University of West Bohemia
Faculty of Applied Sciences
Department of Computer Science and Engineering

Master Thesis

Creating advanced GUI for desktop applications in Java

Pilsen, 2013

Jan Kolena

I hereby declare that this master thesis is completely my own work and that I used only the cited sources.

Pilsen, 4.5.2013

Jan Kolena

Abstract

This master thesis describes development of a new framework, which provides very simple creation of GUI, based on predefined template, applicable to great number of common applications. It will be suitable for both teachers and students, who want to develop a user – friendly application with focusing on the functional side and want to build the GUI quickly but efficiently.

In the thesis, there are analysis of current GUI building situation, the framework design, its implementation and testing.

Content

1 Introduction

In Java programming language there is a GUI provider Swing. Programmers use it when they need to build their GUI. Unfortunately, some important functions, which are frequently needed, are missing in Swing. Furthermore, the programmer often wants to focus only on functionality of his/her program and does not want to bother himself with GUI building.

It is the main goal of this thesis to describe development of a small framework[1] based on Swing, named **KIV framework**. The KIV framework should provide the possibility of simple creation of GUI, based on predefined template, suitable for most of common application. It will be suitable for both teachers and students, who want to develop a user–friendly application with focusing on the functional side of the application and want to build the GUI very quickly but efficiently. On the other hand, it has no ambitions to compete long–term developed frameworks in their functionalities. It takes the best and most important ideas from a few existing solutions and solves the problems in its own way.

The framework is developed in pure Java and on a top of Swing, but opened to re-implement it in some other way.

The first part of this thesis consists of requirements for the framework. Next chapter is about analysis of current situation in GUI building. There are Swing possibilities described in this chapter and afterwards there is a comparison between some existing solutions, which have partially the same functionality as designed framework.

The third part is focused on the frameworks design, based on the previous parts. All parts of the framework are described briefly, together with their diagrams. The next part deals with the implementation of the framework. Each part of the framework is described there in depth both from implementation and API point of view.

The next–to–last part describes testing of the project – JUnit[2] tests as well as manual tests and one performance test were implemented.

The conclusion at the very end of this thesis summarizes achieved results and sketches out plans to the future.

[1] Universal, reusable software library used to develop applications, especially by other programmers than by author of the framework.

[2] JUnit is a unit testing framework. It is used for automatic testing of software written in Java.

2 Requirements for framework

2.1 Application template

Application template provides very quick and simple method to create GUI (Graphical user interface). It consists from template and set of components, which are inserted into the template.

2.2 Form layout

It is often required to arrange components to a form. The framework provides a form layout or a component, which can arrange inserted components to a form.

The form layout/component should provide components input validation.

2.3 Action binding

The framework makes it possible to use actions binding instead of (or together with) listeners. Different components have different types of actions allowed. Actions should be bindable to different components repeatedly.

2.4 Data binding

The framework allows to bind some data source to a table component or list component. The binding makes it easy to synchronize data in the source and in the component. The framework contains default use data source for both of table and list component. The binding package contains an interface to simple implement any other data source than the implemented ones.

2.5 Factories

The framework includes factories for creating buttons, borders, fonts and icons. These factories provide an option to get standard looking buttons, use non-standard fonts or use common icons. The framework contains new fonts and icons packed in framework distribution file.

2.6 Memory filesystem

The framework includes implementation of filesystem, compatible with other filesystems and stored in memory. The main point of this filesystem is to keep private data safe from other processes. The speed of this system is not one of top requirements.

2.7 EXIF

The framework contains class (or package) for reading EXIF data from JPEG files. EXIF data writing is not necessary. The class offers methods for getting common entries directly (like exposure time, focal length, aperture, etc.)

2.8 File downloader

File downloader is usable for downloading of file from computer net. It provides rich process control but does not bother the programmer with streams and such things. The downloader should run asynchronously (in own thread) to not block the rest of application.

2.9 Text/file encrypter

The framework allows to encrypt/decrypt texts and files with miscellaneous methods (AES, DES, …) but without deep knowledge of the problematic. The encrypting class should provide also methods creating hash or CRC of text/file. Encrypted files and texts do not have to be decryptable by any other tool.

2.10 ZIP handler

ZIP handler is class representing single ZIP file. It offers all common operations for files within the ZIP. It can cooperate with the file encrypter described above. Created ZIP files have to be fully compatible with ordinary ZIP handling tools outside the framework.

3 Swing and its alternatives

Swing is an API for providing GUI for Java programs. It was developed to provide a more sophisticated set of GUI components than the earlier AWT. Swing provides a native look and feel that emulates the look and feel of several platforms, and also supports a pluggable look and feel that allows applications to have a look and feel unrelated to the underlying platform. Unlike AWT components, Swing components are not implemented by platform-specific code. Instead they are written entirely in Java and therefore are platform-independent.

Swing is part of JFC[3], which is set of GUI components and services for GUI development. It is included in basic JDK, so the developer does not need to download any additional library. (1)

3.1 Advantages

- The biggest advantage of Swing API is its platform independency. The developer can be sure that his/her software will look the same on all supported platforms.
- The work with Swing is quite simple and there are lot of examples and tutorials on the Internet, including the website of Oracle. (2)
- Swing includes many components for common usage.
- Swing supports many visual themes (called look'n'feels). Everyone can create his/her own, but there are many of them available on the Internet.

3.2 Disadvantages

Many often used functions are not available in Swing. For example:

- Form layout – the form building is quite hard but often needed
- Quick building – although the building is simple, developer has to burden himself with the building
- Data binding – communication between a data layer and GUI has to be programmed by each developer who needs it
- Validation – validation of text inputs against pattern

[3] Java Foundation Classes.

3.3 Alternatives

There are several interface toolkits with similar functions as Swing provides. Two of them are from Oracle (more precisely from Sun that developed the Java programming language before its Oracle takeover) – AWT and JavaFX.

3.3.1 AWT

The AWT (Abstract Windows Toolkit) is a predecessor of Swing. It provides a few widgets (components) based on native libraries for all supported platforms. The biggest advantage and also the disadvantage at the same time is only native look at all platforms. Another disadvantage is occasional problems with components behavior through different platforms. The AWT was released in 1995 (as Java was).

3.3.2 JavaFX

JavaFX is the newest platform for GUI building developed by Oracle. It is designed to be run across many different devices. It currently supports desktop computers and web browsers. It is expected that JavaFX will fully replace Swing, like Swing replaced AWT. On the other hand, now it is a new platform with relatively small support.

3.4 Reasons for Swing

According to the previous paragraphs, the KIV framework could be based on Swing or on JavaFX (AWT is unsatisfactory). Because of better support and widely knowledge (1), Swing was chosen.

4 Existing solutions

There are libraries which solve some issues introduced in chapter 3.2. They have been released under different licenses and usage terms.

4.1 SWT

The Standard Widget Toolkit is an alternative of AWT. Similarly to AWT and on contrary to Swing, the SWT uses native libraries (written in GTK+ etc.) via JNI (Java native interface).

Another similarity to AWT is that SWT does not support any look and feel except the platform specific. Its functionality is almost at the same level as AWT (3).

```
Display display = new Display();
Shell shell = new Shell(display);
Label label = new Label(shell, SWT.NONE);
label.setText("Hello World");
label.pack();
shell.pack();
shell.open();
while (!shell.isDisposed()) {
    if (!display.readAndDispatch ()) display.sleep ();
}
display.dispose ();
```

Code 1: SWT usage example

As shown in Code 1, the SWT usage is quite similar as the AWT or Swing. SWT does not extend functionality of the Swing provider and therefore is not suitable as a solution satisfying all described requirements.

