DockingFrames 1.0.5 - Common

Benjamin Sigg

April 20, 2008

Contents

Abstract

The Common-project is a layer above DockingFrames. It allows to write applications using DF without the need to understand DF. Common does not add new features to DF, Common only combines existing code and reorganizes DF in a way that is easy to use.

1 Introduction

1.1 DockingFrames

What is DF? DF is an open source framework written in Java. It extends Java/Swing with the abbility of "dockable frames". Each frame contains some content, a `JTree` showing a directory or a `Table` showing the results of some game. Each frame has a title and some buttons (like a close-button). The user can move around these frames, the frames will align themself such that there is no unused space between them. DF provides many possibilities how frames can be combined to create a new layout.

2 Common basics

The Common project is divided in three packages. "Facile" contains some lonly classes which can be used to create minor effects. "Support" contains elements which are not even connected to DF, but still usefull.

The most interesting package however is "common". "Common" contains the layer that will enable you to rapidly write applications using DF. This first section deals with the basic features of Common.

2.1 Basic elements

In the understanding of Common, an ordinary application has one `JFrame`, which is called the "mainframe", showing the content of the application. The content is painted through some `JComponent`s called "panels". Each panel represents some view of the content, for example a texteditor might have one panel for each open document.

Common stands between mainframe and the panels, separating them, and allowing the user to move then panels around.

To do so, each panel gets wrapped into a `CDockable`, such a combination is simply called "dockable". Dockables are then put onto a `CContentArea` which is just a child of the mainframe.

Since there is a complex mechanism between the dockables, a control unit is needed. The control unit is provided by the `CControl`, or just the "controller".

2.2 Creating the controller

The first thing that needs to be done before using Common is to set up the controller. This is done by creating a new object of type `CControl`:

```
1 JFrame frame = new JFrame( "Main-Frame" );
2 boolean secure = false;
3 CControl control = new CControl( frame, secure );
```

Let's have a brief look at the code. Clearly in line 1 the mainframe of the application is created. The controller needs to know the mainframe, it is used as parent of Windows which are opened for example when dragging a dockable.

In line 2 it is specified that the application does run in an unsecure environment. An unsecure environment is normal for applications. An applet or a webstart-application would run in a secure environment. The controller needs to know that either it can use non-secure optimations (like globally observing all AWT-Events) or has to use inefficient workarounds.

Finally in line 3 the controller is created.

2.3 Between mainframe and dockable

After creating mainframe and controller, the layer between mainframe and the dockables has to be set up.

2.3.1 Content area

A CContentArea contains all the functionality an application normally needs. The controller grants access to a default-content-area through CControl.getContentArea(). You can use CControl.createContentArea(String) to create additional areas when needed.

```
1 JFrame frame = ...
2 CControl control = ...
3 frame.add( control.getContentArea() );
```

Note line 3, a content-area is just a JComponent and can be added anywhere.

2.3.2 Stations

Sometimes a CContentArea is too much or does not the right thing. In version 1.0.4 the CStation was introduced. A CStation is a thin wrapper around an ordinary DockStation. Clients can write their own CStations or use some of the predefined stations:

CMinimizeArea A station where minimized dockables are shown.

CGridArea A station where normalized and maximized dockables are shown.

An example setting up a layout where normalized dockables can be shown in the middle of a JFrame, and minimized dockables are shown at the bottom only:

```
1 JFrame frame = new JFrame( "Demo" );
2 CControl control = new CControl( frame );
3
4 CMinimizeArea miniArea = new CMinimizeArea( control, "
    mini" );
5 miniArea.setDirection( FlapDockStation.Direction.NORTH );
6 control.add( miniArea, true );
7
8 CGridArea gridArea = control.createGridArea( "grid" );
9
```

```
10  frame.add( gridArea.getComponent(), BorderLayout.CENTER )
        ;
11  frame.add( miniArea, BorderLayout.SOUTH );
```

Note how `miniArea`, defined in line `4`, has to be added to the control in line `6`. The argument `true` tells the control, that `miniArea` is a root station, which means that `miniArea` has no parent.

A second station called `gridArea` is created in line `8`. Here one of the factory methods of `CControl` is used. That method does all the wiring that is necessary, and factory methods should be preferred whenever possible.

2.4 Wrapping a panel

There are some thoughts needed to create a dockable (to wrap a panel into a `CDockable`). Each kind of panel can fall in one of two categories: the number of instances during the lifetime of an application remains the same, or the number changes.

