        control.add( red );
        control.add( green );
        control.add( blue );

        red.setVisible( true );

        green.setLocation( CLocation.base().normalSouth( 0.4 ));
        green.setVisible( true );

        blue.setLocation( CLocation.base().normalEast( 0.3 ) );
        blue.setVisible( true );

        frame.setBounds( 20, 20, 400, 400 );
        frame.setVisible( true );
    }

    public static SingleCDockable create( String title, Color color ){
        JPanel background = new JPanel();
        background.setOpaque( true );
        background.setBackground( color );

        return new DefaultSingleCDockable( title, title, background );
    }
}
```

4 Foundation

This chapter focuses on the foundation of `Common`: `CControl`, the `station`s and `dockable`s.

4.1 Dockables

A `CDockable`s is not much more than a set of properties like a “title-icon” and some `Component` which is to be shown to the user. A `CDockable` does not paint any decorations (like a title), handle any `MouseEvent`s or interacts with the user in any other way. Such jobs are handled by various manager-objects, factories and delegates. These objects need to know the `CDockable`s they work with and they need to be able to store information about them. To accomplish that each `CDockable` must be registered at a `CControl` and is associated with a unique identifier. Registering a `CDockable` is possible with one of the `add`-methods `CControl` offers.

The unique identifier of a `CDockable` is either chosen by the client, in which case we speak of a `single-dockable`, or by the framework itself, in that case we speak of a `multi-dockable`. The user won’t notice a difference for these two types of `dockable`, but internally they are handled quite differently. The framework assumes that the client knows of all the `single-dockable`s and that the client registers either factories or `dockable`s when the application starts up. On the other hand the framework assumes that the client does not know how many `multi-dockable`s exist and what their content is. The client has to provide a factory for different types of `multi-dockable`s and the framework will create new `dockable`s when necessary.

When a `CDockable` is created it is without location and it is invisible. Read chapter `sec:location` to learn about locations and call `CDockable.setVisible` to show the `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`. Subclasses of `DefaultCDockable` can override `createCommonDockable` to use their own implementation of `Dockable`.

4.1.1 SingleCDockable

A `single-dockable` is created once, added to the controller and made visible. It remains in memory until explicitly removed from the `CControl` or the application terminates. Alternatively a `SingleCDockableBackupFactory` can be used to lazily create a `dockable` once it is required.

The interface `SingleCDockable` represents a `single-dockable`, the class `DefaultSingleCDockable` is a convenient implementation of the interface. The class offers methods similar to those known from a `JFrame`, e.g. it has a content-pane where clients can add some `Components`.

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

A set of `multi-dockables` is used if the exact number of instances is not known prior to runtime. Before a `multi-dockable` can be used its factory (of type `MultipleCDockableFactory`) must be registered at a `CControl`. The factory offers methods to store and load the layout of a `dockable`. Of course there can be more than one `multi-dockable` per factory. Once the factory is registered, clients can add and remove `multi-dockables` at any time.

The interface describing all `multi-dockables` is `MultipleCDockable`, a convenient implementation is `DefaultMultipleCDockable`. This class offers most of the methods and properties a client should ever need. Implementing a matching `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.

If a `multi-dockable` is made invisible, it should be removed from its `CControl`. Otherwise old objects fill up memory until the application crashes with an `OutOfMemoryException`. Clients either need to remove the `dockable` by themselves or they can call `setRemoveOnClose(true)` to let the element be removed automatically once it becomes invisible

> Any `multi-dockable` which is no longer required must be removed from the `CControl`, otherwise an `OutOfMemoryException` may happen.

Automatical removal is triggered when the `dockable` becomes invisible. This happens if the element does no longer have a root-parent. If a client makes the parent of a `multi-dockable` invisible, the `dockable` gets marked invisible as well. However, it does not get removed from its parent. Strange effect happen when the parent is made visible again: most of the decoration of the `dockable` is gone and some parts of the framework will ignore its existence.

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.

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.

Why the distinction between `single-` and `multi-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 `multi-dockables` represent this gap.

4.1.3 Visibility

A `dockable` is either visible or invisible, the user can only interact with the `dockable` if it is visible. The easiest way to change the visibility of a `dockable` is to call `setVisible` of `CDockable`. The framework makes sure that a `dockable` keeps its location if it is made invisible but stays registered at the `CControl`.

Visibility may be changed by the user: by clicking the close-button of a `dockable`. The default behavior of the close-button is just to call `setVisible`. Every `DefaultCDockable` offers the method `setCloseable` to change whether the user can click away the element. Visibility may also be changed implicitly when the visibility-state of the parent of a `dockable` changes. Also using a `CGrid` (see chapter 5) or other methods to influence the layout of an application may change the visibility.

A client may need to perform some actions when a `dockable` is made invisible or a client may want to forbid the user to click away a `dockable`. The `CDockableStateListener` and the `CVetoClosingListener` are meant to do this job. The CDockableStateListener just gets informed after visibility changed. The CVetoClosingListener gets informed before and after visibility changes, this listener can cancel the operation if necessary. Both listener can either be added directly to a dockable or to the CControl. In the first case the listener gets informed about events that are related only the one dockable, otherwise it gets informed about any event.

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. Read 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. Read more about locations in chapter 5.

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 CDockables. 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`.

`CStations` 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 `DockStations` 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`:

```
JFrame frame = ...
CControl control = ...

CContentArea area = control.getContentArea();
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 created 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 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, nor 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. An example: `CGridAreaLocation` offers the method `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 `b` from its position. As an example:

```
CDockable a = ...
CDockable b = ...

CLocation location = b.getBaseLocation();
a.setLocation( location );
```

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

```
a.setLocation( location.aside() );
```

`CLocation` is a wrapper around `DockableProperty`. While each `DockableProperty` has its own API and concepts, `CLocations` 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 `CGrid` 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 7.8.

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 gets 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:

```java
File file = new File( "layout.data" );

// write properties
control.write( file );

// read properties
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 at different locations, e.g. as buttons in the title. An action is an

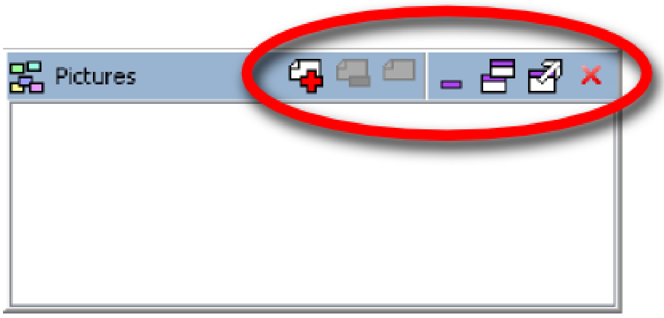

Figure 3: A set of actions on a dockable. The actions are the icons within the red oval.

instance of `CAction`. `Common` provides several subclasses of `CAction`. `CActions` can be added to any `DefaultCDockable` through the method `addAction`. An example:

```
DefaultCDockable dockable = ...
CAction action = new ...

dockable.addAction( action )
```

To separate a group actions from another group a separator is needed. The method `addSeparator` of `DefaultCDockable` adds such a separator. Separators are specialized `CActions`.

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

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

```
public class SomeAction extends CButton{
    public SomeAction(){
        setText( "Something" );
    }

    protected void action(){
        ...
    }
}
```

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:

```
public class SomeAction extends CCheckBox{
    public SomeAction(){
        setText( "Something" );
    }

    protected void action(){
        boolean selected = isSelected();
        ...
    }
}
```

6.3 CRadioButton

In most aspects the `CRadioButton` behaves like a `CCheckBox`. `CRadioButtons` are grouped together, the user can select only one of the buttons in a group. A group is realized with the help of the class `CRadioGroup`:

```
CRadioButton buttonA = ...
CRadioButton buttonB = ...

CRadioGroup group = new CRadioGroup();

group.add( buttonA );
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`.