4.2 Pivot

Apache Pivot is a modern development platform for GUI building, developed as an open source by Apache Software Foundation. It is designed for building installable Internet applications (IIAs). It combines the enhanced productivity and usability features of a modern user interface toolkit with the robustness of Java platform.

```
public class HelloBXML implements Application {
    private Window window = null;
    public void startup(Display display, Map<String, String> properties){
        BXMLSerializer bxmlSerializer = new BXMLSerializer();
        window = (Window)bxmlSerializer.readObject(Hello.class,
"hello.bxml");
        window.open(display);
    }
}
```

Code 2: Apache pivot BXML layout loading

The Pivot library allows developers to easily construct visually-engaging, cross-platform, connected applications in Java or any other JVM language, such as JavaScript, Groovy, or Scala. (4)

```
<Window title="Hello BXML!" maximized="true"
    xmlns:bxml="http://pivot.apache.org/bxml" xmlns="org.apache.pivot.wtk">
    <Label text="Hello BXML!"
        styles="{font:'Arial bold 24', color:'#ff0000',
            horizontalAlignment:'center', verticalAlignment:'center'}"/>
</Window>
```

Code 3: Apache Pivot BXML layout definition

Pivot offers also the possibility to create GUI, based on XML documents, as shown in Code 2. The XML (Extensible markup language) layout definition shown in Code 3 is similar to Android platform.

The Pivot library satisfies some requirements, introduced above, but only a few of them. This library is therefore not suitable as final solution.

4.3 JGoodies

JGoodies is one of the best and widely known frameworks for GUI development. It consists of commercial and freeware parts. Freeware JGoodies library is a very big competitor of the KIV framework. It includes Binding, Forms, Validation and some other libraries. Its functionality mostly surpasses the KIV framework, but it has a lot of unnecessary functions from the other point of view.

JGoodies solves most of introduced issues and the development is quite simple and quick. Its problem lies in license, which can be modified from day to day and it can bring some license problem to derived applications.

```
public JPanel createSettingsPanel() {
FormLayout layout =
  new FormLayout("5dlu,pref,5dlu,pref,pref:grow,3dlu,pref,3dlu,pref,5dlu",
      "pref,5dlu,pref,1dlu,pref,1dlu,pref,1dlu,pref,10dlu,pref,5dlu," +
      "pref,10dlu,pref,5dlu,pref,10dlu,pref,5dlu,pref,10dlu," +
      "pref,5dlu,pref,3dlu,pref,10dlu,pref,5dlu,pref");
  PanelBuilder pb = new PanelBuilder(layout, new ScrollableJPanel());
  pb.setDefaultDialogBorder();

  CellConstraints cc = new CellConstraints();
  final String[] extArr = { ".wav", ".aif", ".rmf", ".au", ".mid" };
  String soundFName=mSettings.getProperty("soundfile","/");
  String msg = mLocalizer.msg("soundFileFilter", "Sound file ({0})",
      "*.wav, *.aif, *.rmf, *.au, *.mid");
```

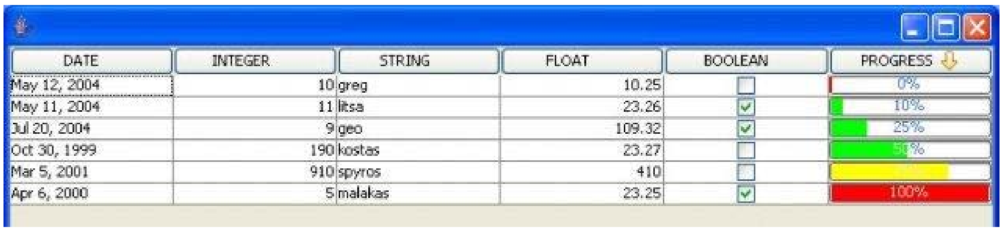
Code 4: JGoodies Forms example

Another imperfection of JGoodies is an absence of mechanism for quick building – something like the application template introduced in section 2.1. The JGoodies GUI building is quite difficult as you can see in Code 4. The JGoodies usage is best with some WYSIWYG UI builder.

However, JGoodies has served as one of the inspirations for the KIV framework design.

4.4 ZEUS

Zeus is a Java Swing Components Library. It provides useful Swing components for easier GUI development. Likewise JGoodies also ZEUS offers a few components, which are suitable for integration to the KIV framework – such as Console, TableSorter or SplashScreen. (5)



DATE	INTEGER	STRING	FLOAT	BOOLEAN	PROGRESS
May 12, 2004	10	greg	10.25	☐	0%
May 11, 2004	11	litsa	23.26	✓	10%
Jul 20, 2004	9	geo	109.32	✓	25%
Oct 30, 1999	190	kostas	23.27	☐	50%
Mar 5, 2001	910	spyros	410	☐	[illegible]
Apr 6, 2000	5	malakas	23.25	✓	100%

Figure 1: ZEUS TableSorter screenshot

There are a lot of described problems, which the ZEUS does not solve and therefore is not suitable as a complete solution. However, it can be used as an inspiration for some functionality.

As shown in Figure 1 ZEUS can sort table columns according to miscellaneous data types.

4.5 Buoy

Buoy (A Better User Interface Toolkit) is a toolkit for creating user interfaces in Java programs. You can think of it as a replacement for Swing and AWT, although that is not entirely correct. Buoy is built on top of Swing, so when you use Buoy to create a user interface, Swing components are still being created behind the scene. (6)

```
public class BLabelDemo extends BFrame{
  public BLabelDemo(){
    super( "BLabelDemo" );
    BorderContainer bc = new BorderContainer();
    FormContainer fc = new FormContainer( 3, 3 );
    ImageIcon icon = new ImageIcon( "icon.png" );
    fc.add( new BLabel( "North", icon, BLabel.CENTER, BLabel.NORTH ), 1, 0,
     new LayoutInfo(LayoutInfo.SOUTH,LayoutInfo.NONE,new Insets(0,0,0,0),
null));
    fc.add( new BLabel( "East", icon, BLabel.CENTER, BLabel.EAST ), 2, 1,
```

Code 5: The BUOY example

According to the description the Buoy is based on very similar idea as the KIV framework is, but the KIV framework needs a few additional functionalities and (primarily) the GUI template. As shown in Code 5 the BUOY has quite complicated positioning definition. Another issue is that Buoy is quite an old release (2009) and it seems to be an inactive project.

4.6 CookSwing

CookSwing is a library which builds Java Swing GUI from XML documents. It is under continuously active development. Unlike many other XUL toolkits, Swing is complete in its capability dealing with Swing and beyond. (7)

CookSwing can be used for creation of unified GUI via a template. CookSwing templates are written in XML and it partially solves the main goal of the KIV framework. Unfortunately, it does not satisfy any other requirement described in chapter 2. Another issue is that common usage requires knowledge of XML structure and it might be harder to use in final consequences.

```
<buttongroup>
    <idref ctor="menu_2" />
    <idref ctor="menu_3" />
</buttongroup>
<buttongroup>
    <idref ctor="button_2" />
    <idref ctor="button_3" />
</buttongroup>
```

Code 6: The CookSwing XML example

There is an example of ButtonGroup in Code 6. It presents synchronization between a menu button and a RadioButton.

4.7 Competition resume

There are several platforms/libraries described above. All of them solve some introduced problems, but no one solves all of them. One possible solution is to combine more of them – for example combination of Buoy with JGoodies and CookSwing would solve almost all of the introduced problems, except the utilities like MFS. On the other hand, this method would lead to fragmentation of the final solution and in final consequences it would be very hard to work with it.

Much better approach is to get inspired by all of described solutions and create own one, standing on strong base of Swing and including some extra features like mentioned GUI template, data binding, form layout and other functions described in chapter 2.

5 Design of framework

5.1 Name convention

Each created class in the KIV framework will have name prefix **ZK**, what is derived from the faculty department name – **Z**ČU **K**IV. Additionally all interfaces will include the **I** letter in their name – it means interfaces prefix will be **ZKI**.

5.2 Components and application template

As mentioned above, it is the principal idea of the framework to allow programmer to create a GUI with a minimal effort. To fulfill this, a new set of components has to be implemented and as well as a new way to handle them. On the other hand, it would be advantageous to have some well – known system behind the new one, to enable the programmer to handle the components in the old way too. In chapter 3.4 there is an explanation, why Swing is the best choice.