Those kinds of panels whose number does not change, should be wrapped into `SingleCDockable`s, the others in `MultipleCDockable`s. `SingleCDockable` and `MultipleCDockable` are only interfaces, for most developers the implementations `DefaultSingleCDockable` and `DefaultMultipleCDockable` will suffice. Only developers interested in writing elements which are parents of dockables need to write new classes and implement the interfaces.

Once a dockable is created, it has to be registered at the controller through the method `CControl.add`. Afterwards it can be made visible using `CDockable.setVisible`. Unless otherwise instructed, the controller will then open the dockable at a default location.

2.4.1 Single dockables

A single dockable is an object of type `SingleCDockable`. These dockables are created once by the application, added to the controller and made visible. Then they remain in the memory until they are explicitly removed from the controller, or the application terminates.

Every single dockable needs a unique identifier. This identifier allows the controller to store information about a dockable persistently and later to find the information again.

An example for a single dockable would be the list of documents that are currently open in a text editor.

```
1  CControl control = ...
2  SingleCDockable documents =
3          new DocumentList( "myapp-document-id" );
4  control.add( documents );
5  documents.setVisible( true );
```

2.4.2 Multiple dockables

A multiple dockable is an object of type `MultipleCDockable`. These dockables are created by the application or the controller, shown for some time and then

removed from memory. Unlike a single dockable, a multiple dockable can be created or deleted at any time.

Every multiple dockable needs a `MultipleCDockableFactory`, which must have been registered at the controller.

Example: the documents of a text editor

```
1  CControl control = ...
2  MultipleCDockableFactory factory = new
       DocumentDockableFactory();
3  control.add( "myapp-document-factory-id", factory );
4
5  MultipleCDockable document = new DocumentDockable(
       factory, "/home/beni/Desktop/file.txt" );
6  control.add( document );
7  document.setVisible( true );
```

2.5 Grouping dockables

Every time the user loads a previously stored layout, the multiple dockables will be deleted and new instances created. That can be very annoying and disturbing for user and developer. "working-areas" are designed to prevent such a behavior. Every `CStation` whose method `isWorkingArea` returns `true` is considered to be a working area. Most clients will use the class `CWorkingArea` and so this chapter will from now on use working-area and `CWorkingArea` as synonym.

`CWorkingArea`s can be fetched from the controller using `CController.createWorkingArea`. After their creation they are handled like every other single dockable, except that the `createWorkingArea`-method already added them to the controller.

Every dockable which should sit on a `CWorkingArea` needs to be put there. The most convenient way is to use `CDockable.setWorkingArea`. The controller will store this property when storing the layout. Have a look at the example:

```
1  CControl control = ...
2  CWorkingArea area = control.createWorkingArea( "area" );
3  SingleCDockable dockable = new ...
4
5  area.setSuppressTitle( false );
6  area.setTitleText( "Some title" );
7
8  area.setVisible( true );
9
10 control.add( dockable );
11 dockable.setWorkingArea( area );
12 dockable.setVisible( true );
```

An analysis: in line `2` a new `CWorkingArea` is created. Then in lines `5,6` the decorations of `area` are activated and the title set. In line `8` `area` is made visible. In line `10` a new dockable is added to the structure, and its preferred parent is set in line `11`. Afterwards `dockable` is made visible as well.

2.6 Changing the mode of a dockable

Every dockable needs to be in one mode: minimized, normalized, maximized or externalized. The user can change the mode of a dockable by clicking some buttons or moving the dockable around. Clients can read or change this mode as well, they just call `CDockable.getExtendedMode()` or `CDockable.setExtendedMode`.

The client can also specifiy, what modes are available for a dockable. When using the default-`CDockable`s, then `setMinimizable`, `setMaximizable` and `setExternalizable` can be used.

2.7 Close a dockable

When a dockable is no longer of use, it needs to be removed. There are several ways to remove a dockable and they have different effects.

With `CDockable.setVisible` the visibility of a dockable can just be changed to `false`. Single dockables will remain in the structure and can be reopened at a later time. `MultipleCDockable`s will be removed from the controller unless the `removeOnClose` property was set to `false`. When using a `DefaultMultipleCDockable` then this property can be changed through `setRemoveOnClose`.

With `CControl.remove` the dockable is not only made invisible, it is also removed from the controller. All properties related to the dockable will be lost. Unless added to a controller again, it is no longer possible to reopen the dockable.

Or give the user the possibility to close the dockable: setting the property `closeable` of the default-`CDockable`s to `true` will add a close-button to the title. A client can add a `CDockableStateListener` to the dockable in order to get informed when the visibility state changes. Clicking the close-button has the same effects as calling `setVisible(false)`.