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.
- `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.

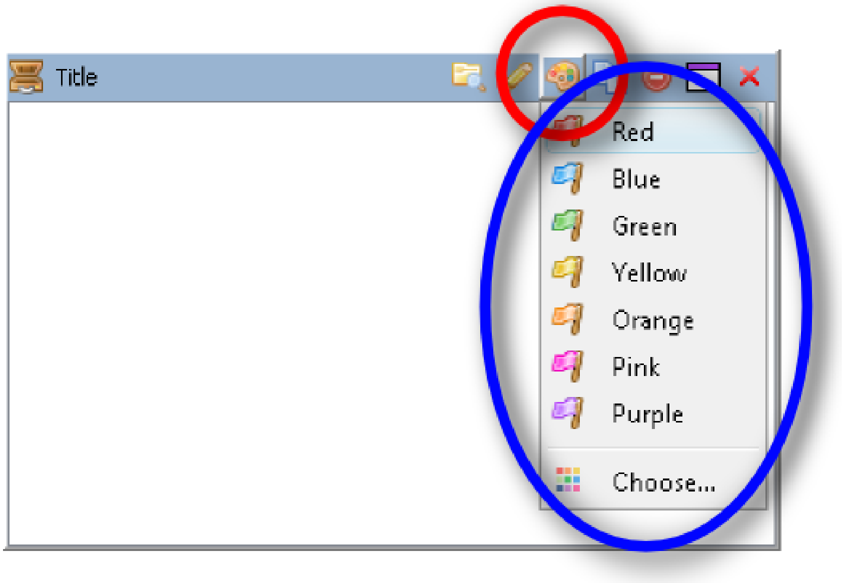

Figure 4: An open `CMenu`. The action itself is at the top within the red circle. Its menu consists of `CButtons` and a separator, the menu is within the blue oval.

6.6 CPanelPopup

Basically a button that opens a popup with an arbitrary component as content. The popup appears at the same location the menu of a `CMenu` would appear. In a menu a `CPanelPopup` appears as menu-item and opens the popup in the middle of the `CDockable` to which it is attached. The class provides methods for clients to modify its behavior, e.g. to replace the popup by another implementation.

6.7 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.

6.8 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:

```
SingleCDockable dockable = ...
CAction replacement = ...

dockable.putAction( CDockable.ACTION_KEY_MAXIMIZE, replacement );
```

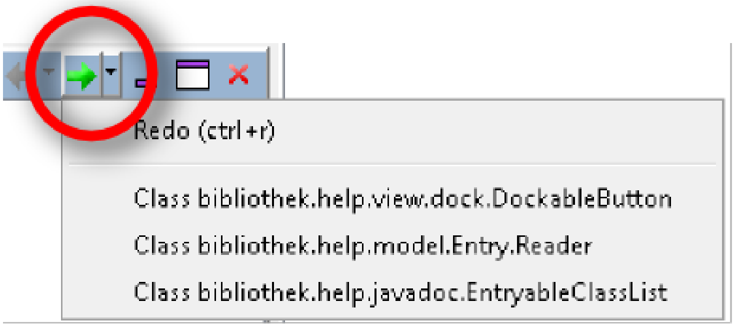

Figure 5: A `CDropDownButton` within a red circle.

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.

6.9 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 Other Effects

`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:

```
CDockable dockable = ...
ColorMap map = dockable.getColors();
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 `LookAndFeels` 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

It is also possible for client code to request a specific size for one or many `CDockables`. Clients need to call `setResizeRequest` and maybe `handleResizeRequest` like in the example below:

```
1  CControl control = ...
2
3  DefaultCDockable a = ...
4  DefaultCDockable b = ...
5
6  a.setResizeRequest( new Dimension( 200, 300 ), false );
7  b.setResizeRequest( new RequestDimension( 500, true ), false );
8
9  control.handleResizeRequests();
```

In this example two resize requests are handled at the same time. In line `6` the resize request of `a` is set to $200, 300$, the argument `false` tells `a` not yet to process the request. In line `7` the resize request of `b` is set, `b` should have the width 500 but should not care about its height. Finally in line `9` all the requests are processed together. If the second parameter in line `7` would be `true` instead of `false`, then line `9` would not be necessary.

Not processing a request directly, but collect them, allows requests to interact with each other. Assume three dockables in a line and the task to resize the two elements at the begin and end of the line. If one resize request is handled before the other, than the second request might destroy the work of the first one.

Every object can add a `ResizeRequestListener` to `CControl`, this listener will be called when resize requests need to be processed. Most of the `CStations` add such a listener. The only station on which requests can have complex interactions is the `CGridArea` (and the `CContentArea`). With the `PropertyKey` `RESIZE_LOCK_CONFLICT_RESOLVER`, defined in `CControl`, clients can set the algorithm that is used to solve contradictions in a `CGridArea`.

7.4 Maximizing

When maximizing a `CDockable` first the global `CMaximizeBehavior` is asked for the real element to maximize. If for example a dockable is part of a whole stack of dockables, then the maximize-behavior might decide to maximize the whole stack instead of the single dockable. In a second step the tree of stations and dockables is traversed upwards until a `MaximizeArea` is found, the element is then shown on this area. If no `MaximizeArea` can be found in the ancestors, then a default area is taken.

Clients are not (yet) able to influence the second step, but they can change the `CMaximizeBehavior`:

```
1 CControl control = ...
2 CMaximizeBehaviour behaviour = ...
3
4 control.getStateManager().setMaximizeBehavior( behavior );
```

In line 2 a custom behavior is declared, in line 4 the behavior is set.

Most the things that need to be done for changing the extended mode (like the “maximized mode”) are handled by the `CStateManager`.

7.5 Preferences

Common allows users to set some properties like the keys that need to be pressed in order to maximize a dockable (`ctrl+m`). Normally this mechanism is deactivated and clients first need to activate it:

```
1 CControl control = ...
2 PreferenceModel preferences = new CPreferenceModel( control );
3
4 control.setPreferenceModel( preferences );
```

This piece of code activates the preference mechanism. In line 2 the set of preferences that can be changed by the user is set up, a `CPreferenceModel` is often the best choice. Then in line 4 the model is connected to `control`. Calling `setPreferenceModel` will activate persistant storage for `model` and also immediatelly load values into the model.

The model can later be presented to the user:

```
CControl control = ...
PreferenceModel model = control.getPreferenceModel();
Component owner = control.intern().getController().findRootWindow();
```

```
5  if( model instanceof PreferenceTreeModel ){
6      PreferenceTreeModel tree = (PreferenceTreeModel)model;
7      PreferenceTreeDialog.openDialog( tree, owner );
8  }
9  else{
10     PreferenceDialog.openDialog( model, owner );
11 }
```

In line 3 the root window of the application is searched, it is used as parent window for any dialog that needs to be opened. In line 7 or line 10 a dialog is opened that shows the preferences. There are two different dialogs, one with a tree at the left side to make select a subset of preferences, one without tree.

There are different preference models. `CPreferenceModel` contains all possible preferences for Common, it consists of four other models:

- `CKeyStrokePreferenceModel`: The different key combinations that, when pressed, initiate some action.
- `CLayoutPreferenceModel`: General settings for the themes.
- `BubbleThemePreferenceModel`: Settings affecting the eclipse-theme.
- `EclipseThemePreferenceModel`: Settigns affecting the bubble-theme.

Internally each item of the model is a `Preference`, clients can put together their own model.

The class `CPreferenceMenuPiece` can act as a menu-item for opening the preference-dialog, read more about menus in chapter 7.8.

7.6 Themes

A theme sets look and behavior of `DockingFrames`. Themes are managed by the `ThemeMap`, this map contains `String`s as keys and `ThemeFactory`s as values. `ThemeMap` is however more than just a map, it also tells which theme is currently selected. Clients can call `select` to change the selection.

In the current version 5 themes are always installed per default, the keys of these 5 themes are stored as constants directly in `ThemeMap`.

Working with the `ThemeMap` could look like this:

```
1 CControl control = ...
2 ThemeMap themes = control.getThemes();
3
4 themes.select( ThemeMap.KEY_FLAT_THEME );
5
6 themes.add( "custom", new CustomFactory() );
```

In line 2 the map is accessed. In line 4 one of the preinstalled themes is selected, this theme is applied to `control`. In line 6 a factory for a custom theme is installed.

A theme has much freedom in how to present the dockables. But `Common` allows clients to set color and font of various elements associated with a `CDockable`. The standard themes of `Core` would not respect these settings, hence `Common` needs some modified themes. The `ThemeMap` is an attempt to hide this ugly fact from developers and to make sure they don't use the wrong theme.

7.7 LookAndFeel

`LookAndFeel` tells a `Swing` application how to paint things and how to behave. The relation between `LookAndFeel` and `Swing` is like the relation between theme and `DockingFrames`. The `LookAndFeel` can be changed while the application runs, but the method `updateUI` must be called for each and every existing `JComponent` by the client itself.

Of course, clients are free to implement such a function. `DockingFrames` will detect a change of the `LookAndFeel` and update itself where necessary, but it will not update the `JComponents`.

But `Common` includes better support for `LookAndFeel` changes. The class `LookAndFeelList` provides a list of all available `LookAndFeels` and allows to change the current selection. Per default the list does not exist but clients can easily create one:

```
1  LookAndFeelList list = LookAndFeelList.getDefaultList();
2
3  CControl control = ...
4
5  ComponentCollector collector =
6      new DockableCollector( control.intern() );
7  list.addComponentCollector( collector );
8
9  XElement xsettings = ...
10 list.readXML( xsettings );
```

In line `1` a `LookAndFeelList` is accessed, calling `getDefaultList` will create it. In order to automatically update `JComponents` they need to be connected to the list. This is done with the help of `ComponentCollectors`. If for example a `CControl` like `control` (line `3`) is given, then the class `DockableCollector` (lines `5-7`) is able to collect *all* components related to it. This includes all dockables but also the root-window of the application. The `LookAndFeelList` can store its state persistantly and later read the state, for example in line `9` some earlier setting is accessed and in line `10` the settings are applied.

If using a `CLookAndFeelMenuPiece` then everything in the example snippets gets done automatically. Read chapter 7.8.2 to learn more about this menu.

7.8 Menus

Most `Swing` applications use menus (like in figure 6). `DockingFrames` contains a few actions that fit nicely into a menu, for example store and load a layout.

For a given option the number of required menu-items may change during runtime, e.g. every stored layout requires one item. But developers may not want to add one `JMenu` for each option of `DF`. To resolve this problem `Common` introduces a very small framework that allows the management of dynamically growing or shrinking menus.

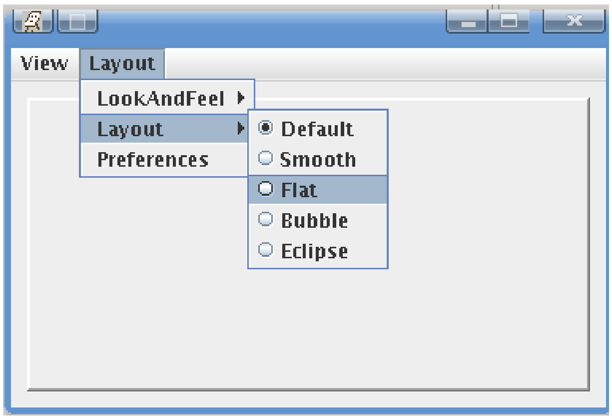

Figure 6: Some menus.

The most important class of the menu-framework is the `MenuPiece`. Basically a `MenuPiece` is a list of `Components` which informs observers if it changes its size. There are around 15 subclasses of `MenuPiece`, they allow to compose many pieces to one big piece or have more specific duties like providing the stored layouts.

An incomplete list of composing `MenuPiece`s contains:

`RootMenuPiece` : Represents a whole `JMenu`.

`SubMenuPiece` : A wrapper around a `RootMenuPiece` allowing it to act like a submenu.

`NodeMenuPiece` : Just a list of `MenuPiece`s that act like one big piece.

`SeparatingMenuPiece` : A wrapper around another `MenuPiece` introducing separators at the top and/or bottom.

Other `MenuPiece`s that might be interesting are:

`BaseMenuPiece` : A good base class for custom `MenuPiece`s, allows to add or remove `Components` directly.

`FreeMenuPiece` : A piece that does not add children by itself but has public methods which can be invoked by clients to modify the piece directly.

In the remainder of this section the more complex `MenuPiece`s are introduced.

7.8.1 Themes

`Common` has several themes built in, a theme tells how to paint certain components or how to react on certain events. The theme mechanism is described in more detail in chapter 7.6.

Clients can use a `CThemeMenuPiece` to quickly create a menu that changes the theme. The menu tracks any changes in the `ThemeMap` of the associated `CControl`.

> If a `CThemeMenuPiece` is no longer required, then clients should call its method `destroy`.

7.8.2 LookAndFeel

`Common` already supports `LookAndFeels`, more about this feature can be read in chapter 7.7. The `CLookAndFeelMenuPiece` adds a menu that lists all the available `LookAndFeels` and allows to exchange them.

> If a `CLookAndFeelMenuPiece` is no longer required, then clients should call its method `destroy`.

> Each `CLookAndFeelMenuPiece` will store the selection persistant, assuming that clients call `write` of `CControl` or of `ApplicationResourceManager`. If this behavior is not whished, then the `LookAndFeelMenuPiece` provides similar behavior but without the persistant storage.

7.8.3 Layout

The layout is the location of all dockables as described in chapter 5.3. The `CLayoutChoiceMenuPiece` offers users several actions to work with layouts:

Save : Saves the current layout. If the current layout has not yet a name then a dialog pops up so the user can enter a name.

Save As : Saves the current layout but always asks the user to enter a new name for the layout.

Load : Loads a previously saved layout, the current layout gets not stored.

Delete : Deletes a previously saved layout.

7.8.4 List of Dockables

All closeable `SincleCDockables` known to a `CControl` can be listed in a `SingleCDockableListMenuPiece`. With this menu the user can make the dockables visible or invisible. The menu will update its content automatically as dockables are added or removed from the `CControl`.

7.8.5 Preferences

`Common` supports preferences as described in chapter 7.5. The class `CPreferenceMenuPiece` adds a single item that opens a dialog with the preferences of a `CControl`.

Per default the preference system is disabled. Clients can acivate the preference system in two ways:

- Call `setPreferenceModel` of `CControl` with the preferences that should be editable.
- Call `setup` of `CPreferenceMenuPiece` to optain a new menu and set the default model (`CPreferenceModel`) in the same step.

8 Suggestions

Users and developers made a lot of good suggestions, this chapter is an incomplete list of them.

Some word of warning: this is an open source project, as such its developer(s) are not so much interested in selling the framework to as many people as possible, but on having fun writing something cool. Hence some things that people would like to have will never be implemented because the developers don't have fun doing this stuff.

8.1 Of people using the library

- **Question:** When showing tabs, would it be possible to show a drop-down menu when there is not enough space for all the tabs?
  Answer: This will be implemented and has high priority.
- **Question:** Tabs: would it be possible to show them on the left, right, bottom, top rotate etc...?
  Answer: Whilst it would be easy to just put them at another place, there needs more to be done. This feature requires to upgrade most of the painting code. In theory the `StackDockComponent` would already provide developers with the ability to use their very own tabs (at their own place), but not to reuse the existing tabs. More settings would be a nice improvement of the framework and will most certainly be implemented.
- **Question:** AWT, it needs better support (e.g. things should be painted over AWT panels as well).
  Answer: AWT and Swing don't work together. This framework is based on Swing, any attempt to support AWT will result in a lot of ugly hacks. Also given the fact that AWT isn't hardly used anymore (except for applications playing video or rendering 3D scenes) this feature has little to none chances of getting implemented.
- **Question:** Could the framework be made available for [insert your favorite tool here]? E.g. in a Maven repository or for the Netbeans GUI Builder.
  Answer: Making the framework available in/for any special tool immediately yields two new problems. First, as soon as one tool is supported people will ask for another tool, this will never end... Second, a library does no get better because it does support many other tools, it does get better because it has lesser bugs, more settings or features.
- **Question:** Assume an externalized `CDockable`, if it gets maximized, could it be maximized like a `JFrame`? It would will the entire screen instead of falling back to the nearest `CContentArea`.
  Answer: This is a good idea. It is not yet clear how to implement this, but it is among those things that will be done.

8.2 Of the developers

Since the framework has its own forum many questions have been asked, and most of them were answered as well. From these questions some observations can be made:

- Problems arise both in `Core` and in `Common`. The problems are however of different nature. In `Core` most problems concern small things, e.g. how to place the tabs. Most of these problems can be solved with small patches.

  The problems related to `Common` are a lot more serious. Often the answer is "`Common` is not able to do that". And even worse, there is often no small patch. In short: *`Common` has serious design flaws.* Especially `Common` lacks the ability to customize components.

  Hence most future work must be spent on `Common`.

- The features now available seem to be sufficient for most applications. The requests for things that are entirely missing has dropped to almost zero. There is no need for new features, there is need to improve existing features.

Putting the pieces together the areas that will make the framework better are most likely:

- The `StateManager`, this class is responsible for managing the "extended mode". The class has continually grown and has become a major hindrance for customization. Currently there is absolutely no abstraction in this class, it needs to redesigned from scratch. This class is almost as important as `DockController` or `CControl`, its redesign will affect a lot of other classes. The effect will be, that a) any station can have any function, or many functions at the same time (e.g. minimizing could be mapped to a custom component). And b) clients would be able to introduce their very own extended modes.

- `CControl` and other classes use a lot of anonymous classes. They need to be named and made public, and clients need to be able to exchange them by their own implementations. New factories, also factories with customizable properties, could help.

- Clients need more control over `CDockable`s, or better their representation as `Dockable`. One possibility would be a second series of `CDockable`s that extend directly `DefaultDockable`.

- There should also be more observers, clients should be able to register and react (or cancel) to almost all actions of the framework.

A Properties

Core allows clients to set a number of properties, Common adds a few more. All properties can be set or read by putProperty and getProperty of CControl. An example:

```
CControl control = ...

PropertyKey<KeyStroke> key = control.KEY_CLOSE;
KeyStroke value = KeyStroke.getKeyStroke( "shift X" );

control.putProperty( key, value );
```

The keys for all properties of Common are stored as constants in CControl. The complete list:

CControl.KEY_MAXIMIZE_CHANGE

Type KeyStroke

Default ctrl + m

Usage If pressed then the focused dockables changes between maximized and normal state.

KEY_GOTO_MAXIMIZED

Type KeyStroke

Default null

Usage If pressed then the focused dockable becomes maximized.

KEY_GOTO_NORMALIZED

Type KeyStroke

Default ctrl + n

Usage If pressed then the focused dockable becomes normalized.

KEY_GOTO_MINIMIZED

Type KeyStroke

Default null

Usage If pressed then the focused dockable becomes minimized.

KEY_GOTO_EXTERNALIZED

Type KeyStroke

Default ctrl + e

Usage If pressed then the focused dockable becomes externalized.

KEY_CLOSE

Type KeyStroke

Default ctrl + F4

Usage If pressed then the focused dockable is made invisible.

RESIZE_LOCK_CONFLICT_RESOLVER

Type `ConflictResolver<RequestDimension>`

Default an instance of `DefaultConflictResolver`

Usage Tells how to distribute space when two or more dockables have conflicting size requests. See also chapter 7.3.