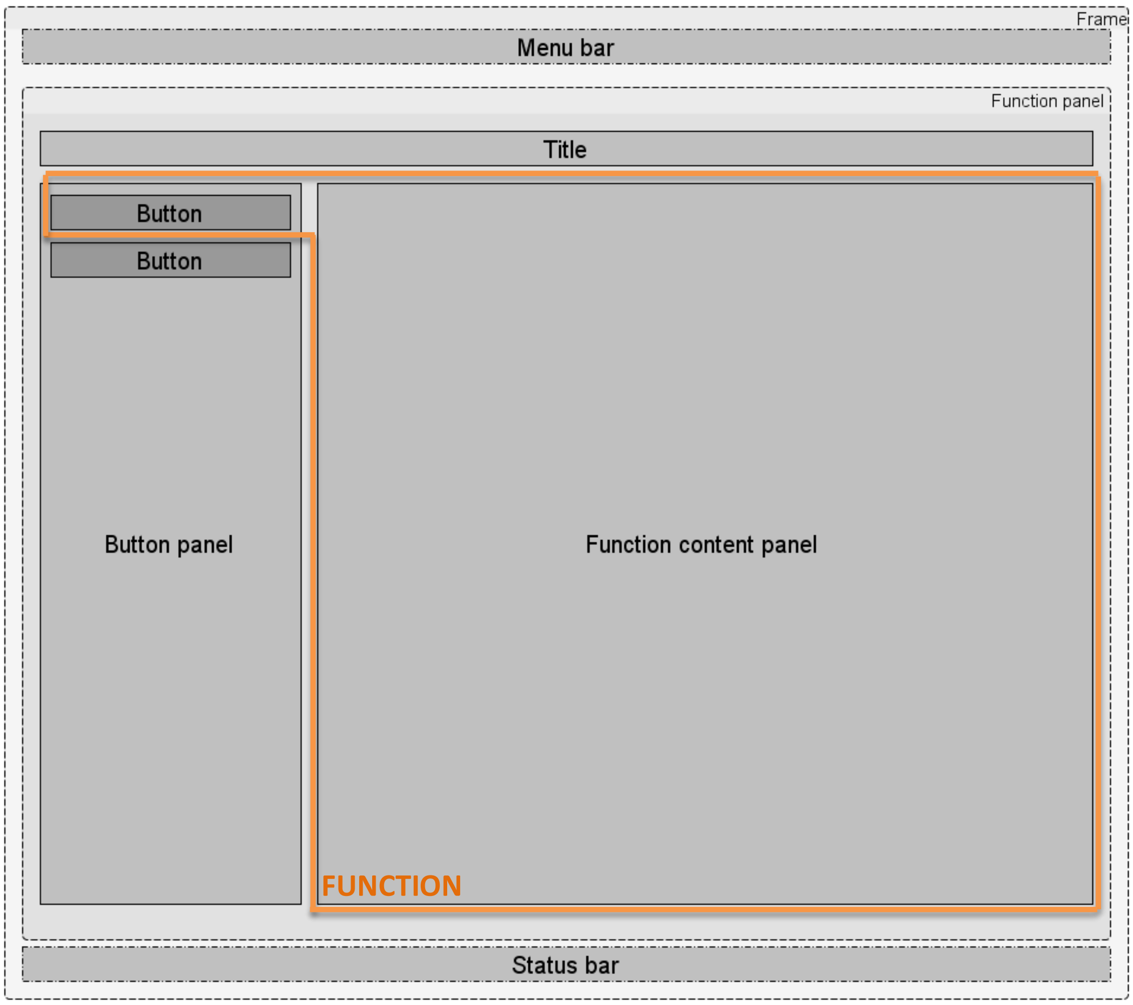


Figure 2: The template layout

5.2.1 Template layout

Figure 2 shows the template layout. The application window (frame) contains a function panel (hereinafter **FP**). The FP contains one or more functions. Each function is pair of a content panel and a button, which shows a related content. All buttons are placed in the button bar on the left side of the FP.

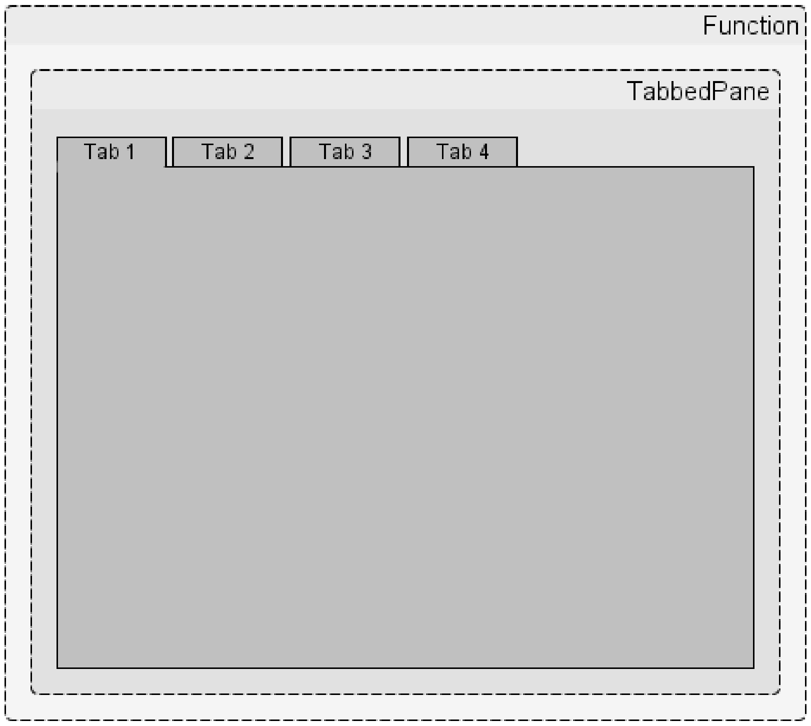


Figure 3: Function with TabbedPane

Figure 3 shows a function, which has a tabbed pane inside (function with a few tabs).

5.2.2 Interfaces draft

According to current trends in software engineering, interfaces are to be created first. The interfaces serve for accessing the components. Because of Swing, it is advantageous to use as many existing method as possible – usage of same method names could lead to automatic implementation of many methods (overtaken from extended Swing component).

This is a complete list of designed general (not component-specific) interfaces:

- ***`ZKIComponent`***
  - used by every component
- ***`ZKIContainerComponent`***
  - components, which can contain other components, such as panel etc.