After some time, the need for any elements of the framework might vanish. Using `CControl.destroy()` will release as many resources as possible.

3 Common features

There are more advanced features in Common. This section will introduce you to some methods allowing the fine-tuning of Commons.

3.1 Actions

Most dockables have some actions associated, for example a dockable showing some image might have the actions "zoom in" and "zoom out". Common provides a way to write such actions and attach their graphical representation (for example a button) to the titles of dockables.

Actions are modeled by `CAction` and various subinterfaces. All default-`CDockable`s have methods to add or remove these actions.

In the source, that might look like this:

```
1 DefaultSingleCDockable image = ...
2 CAction zoom = new ZoomAction();
```

```
3  image.addAction( zoom );
```

3.1.1 CButton

A `CButton` is just clicked by the user, and then executes some action. It is almost the same as a `JButton`. There is only one way to write a new `CButton`-action:

```
1  public class MyActions extends CButton{
2          public MyAction(){
3                  setText( "Action" );
4          }
5
6          public void action(){
7                  // do something
8          }
9  }
```

3.1.2 CCheckBox

The `CCheckBox` is an action with two states: selected or not selected. The state changes every time the user triggers the action. It is almost a `JCheckBox`. Writing a new `CCheckBox` is similar to writing a new `CButton`, except that the method `changed()` instead of `action()` has to be overriden.

3.1.3 CRadioButton

A `CRadioButton` is the same as a `CCheckBox` with one big difference: several `CRadioButton`s can be grouped together, and only one button of the group can be selected. The currently selected button gets unselected whenever the user triggers another button. Grouping is done with the help of a `CRadioGroup`:

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

3.1.4 CMenu

A `CMenu` is a list of actions. When the user triggers the `CMenu`, a popup-menu will appear and show the contents of the `CMenu`.

3.1.5 CDropDownButton

Something similar to the `CMenu`, but more advanced. The `CDropDownButton` serves two purposes. It is a menu like `CMenu`, and it remembers which action was triggered last. Triggering the button will trigger that last action again.

An example:

```
1  CDropDownButton dropDown = new CDropDownButton();
2
3  CButton buttonA = ...
4  CButton buttonB = ...
5
6  dropDown.add( buttonA );
7  dropDown.add( buttonB );
8
9  dropDown.setSelection( buttonA );
10
11 buttonB.setDropDownSelectable( false );
```

Let's analyze this code. In lines `1-4` some actions are created. In lines `6,7` the two `CButton`-actions are inserted into the menu of `dropDown`. Then in line `9` one action is marked as the last triggered action. `dropDown` will now show icon and text from `buttonA`. Something interesting happens in line `11`: the call to `setDropDownSelectable` will make sure that even when `buttonB` is triggered, it will not be marked as the last triggered action. Sometimes one wants to use `CDropDownButton` just as an ordinary drop-down-menu without more functionality.

3.1.6 CBlank

The action `CBlank` is never visible nor does the action anything. It can be used in places where a non-`null` action is required, but the developer does not want to show an action.

3.1.7 Predefined actions

There are a number of actions which are added automatically to a `CDockable`. For example the close-action or the maximize-action. These actions are often rooted deeply in the system and it is not possible to access them. However it is possible to override them. Every `CDockable` has a method `getAction` which is called before one of the predefined actions is shown. If that `getAction` method returns a value other than `null`, the predefined action gets replaced.

An example:

```
1  DefaultSingleCDockable dockable = ...
2  dockable.setMaximizable( true );
3  dockable.putAction( CDockable.ACTION_KEY_MAXIMIZE, CBlank
       .BLANK );
```

This code produces a dockable that is clearly maximizable (line `2`). However when the programm is running, the maximize-action is not visible. That is because in line `3` the predefined action for maximizing gets overridden by a new action. Since the new action is a `CBlank`, the action gets in fact invisible.

This feature has to be treated very carefully. Every predefined action has a special meaning, however the replacing actions depend on the client only. The developer has to ensure, that the replacing actions do something meaningful.

3.2 The location of dockables

Naturally a dockable has some location. This section deals with the various forms of locations that are used in DF.

3.2.1 for a single Dockable

For one dockable alone, the location is represented through a `CLocation`. Calling `CDockable.getLocation()` gets the current location (if any) of a dockable, `CDockable.setLocation` immediatelly changes the location of a dockable.