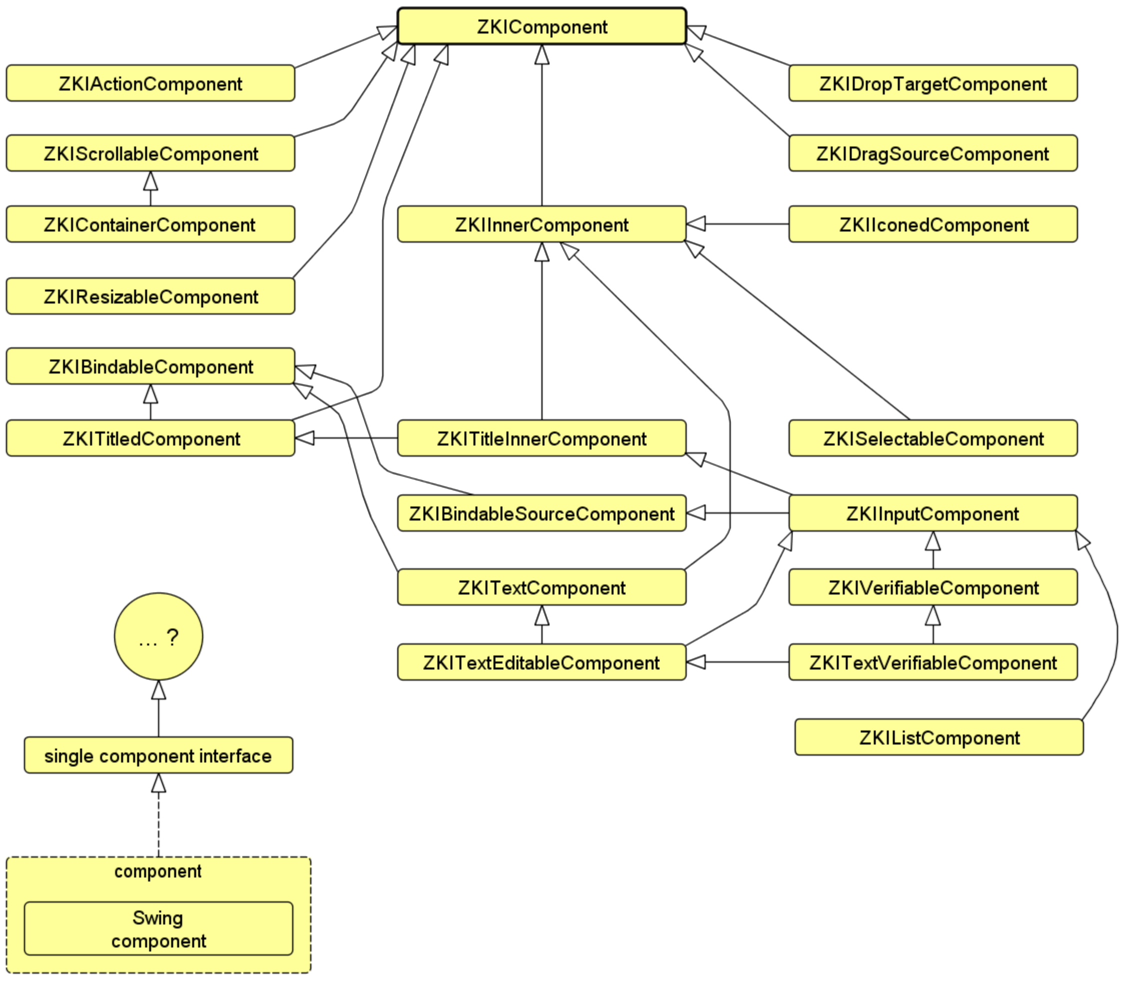


Figure 4: Tree of interfaces

- ***ZKIInnerComponent***
  - inner components (almost all except a frame)
- ***ZKIBindableComponent***
  - component, which can be used as binding target
  - does **not** extending ***ZKIComponent*** – used in ***ZKTableColumn***, which is **not** a component
- ***ZKIBindableSourceComponent***
  - component, which can be used as a binding source (and target)
  - extends the ***ZKIBindableComponent*** interface (therefore **not** extends ***ZKIComponent***)
- ***ZKIActionComponent***
  - all components which have some kind of default action (e.g. after a click)

- ***ZKIInputComponent***
  - all components with some kind of input
- ***ZKIListComponent***
  - all components displaying list of values
- ***ZKITextComponent***
  - components holding some text
- ***ZKITextEditableComponent***
  - components with editable text
- ***ZKIVerifiableComponent***
  - used by some input components (form validation etc.)
- ***ZKITextVerifiableComponent***
  - used by some text components with input pattern
- ***ZKISelectableComponent***
  - selectable components like checkbox, radiobutton etc.
- ***ZKIResizableComponent***
  - almost all components, except tab panel and few others
- ***ZKIIconedComponent***
  - all components with icon
- ***ZKITitledComponent***
  - component with title
- ***ZKITitledInnerComponent***
  - inner components with title (***ZKIInnerComponent*** + ***ZKITitledComponent*** + **showTitle(boolean)** method)
- ***ZKIScrollable***
  - component with scrollable content
- ***ZKIDragSourceComponent***
  - components which support their dragging
- ***ZKIDropTargetComponent***
  - components which support dropping some dragged component on them

As you can see in Figure 4, each component will implement only one interface, but the interface can extend various other interfaces.

Example:

ZKIFunction will extend ***ZKIInnerComponent***, ***ZKIContainerComponent***, ***ZKITitledComponent*** and add some own methods.

5.2.3 Components

All common used Swing components will be extended to fit the KIV Frameworks requirements. Additionally there will be a few new components, derived from the standard or integrated from third party library.

The extended Swing components list:

- ***ZKButton***
- ***ZKCheckBox***
- ***ZKComboBox***
- ***ZKEditorPane***
- ***ZKFormattedField***
- ***ZKFrame***
- ***ZKLabel***
- ***ZKList***
- ***ZKPanel***
- ***ZKPasswordField***
- ***ZKProgressBar***
- ***ZKRadioButtonGroup, ZKRadioButton***
- ***ZKSpinner*** (handles only numbers, see ***ZKObjectSpinner*** below)
- ***ZKSplitPane***
- ***ZKTabbedPane***
- ***ZKTable***
- ***ZKTextPane***

The following list contains new components with brief description:

- ***ZKCalendar*** – from 3rd party library (8), will display given date and enable to choose a new one
- ***ZKCanvas*** – derived from ***ZKPanel***, will provide functionality to direct draw by calling standard ***Graphics*** methods
- ***ZKDateField*** – from 3rd party library (8), will enable to edit date value by direct editing or by choosing from a floating calendar
- ***ZKForm*** – derived from ***ZKPanel***, will arrange given input components to a form layout
- ***ZKFunction*** – the function (is described in 5.2.1)
- ***ZKFormFunction*** – composite of ***ZKFunction*** and ***ZKForm***
- ***ZKFunctionPanel*** – the function panel, as described in 5.2.1
- ***ZKIcon*** – derived from ***ZKLabel***, will wrap the standard Icon instance, with all necessary methods for direct usage, including drag-and-drop as described in 5.3

- ***ZKObjectSpinner*** – as standard ***JSpinner***, can handle miscellaneous objects
- ***ZKMainFrame*** – derived from ***ZKFrame***, designed as main window of the application, will contain few ***ZKFunctionPanel*** instances and a ***ZKStatusBar***
- ***ZKStatusBar*** – derived from ***ZKPanel***, will be used for displaying some information in frame footer
- ***ZKTab*** – derived from ***ZKPanel***, will serve as one single tab in ***ZKTabbedPane***
- ***ZKTabbedFunction*** – composite of ***ZKFunction*** and ***ZKTabbedPane***
- ***ZKWrapLayout*** – derived from ***FlowLayout***, will allow component wrapping

Detailed information about the components is provided in the implementation part of this thesis.

5.3 Drag-and-drop

The drag-and-drop (with shortcut D’n’D) function is a mouse gesture in which we can drag some object onto another and invoke some action with that. The D’n’D simple gesture is very popular by users, because it can replace a few clicks and is quite intuitive.

Implementation of D’n’D can be complicated in some cases. Although Java provides support for D’n’D for most components, not all the components support is comfortable. This is the problem which the KIV framework wants to solve. The KIV framework will make implementation of D’n’D easier for some components and will also make possible to use D’n’D with components not originally supported by Oracle.

5.4 Action binding

In standard Swing there are several listeners, which can help the programmer to catch several user actions, like mouse move or click. This possibility will be preserved but a new alternative to assign a predefined action to the component will be added. There will be a few actions implemented and the programmer will have possibility to assign action instance to the object. Depending on the type of the action the object will map the action to a proper listener and the programmer will not have to care about it.

5.4.1 Actions

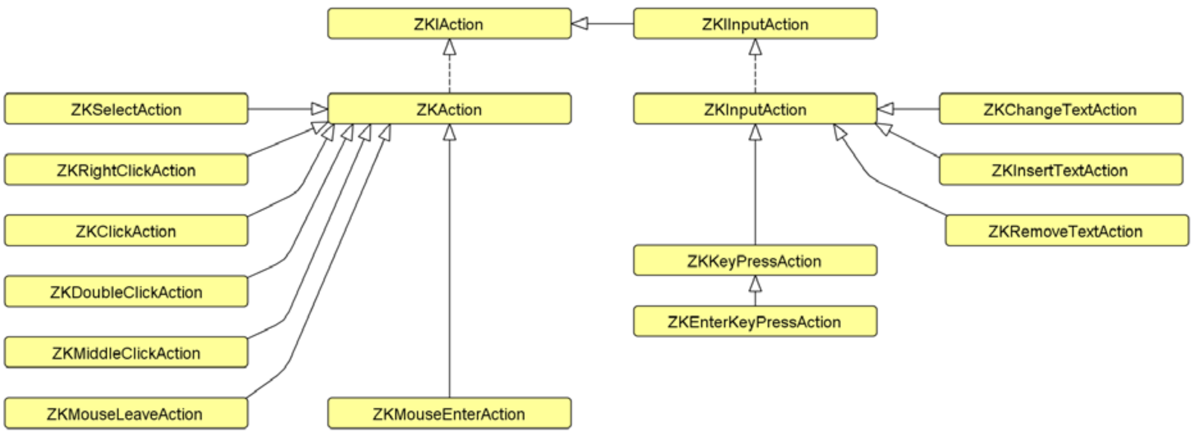


Figure 5: Actions diagram

All the predefined actions will extend a ***`ZKAction`*** or ***`ZKInputAction`***, which are abstract classes, implementing ***`ZKIAction`*** or ***`ZKIInputAction`*** interface, as shown in Figure 5. The interfaces ensure the action class has an **`action()`** method, which the programmer will have to implement and a **`start()`** method, which will invoke the **`action()`** method. The **`action()`** method will not be called directly, because there will be a possibility (via constructor) to run the action in its own thread and the **`start()`** method will handle it.

All designed actions are visible in Figure 5. Their names are proposed to be self–explainable.

5.5 Data binding

It is often required to display some data from various sources. The programmer usually needs to create a loader for the data and also implement a listener, which will update the data in the GUI. The worse case is when the end user can change the data and the program should save the data back to the original source. Another situation is when he/she wants to simply change the section header depending on some user choice.

Situations described above are very difficult and stressful to solve. This is the reason why the data binding exists. The programmer says what the data source and the target are, a binding strategy (whether the binding should by only from source to target or two–way also back to the source) and the binder will synchronize the source and the target automatically.

In the KIV framework there will be a simple binder. It will be possible to synchronize some properties of components (like the section header described above) or to synchronize some data source to the table etc.

5.6 Content verification

Every programmer needs a user input verification from time to time. This is important especially in form, when the user should enter some text according to a pattern. The KIV framework text components ***ZKFormattedTextField***, ***ZKTextField*** and ***ZKTextPane*** will implement ***ZKITextVerifiableComponent***, which secures that the programmer is able to set an allowed pattern for the component and the component will automatically validate the input against the pattern.

The components will have methods for validity detection and they will automatically change their appearance according to the current validity status.

5.7 Tooltip formatter

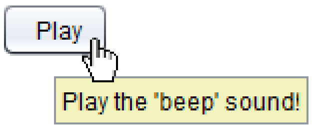


Figure 6: Tooltip example

The standard Swing tooltips[4] layout is simple and strictly functionalistic, as shown in Figure 6. The programmer cannot change its appearance or format the text. The only possibility to format the tooltip is to use a HTML code, but that is quite difficult and time consuming.

The KIV framework will offer a tool named ***ZKTooltipFormatter***, providing an option to format the text very quickly, because the formatter will do the real formatting internally. There will be several classes representing most frequently used HTML tags and the programmer will have to do minimum for having nice and structured tooltips.

5.8 Factories

5.8.1 Factories in general

There will be several factories implemented in the KIV framework. A factory is an object that creates an unrelated object on behalf of another unrelated object. The intent of using a Factory is to reduce the coupling between classes, and to make class more reusable by making it independent on other classes. (9)

[4] The tooltip is a floating hint, which will appear after pointing with a mouse to some component. Each component typically has its own tooltip. The tooltip should explain function of the component more to the depth than the component title (if it has any).

5.8.2 Dialog factory

The dialog factory should provide the programmer even more simplified dialog creation then Swing offers. It will contain methods with less options then Swing has, such as **`showErrorMessage(Throwable throwable)`** or **`showMessage(String message)`**, which should lead to more comfort usage.

5.8.3 Icon factory

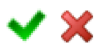


Figure 7: Example of icons

The icon factory should offer the programmer a set of commonly used icons and small pictures as "OK" or "CANCEL" (as in Figure 7) icon or picture used to represent loading process. There will be also icons for MS documents, for images or for PDF documents.

5.8.4 Border factory

Although there is a quite simple border factory in Swing, it can still be simplified, which is a task of ***`ZKBorderFactory`***. It will contain methods like **`roundedBorder(Color)`** or **`plainBorder(Color)`**.

5.8.5 Button factory

The button factory will have methods to quickly get some predefined buttons (with related icons) – "OK" button, "CANCEL" button and few others. It will use ***`ZKIconFactory`*** for the icons loading and have a method for each of the available icons.

5.8.6 Font factory

Although Java can use almost every font installed in the PC, there are only few fonts which the programmer can rely on their presence.

The font factory will make easier to choose from the trusted fonts. It will also provide some new fonts and predefined colors – the programmer will be able to choose from some non–standard fonts, which will be included in JAR (Java archive) with the framework.

5.8.7 Sound factory

This factory will be a part of utilities and is described in chapter 5.9.8.

5.9 Utilities

An independent part of the KIV framework will be utilities. It will not depend on anything in the rest of the framework so the programmer will have a chance to get a "light release"[5] and use only the utilities.

The utilities will include miscellaneous things which are not related to the GUI building but are useful in every application the programmer makes.

5.9.1 Memory file system – MFS

The MFS will allow the programmer to copy a file to a memory (RAM) and handle it almost like an ordinary file (through the ***`Path`*** pointer). The advantage against the disk holding method is a protection of the file from all other processes and also from the user. For example, when some important configuration file is stored in encrypted ZIP archive[6], the program has to extract it to the disk to access it. At this very moment the file is accessible by the rest of the system, including some harmful processes which can damage the file or change important values inside. On the other hand, if the program extracts the file from the ZIP directly to the memory file, no decrypted data are stored on the disk and the file is accessible only from the one program[7].

5.9.2 ZIP files

The framework will include functionality for basic operations with files in the ZIP archive. The programmer will be able to create/open archive; add files/directories, delete them, extract them to the disk/to a memory file. The ZIP files will have an option to be encrypted with a password.

5.9.3 Encryption

There is an occasional requirement to encrypt some text or file. The framework will contain functionality to encrypt/decrypt of the mentioned above. The ***`ZKCrypt`*** class will include method to create a hash of some ***`String`*** or file too.

5.9.4 File utilities

The file utilities class named ***`ZKFileUtils`*** will have a few methods for file handling, like getting file content to a ***`String`*** or calculating the file CRC.

[5] A release where will not be all parts of the framework.
[6] Also one of the functions of the framework – see 4.8.2.
[7] Memory protecting is one of the basic functionalities of the OS.

5.9.5 Download

In modern applications, there is often a requirement to download some file from the Internet (or from the computer net in general). The KIV framework will contain an abstract class giving the programmer a possibility to simply download the file. The programmer will have to implement his/her own derivate of ***`ZKFileDownloader`***, because the class will have a few methods serving as a process callback and they will have to be implemented.

5.9.6 EXIF

All the modern photo cameras include a special metadata named EXIF to the photos they make. The EXIF contains information about the camera, the photo exposure (shutter speed, aperture, etc.) and sometimes a GPS coordinates too. The ***`ZKExif`*** class will allow the programmer to read EXIF (version 2.2) metadata from a JPEG file.

5.9.7 JSON converter

The JSON is a format for describing an object. It is used mainly in web programming, especially in Javascript.

The ***`ZKJsonConverter`*** tool will be able to convert a standard Java properties file to a standard JSON format. This is useful e.g. when the programmer in Java EE has an internationalization properties file and he/she needs to access it from client–side programming language (like mentioned Javascript).

The class will be prepared to be used in ANT script.

5.9.8 Sound factory

The programmer often needs to play some sound, which should point out some action or notify an error. The classic example is the end of long action running on the background of application (e.g. download).

`ZKSoundFactory` will provide methods to play some often used sounds (included in JAR) and also a method for playing any WAV/MP3 file.

5.9.9 Czech comparator

Although Java can determine the default ***`Locale`*** instance from the computer which it is running on, there is a problem with sorting of ***`String`*** set with all Czech letters, including the accents. The programmer has to use a manually created ***`Locale`*** and ***`Comparator`*** instance to sort the set properly. The KIV framework will include the correct comparator named ***`ZKCzechComparator`***.

5.9.10 Console

Java has a very inconsistent method to access the console. Although there was the ***`Console`*** class introduced in JDK1.6, it still does not have all functions to make reading and writing from/to the console comfortable.

The KIV framework will have a ***`ZKConsole`*** class, which will offer all commonly used methods for reading and writing the console. There will be also a ***`ZKCommandParser`*** class implemented, which will offer commands parsing.

5.10 Testing of project

5.10.1 Testing in general

5.10.1.1 Automatic testing

There are 4 basic types of SW tests:

1) Unit test – the programmer writes a test for his/her own code to find possible errors
2) Integration test – a SW tester writes a test for verifying cooperation of individual components
3) System test – a SW tester writes tests to verify that the system meets its requirements
4) Load test – test which should measure performance of some module

All described tests are automatic. The programmer or tester writes them once and they can be run repeatedly to be sure it did not stop working. (10)

5.10.1.2 Manual testing

Unfortunately, there are some cases which cannot be tested automatically or the testing is very complicated. There is a simple solution of this situation: manual testing by the programmer or tester. An example of this case is the control of visual components – automatic tests cannot verify correct display of the component satisfyingly but a person can verify it visually.