`CLocation`s should be created using some factory methods. The first `CLocation` is optained through the static methods of `CLocation`, for example `CLocation.base()`. Afterwards one can use the methods of the newly created object.

```
1 CDockable dockable = ...
2 CLocation location = CLocation.base().normalSouth( 0.5 ).
    east( 0.5 ).stack( 2 )
3 dockable.setLocation( location );
```

Let's have a look at line 2. First a base-location is created, indicating that `dockable` will be a child of the main content-area. Then `normalSouth( 0.5 )` tells us, that `dockable` will be normalized, and in the bottom half of the mainframe. The step `east( 0.5 )` puts `dockable` in the lower right quarter. And finally `stack( 2 )` allows `dockable` to be combined with other dockables that are already at that location. If there are already combined dockables at that location, then `dockable` will be inserted as the 3. element (counting starts at 0).

A common task is to open a dockable at the same location where another dockable is.

```
1 CDockable oldDockable = ...
2 CDockable newDockable = ...
3 CLocation location = oldDockable.getLocation();
4 newDockable.setLocation( location );
```

This fragment will move `newDockable` at the location of `oldDockable` and put `oldDockable` at a new location.

It looks very strange when a dockable moves away to give space for a new dockable. Often the new dockable should just be "aside" the old one. The method `CLocation.aside()` provides the client with such a location.

```
1 CDockable oldDockable = ...
2 CDockable newDockable = ...
3 CLocation location = oldDockable.getLocation();
4 newDockable.setLocation( location.aside() );
```

3.2.2 for a group of Dockables

Several dockables have to be registered, positioned and made visible when an application starts up. The `CGrid` is a class that can help doing all these things in one step. A `CGrid` represents a collection of dockables, where each dockable has some boundaries. The `CContentArea` and `CWorkingArea` can read the grid,

take over all dockables of the grid and arrange them such that their boundaries are matched as good as possible.

An example:

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

What happens here? In lines `1`, `3` and `4`, new objects are created. In line `6` a new `CGrid` is created. The new grid will add all dockables to `control`. Then in lines `7` and `8` dockables are added to the grid. `single` and `multi` will flank each other horizontally, `single` on the left side, `multi` on the right. The size of `multi` will be twice the size of `single`. Finally in line `10` the contents of the grid are put onto the `CContentArea` of the application.

3.2.3 Store the layout

With "layout" are all locations and relations of the dockables meant. It's a great help for the user if he can choose between various (predefined) layouts. For example: a LaTeXeditor might have a layout for editing the document, and a layout for looking at the compiled document.

The controller allows to store the current layout, load older layouts and delete layouts during runtime. Four methods deal with the layout: `save`, `load`, `delete` and `layouts`.

The method `CControl.save` gives the current layout some name and stores it. The method `CControl.load` searches a layout with a given name and changes the location of all dockables. The method `delete` removes a layout, and `layouts` returns a `String[]` with the names of all currently known layouts.

3.2.4 Store the layout persistent

Good applications store all their properties persistent (although many developers forget the directory which the file-chooser showed last). The user should not find any changes when he closes and restarts the applications.

The controller can use an `ApplicationResourceManager` to store its properties. However the client is responsible to tell the controller when and where to store its properties:

```
1  CControl control = ...
2  try{
3          control.getResources().readFile(
4                  new File( "/home/user" ));
5          control.getResources().writeFile(
6                  new File( "/home/user" ));
7  }
```

```
 8  catch( IOException ex ){
 9          ex.printStackTrace();
10  }
```

The properties can also be stored in xml-format. But the purpose of that option is mainly for debugging and migration, most applications will do just fine using a byte-stream.

3.2.5 Lazy initialisation

While `MultipleCDockable`s are created lazy anyway, `SingleCDockable`s normally are created in advance. Since that can use a lot of memory, lazy initialisation for `SingleCDockable`s was added in version 1.0.4.

Lazy initialisation is done through a `SingleCDockableBackupFactory` which is added to `CControl`. Each backup factory represents exactly one `SingleCDockable`. If the dockable is required (for example because the layout is read from a file), the factory will create the dockable and it will be automatically added to `CControl`. If a `SingleCDockable` is removed from `CControl` but a backup factory still is in place, then the properties of that dockable are not deleted. If another dockable is added later to the `CControl` with the same unique id, then this other dockable will inherit the old properties.

3.3 The size of dockables

Each `CDockable` has a certain size. Some `Component`s have an optimal size, others are flexible and can have any size. There are two interesting features for the first kind of `Component`s.

3.3.1 Lock size

Every `CDockable` has a method `isResizeLocked()`. If that methods returns `true`, then the `DockStation`s try not to change the size of the `CDockable` when the stations are resized themselfs. The user still can resize the `CDockable` by hand, and there are some other actions that will change the sizes as well.