5.10.2 Project testing

According to the previous paragraphs, some parts of KIV framework are suitable for automatic and some for manual testing.

A utilities package can be tested by JUnit tests. JUnit tests can be written for almost every class or package from the utilities to verify correct functionality of the program. ***`ZKSoundFactory`*** is an exception from the rule mentioned above. It cannot be tested automatically. The manual testing described in chapter 5.10.1.2 is used for this and a few others classes.

As explained above there is a problem with GUI testing. It do exist solutions for GUI testing, but they are all focused on existing GUI provider such as Swing. It could be a problem to use it in this framework. Another issue is that creation of these tests would be very time consuming. After evaluation of all aspects it was decided to use particularly manual testing (JUnit tests only for a few cases described below). Unlike a classic code (like the utilities) the GUI can be tested to the depth only few times and then only checked some key points repeatedly.

On the other hand, some parts of the GUI can be tested automatically. It is e.g. management of tabs, columns in table or behavior of date parsing in ***`ZKDateField`***. It can be implemented tests for these cases.

6 Implementation

6.1 Preamble

The framework is released under the LGPL (Gnu Lesser General Public License)[8].

The framework needs JDK 7 to be used. All parts of the framework are interconnected together, so although some parts may be re-compiled under the JDK 6, the complex framework will never work. The JDK 7 brought many significant changes to Java development (11). Some of these improvements were used for better efficiency of implementation; some functionality strongly depends on classes introduced in JDK 7 (e.g. 6.9.1).

The project has its own page at http://sourceforge.net/projects/kivframework/ where a GIT repository is also available.

6.2 Application template

6.2.1 Events

6.2.1.1 Visibility events

All implemented components have two pairs of methods related to their visibility – **`boolean beforeShow()`** + **`void afterShow()`** and **`boolean beforeHide()`** + **`void afterHide()`**.

```
ZKIPasswordField passwordField = new ZKPasswordField("The password")
{
    @Override
    public void afterShow()
    {
        System.out.println("The password field has been shown!”);
    }
};
```

Code 7: Events overriding example

These methods are invoked while changing visibility of related component. Each component has all of these methods implemented with empty body (or returns TRUE), prepared to be overridden by the programmer as it is shown in Code 7. The second obvious possibility is to extend the class and implement the method(s) for all instances of it.

[8] Available at http://www.gnu.org/licenses/lgpl-3.0.txt.

```
ZKIPasswordField passwordField = new ZKPasswordField("The password")
{
    @Override
    public void beforeHide()
    {
        return (canBeHidden);
    }
};
```

Code 8: A beforeHide event function

The before* methods return a ***boolean*** value. These methods determine, whether the component can be show/hidden and stop the visibility change by returning **FALSE** or allow it by returning TRUE. This can be seen in Code 8.

6.2.1.2 Other events

Another event which components implement is **`afterInit().`** This event is invoked from constructor of each component, immediately after all variables and properties initialization. This event can be overridden too, for example for setting own properties or changing the components appearance. In most cases, this can be done also by calling the code after a creation command, but the **`afterInit()`** method could be advantageous is some cases.

Most components have their own events, invoked in various situations. For example ***ZKISelectableComponent*** (used in ***ZKCheckBox*** and ***ZKRadioButton***) contains other two pairs of events in addition to the visibility events: **`boolean beforeSelected()`** + **`void afterSelected()`** and **`boolean beforeUnselected()`** + **`void afterUnselected()`**. The meaning of these methods is quite similar to the visibility events, but there is a deeper consequence here.

There is a problem with ***ZKRadioButton***, because only one button from a related group can be selected at one moment. Both the selected buttons **`beforeUnselected()`** method and the selecting buttons **`beforeSelected()`** method can stop the change of selection.

There are a few more events in the components package, like **`beforeDrop()`** and **`afterDrop()`** in ***ZKIDropTagretComponent*** etc.

6.2.2 Content verification

As described above in section 5.6, some components implementing the ***ZKITextVerifiableComponent*** interface offer a user input validation. The pattern is realized by the ***Pattern*** class. It means the input is verified by a regular expression. The content change listeners are used for automatic behavior and, moreover, when the components are inserted into the ***ZKForm*** instance, the form can react to the components status change and change its appearance too.

6.2.3 Template layout

Both ***ZKFunctionPanel*** and ***ZKFunction*** extend an ordinary ***ZKPanel***.

6.2.3.1 Function

ZKFunction is an ordinary ***ZKPanel*** with associated button. It has taken over all properties of ***ZKPanel***, including a scrollable option, as described in 6.3.1.6.

There are ***ZKFormFunction*** and ***ZKTabbedFunction*** implemented in the framework. ***ZKFormFunction*** is simple ***ZKFunction*** containing instance of ***ZKForm***. Only ***ZKIInputComponent*** instances can be added into it. ***ZKTabbedFunction*** is ***ZKFunction*** containing ***ZKTabbedPane*** and only ***ZKITab*** instances can be added into it.

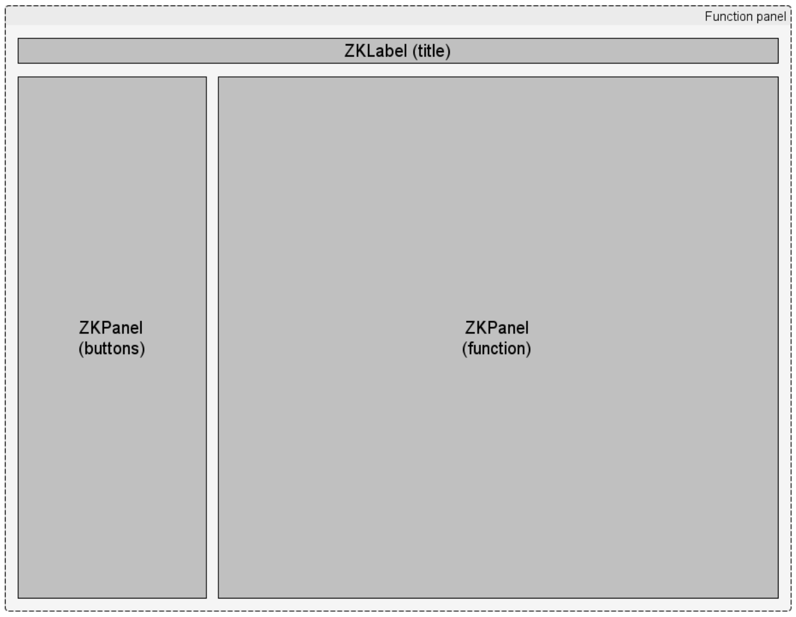


Figure 8: The ZKFunctionPanel component

The common ancestor of ***ZKFunction***, ***ZKFormFunction*** and ***ZKTabbedFunction*** is the ***ZKIAbstractFunction*** interface. This is because ***ZKIFunction*** extends also the ***ZKIContainerComponent*** interface, which contains methods like **add(Component)** which is not permissible for ***ZKFormFunction*** and ***ZKTabbedFunction***.

6.2.3.2 Function panel

As ***ZKFunction***, ***ZKFunctionPanel*** extends ***ZKPanel*** too.

ZKFunctionPanel contains a list of inserted functions (instances of ***ZKAbstractFunction***). It has also three subcomponents as shown in Figure 8. The label on the top shows a title of currently shown function. The function is shown in content panel, placed on the right side. On the left side, there is a button panel containing buttons from all associated functions.

6.2.4 ZKFrame, ZKMainFrame

6.2.4.1 ZKFrame

ZKFrame extends the ***JFrame*** Swing component and adds some extra features like scrolling (as described below in 6.3.1.6). It has also methods for positioning. The menu bar and status panel are contained in all ***ZKFrame*** instances automatically. Their layout is visible in Figure 2.

```
public static boolean assignAction(ZKIComponent component, ZKIAction action)
{
    switch (action.getType())
    {
        ...
        case SELECT:
        {
            if (!(component instanceof ZKISelectableComponent))
                return false;

            ((ZKISelectableComponent) component).addActionListener(
            new ActionListener()
            {
                @Override
                public void actionPerformed(ActionEvent e)
               {
                    action.start();
                }
            });
            return true;
        }
        return false;

 ... ...
```

Code 9: ZKBasicActionHandler

6.2.4.2 ZKMainFrame

ZKMainFrame is a successor of ***ZKFrame***, adding the function panels management. It also contains a list of the panels and a few methods to manage them.

Likewise the functions has ***ZKIAbstractFunction*** as an ancestor interface, frames has the ***ZKIAbstractFrame*** interface. The reason is similar to the functions too – ***ZKMainFrame*** should not have **add(Component)** and similar methods, on the contrary of ***ZKFrame***, which should accept all types of components.

6.3 Components

6.3.1 Basic methods

All implemented components have several methods in common. Some of them are introduced in the following paragraphs.

6.3.1.1 assignAction

The **`assignAction(Action)`** method calls ***`ZKBasicActionHandler`***, which should bind a given action to related listener, as shown in Code 9.

6.3.1.2 getUnderlyingComponent

This method is important for backward compatibility with Swing. Every container component in the framework calls this method in final consequences, because this is the only way to connect all the components together.

6.3.1.3 getRealComponent

There are some composed components (see chapter 6.3.3) implemented where the main component is surrounded by ***`ZKPanel`*** or similar component. The **`getRealComponent()`** method always returns the main component.

6.3.1.4 setVisible

This method is in standard Swing too but it is overridden in all frameworks components because it handles the visible events, as described in 6.2.1.1.

```
public void setVisible(boolean visible)
{
    if (visible)
    {
        if (!beforeShow()) return;
    } else {
        if (!beforeHide()) return;
    }
    super.setVisible(visible);
    if (visible)
    {
        afterShow();
    } else afterHide();
}
```

Code 10: The setVisible method

In Code 10 there is the **`setVisible(boolean)`** method calling the events according to the description.

6.3.1.5 setConstSize

```
public void setConstSize(Dimension size)
{
    setSize(size);
    setPreferredSize(size);
    setMinimumSize(size);
    setMaximumSize(size);
}
```

Code 11: The setConstSize method

There is an implementation of the **`setConstSize(int,int)`** method in Code 11. The miscellaneous methods calls are important because of the fact that different layout managers reflect different method result. For example the ***`FlowLayout`*** takes the preferred size but the ***`GridLayout`*** does not.

6.3.1.6 setScrollable

Compared to other components ***`ZKPanel`*** includes one important method in addition. The method sets the panels content scrollable or static. The component extends the ***`JPanel`*** and it contains another ***`JPanel`*** as a content panel. Depending on **`setScrollable(boolean)`** parameter value, it puts the content panel into or outside from a ***`JScrollPane`***.

6.3.2 Simple components

Some components (***`ZKPanel`***, ***`ZKButton`***, ***`ZKLabel`***, ***`ZKCheckBox`***, ***`ZKList`***, ***`ZKObjectSpinner`***, ***`ZKProgressBar`***, ***`ZKRadioButton`***, ***`ZKTabbedPane`***, ***`ZKTable`***) are simply extended Swing components with a few new methods implemented (as described in 6.3.1) and therefore it is useless to describe each component to the depth.

A more sophisticated component is ***`ZKTable`***.

6.3.2.1 ZKTable

`ZKTable` extends the ***`JTable`*** and adds some important functionality. Beside the events, which all the components have, the ***`JTable`*** has more advanced management of data source and columns.

Columns

The columns management is quite confusing in ***`JTable`*** in standard Swing. The KIV framework has a class name ***`ZKTableColumn`***, which solves some common issues as a cell rendering or data sorting.

The programmer can easily set whether the column should have editable cells, or how to display the cells content. ***ZKTableColumn*** has a few successors:

1) ***ZKColoredTableColumn***
   - has a colored cells, where the color represents the value
   - an abstract class, the **getColor(String)** method has to be implemented
2) ***ZKFormattedTableColumn***
   - displays value according to defined format, the cell editor can have some ***Pattern*** set
3) ***ZKNumberTableColumn***
   - displays a number in a defined format
   - ***ZKSpinner*** is used as an editor
4) ***ZKCheckBoxTableColumn***
   - displays a colored or selectable cells (colored by default)
   - ***ZKCheckBox*** is used as an editor
5) ***ZKDateTableColumn***
   - displays date in defined format
   - ***ZKDateField*** is used as an editor

Data source

Standard Swing uses the ***TableModel*** interface implementations such as ***DefaultTableModel*** as data source. Working with ***DefaultTableModel*** is easy, but it is suitable for simple data display only.

The KIV framework contains the ***ZKITableDataSource*** interface which serves as a replacement of ***TableModel*** (which ***ZKITableDataSource*** extends). There is also the ***ZKIWritableTableDataSource*** interface (extends ***ZKITableDataSource***), which is used for sources which supports writing and not only reading. Currently there is only one usage of these interfaces. It is ***ZKDefaultTableDataSource*** which implements the ***ZKIWritableTableDataSource*** interface. It has a very similar functionality as ***DefaultTableModel*** has, adding some events and offering functionality to sort or set data.

The main reason for implementation of these classes is the data binding. Another data source e.g. that connected to a database can be implemented quite easily, as well as one connected to any other physical data source like XML.

6.3.3 Composed components

There are some composed components in the framework: ***ZKComboBox***, ***ZKFormattedTextField***, ***ZKSpinner***, ***ZKPasswordField***, ***ZKRadioButtonGroup***, ***ZKTextField, ZKTextPane*** and ***ZKList***.

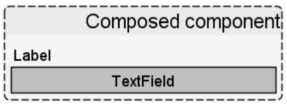


Figure 9: The composed component layout

All the components listed above are composed together with a title label and inserted into ***ZKPanel***, as shown in Figure 9. They support standard methods (like they were simple components) but also have methods for working with title (set/get/show).

6.3.3.1 ZKList

ZKList is implemented very similarly as ***ZKTable***, except there are no cell renderers implemented. ***ZKList*** is in fact ***ZKTable*** with only one column, so there is no neccesity to solve the columns management as for ***ZKTable***. On the other hand the data sources can be used in very similar way (except the genericity can be used).

Data sources

The ***ListModel*** class is used in standard Swing. The most commonly used implementation is ***DefaultListModel***, which is created automatically in ***JList***, when we do not provide our own instance of ***ListModel***.

Similarly to ***ZKTableDataSource*** there were ***ZKIListDataSource*** and ***ZKIWritableListDataSource*** designed, with ***ZKDefaultListDataSource*** implementation. The ***ZKDefaultListDataSource*** class extends the ***DefaultListModel*** and adds some functionality to it. ***ZKList*** has methods **bindDataSource** and **unbindDataSource** for working with the source binding.

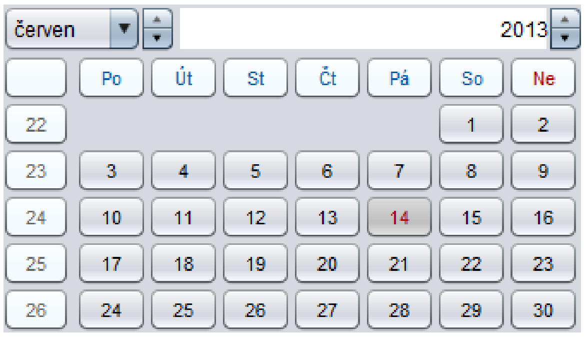


Figure 10: ZKCalendar component

6.3.1 New components

Except the template layout components described in 6.2.3 and 6.2.4 and the complex composed components, there are 8 new components, which will be described in the following paragraphs.

6.3.1.1 ZKCalendar

`ZKCalendar` is a component derived from ***`JCalendar`*** (8). It can be classified as a composed component, because it is surrounded by ***`ZKPanel`*** and there was a title added.

`ZKCalendar` makes it possible to select date. It looks like an ordinary datepicker, as it is visible in Figure 10.