Clients that use `DefaultSingleCDockable` and `DefaultMultipleCDockable` can use the method `setResizeLocked` to change the behavior.

3.3.2 Request size

Each `CDockable` can have a "size request" provided by `getAndClearResizeRequest`. Whenever `handleResizeRequest` of `CControl` is called, all resize requests are read and the stations try to change the size of their children such that the requests are fullfilled. There are no guarantees that the request can be fullfilled. And requests that cannot be fullfilled are simply lost.

Clients that use `DefaultSingleCDockable` and `DefaultMultipleCDockable` can use the method `setResizeRequest` to issue such a request. The method `setResizeRequest` can just store the request, or can process the request as well.

3.4 Colors associated with dockables

Every `CDockable` has a map full of colors. This map, the `ColorMap`, is read by the `DockThemes` which are included in the Common-project. If a client changes a color in the map, the change will be propagated to the graphical user interface immediatelly. There are various keys for different colors. If a client wants to change the color of a tab, it might use this piece of code:

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

4 Menus

Some things, like the list of all known dockables, can be presented to the user through a menu. Common uses `MenuPieces` to build new menus, each `MenuPiece` is set of items of a `JMenu`. A `MenuPiece` can be just a child of a bigger set of items. Several `MenuPieces` are put together in order to create the contents of one `JMenu`.

A collection of `MenuPieces`:

RootMenuPiece is the root of a tree of `MenuPieces`. It represents a whole `JMenu`. The `RootMenuPiece` is also a `NodeMenuPiece`.

NodeMenuPiece is a list of `MenuPieces`. Children can even be added or removed while the `JMenu` is visible.

FreeMenuPiece can be used by the client to insert its own `JMenuItems` in the menu.

SubMenuPiece represents a whole submenu. It adds a child-`JMenu` into a parent-`JMenu`. The child menu has its own `RootMenuPiece`.

SeparatingMenuPiece is a wrapper around another `MenuPiece`. It can add some separators above and below its child.

4.1 List of dockables

The `SingleCDockableListMenuPiece` contains a list of `JCheckBoxMenuItems`. Each item represents one "single dockable" of a controller. When the user clicks onto one item, then the associated dockable is either made visible or invisible.

The construction of a `CSingleDockableListMenuPiece`:

```
1 CControl control = ...
2 RootMenuPiece root = new RootMenuPiece( "List", false );
3 root.add( new SingleCDockableListMenuPiece( control ));
4 JMenu menu = root.getMenu();
```

4.2 Layout

Layout refers to the positions and relations of all the dockables. A controller can store various layouts, and a `CLayoutChoiceMenuPiece` allows the user to load, add or remove layouts whenever he wants to.

4.3 List of themes

A theme is something like a `LookAndFeel`. It defines the look and to some degree also the behavior of DF. There are various themes available, and the user can exchange them through a `CThemeMenuPiece`.

Creating a `CThemeMenuPiece` has an additional effect: the theme will be stored persistent if the client stores the layouts persistent.

4.4 LookAndFeel

There is a menu that exchanges the `LookAndFeel`, its name is `CLookAndFeelMenuPiece`.

Common uses the class `LookAndFeelList` to manage the `LookAndFeel`. This list offers a set of `LookAndFeel`s, and the `CLookAndFeelMenuPiece` is a reflection of that set. Only one `LookAndFeel` can be set at a time, and when changing the `LookAndFeel` the method `JComponent.updateUI()` has to be called on all `JComponent`s. The `CLookAndFeelMenuPiece` will make sure, that the `updateUI`-method gets called on any `Window` showing at least one dockable of the controller the menu is associated with. However if there are other windows, then their user interface will not get updated automatically. Then the client has to create a new `ComponentCollector` and add this collector to the `LookAndFeelList`

An example:

```
1  final JDialog dialog = ...
2
3  ComponentCollector collector = new ComponentCollector(){
4          public Collection<Component> listComponents(){
5                  List<Component> result = new ArrayList<
                       Component>();
6                  result.add( dialog );
7                  return result;
8          }
9  };
10
11 LookAndFeelList list = LookAndFeelList.getDefaultList();
12 list.addComponentCollector( collector );
```

Let's have a look at the code. In line `1` some dialog is defined. In lines `3-9` a `ComponentCollector` is defined. `collector` contains one method: `listComponents`. This method just collects some `Component`s, in this case `dialog`. For every `Component` found through a `ComponentCollector`, the `LookAndFeelList` will search the upermost parent and then update all children of this parent. That includes even child-`Window`s.