6.3.1.2 ZKCanvas

```java
public void drawLine(int x1, int y1, int x2, int y2)
{
    addAction(new Class[]{int.class, int.class, int.class, int.class},
    new Object[]{x1, y1, x2, y2});
}
```

Code 12: ZKCanvas drawing method example

The programmer in Java often faces a problem that he/she wants to draw on some component surface (typically ***`JPanel`***). Unfortunately unlike e.g. in Delphi, Java does not support direct drawing. The programmer has to override the **`paintComponent(Graphics)`** method. In the body of the method he/she can use the ***`Graphics`*** instance and draw whatever he/she wants, but this is suitable only for static drawing. There is a problem when he/she wants to draw something dependent e.g. on users behavior.

```java
StackTraceElement elem = Thread.currentThread().getStackTrace()[2];
Method method = Graphics.class.getMethod(elem.getMethodName(), types);
actions.add(new ZKCanvasAction(method, values));
```

Code 13: ZKCanvasAction creation

```java
for (ZKCanvasAction action : actions)
{
    Method method = action.getMethod();
    try
    {
        method.invoke(g, action.getParams());
    } catch (Exception e)
    {
        throw new RuntimeException(e);
    }
}
```

Code 14: ZKCanvas drawing actions invocation

The KIV framework offers a new component ***ZKCanvas***, which supports direct drawing, like components in other programming languages. The component itself contains an ordinary ***ZKPanel***, but there are standard drawing methods (from the ***Graphics*** class) implemented.

While calling any of these methods (like in Code 12), a new ***ZKCanvasAction*** instance is added to the list of actions, which ***ZKCanvas*** contains.

There is a passage of **addAction(Class[],Object[])** method in Code 13. ***ZKCanvas*** uses a method reflection for the drawing. There is a method instance created from given parameters (***Class*** array) and stored in the ***ZKCanvasAction*** instance together with the parameters values.

While the JVM calls the **paintComponent(Graphics)** method, the list is gone through and each stored action is invoked; it leads into direct drawing effect. The invocation process is shown in Code 14.

6.3.1.3 ZKDateField

ZKDateField is a combination of ***ZKFormattedField*** and ***ZKCalendar***. By default, the field and the button are displayed. The date can be written into the field manually or it can be selected from the floating calendar, which appears after the button click.

6.3.1.4 ZKForm

It is often required to arrange components in a form. A simple example of the form is a login dialog, which contains only two fields and one button. The login dialog can be created quite easily, but more complex forms are very difficult to be build, because Java does not offer anything like a form layout. This is the reason why many developers have implemented such a thing, and also the KIV framework has a simple form layout implemented.

It is implemented like a component named ***ZKForm***. ***ZKForm*** extends ***ZKPanel*** but has different methods implemented, because only the components implementing ***ZKIInputComponent*** can be inserted. The layout is realized by ***GridBagLayout*** and all the rows of the form are enclosed in the ***ZKFormRow*** instance.

6.3.1.5 ZKIcon

In standard Swing there is a problem with pictures handling. The Icon interface (and its most common implementation ***ImageIcon***) has to be inserted into ***JLabel*** (or some other suitable component) to be displayed on the GUI form. Unfortunately, there is often a requirement to use the icon in drag-and-drop process, which Java also does not support. The drag-and-drop is closely related to a copy-paste mechanism, which is also problematic.

The KIV framework has a component ***ZKIcon*** which solves the problems described in the previous paragraph. ***ZKIcon*** extends ***JLabel*** but also implements the ***Icon*** and the ***Transferable*** interfaces. ***JLabel*** has to be used instead of ***ZKLabel*** because access to the standard labels icon is needed and ***ZKLabel*** offers access only through the ***ZKIcon*** class, which would lead to infinite recursion. The ***Transferable*** interface serves for the copy-paste and drag-and-drop services.

Because ***ZKIcon*** is extended ***JLabel***, it can be used as a component directly, without putting it into any other component. On the other hand, all other components (implementing ***ZKIIconedComponent***) accept the ***ZKIcon*** instance as **setIcon(ZKIcon)** method parameter.

6.3.1.6 ZKStatusBar

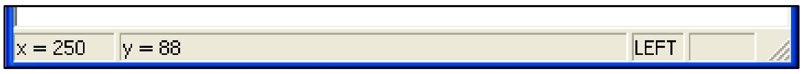


Figure 11: The status bar example (12)

The component ***ZKStatusBar*** should serve as a footer panel with a few cells displaying pieces of information like current time, status etc. as visible in Figure 11.

ZKStatusBar extends ***ZKPanel*** too and have method to set/get current cells count and their content.

6.3.1.7 ZKTab

The ***ZKTab*** class is a simple encapsulation of a single tab, prepared for inserting into ***ZKTabbedPane***. The class extends ***ZKPanel*** and contains a title, tooltip and the ***ZKIcon*** instance. While inserting ***ZKTab*** into ***ZKTabbedPane***, the pane extracts all necessary data from ***ZKTab*** object and display the tab correctly without any following adjustments.

6.3.1.8 ZKWrapLayout

ZKWrapLayout is a simple enhancement of standard Swing ***FlowLayout***. It is adopted from 3rd party library (13) and it makes components wrapping possible. ***FlowLayout*** straightens inserted components into a single row, not wrapping an invisible component to a new row, while some component ran out from the visible area of the window. ***ZKWrapLayout*** can handle this situation and it also reacts to the parent panel resizing and handles the components position.

6.4 Drag-and-drop

Because the drag-and-drop is implemented for most components in Java Swing, it had to be implemented D'n'D only for ***ZKLabel***, which also includes the ***ZKIcon*** class. There are two interfaces implemented ***ZKIDragSourceComponent*** for components, which can be source of the D'n'D, and ***ZKIDropTargetComponent*** used by all components, which can be a D'n'D target.

The ***ZKIDragSourceComponent*** interface includes methods for manual managing ***DragSourceListener*** and events **beforeDragStart(…)** and **afterDragStart(…)**both of them having parameter with the ***Transferable*** object.

ZKIDropTargetComponent is very similar: it contains methods for managing ***DropTargetListener*** and events **beforeDrop(…)** and **afterDrop(…)**. The **beforeDrop(…)** method returns FALSE by default, so every drop above this component is rejected. The programmer has to handle the ***Transferable*** object given as parameter manually and decide whether to accept or reject the drop.

There is a problem with the D'n'D for ***ZKTable*** and ***ZKList***. It cannot be predicted what data types will be used in these components and ***ZKIDragSourceComponent*** cannot be implemented because of that. If the D'n'D is requirement for these components, the programmer has to extend the component and implement the interface by himself. On the other hand, both components can serve as D'n'D target so they implement the ***ZKIDropTargetComponent***.

6.5 Action binding

Action binding was described in chapter 5.4. All the designed actions were implemented.

There are two interfaces ***ZKIAction*** and ***ZKIInputAction*** (which extends the first one). The second one is used by all actions which has to deal with a component input, the first one is used in every non-input actions.

```
final ZKIProgressBar progressBar = new ZKProgressBar();
progressBar.assignAction(new ZKClickAction()
{
    @Override
    public void action()
    {
        progressBar.increaseValue(10);
    }
});
```

Code 15: The ZKClickAction usage example

All the specific action classes (e.g. ***ZKClickAction***) are abstract classes with unimplemented **`action()`** method. The programmer assigns the action to related object by calling **`assignAction(ZKIAction)`** method, as shown in Code 15.

The ***ZKBasicActionHandler*** class secures the action binding on lower level, because it binds the specific actions to Swing listeners. ***ZKBasicActionHandler*** code was shown in Code 9.

6.6 Data binding

The sense of data binding was described in 5.5. The implementation can be divided into two parts – for simple and for complex components.

6.6.1 Binding for simple components

There were ***ZKIBindableComponent*** and ***ZKIBindableSourceComponent*** interfaces introduced in section 5.2.2.

Every component implementing ***ZKIBindableComponent*** can be a target of binding. It means that, for example, its title can be changed automatically based on other components text. ***ZKIBindableSourceComponent*** is implemented by components, which can be source components. Other components can change their titles according to these ones.

This binding should be called a properties binding. The properties binding is handled by a few classes from the ***binding*** package. The most important class is ***ZKBindingManager***. Its main method **`bind(ZKIBindableSourceComponent, ZKIBindableComponent, String, String, ZKBindingStrategy)`** adds a new ***ZKBindingRelation*** to the relations list.

The binding uses an observer-observable pattern. ***ZKBindingManager*** implements the ***Observer*** interface, because it catches update events from binding clients. On the contrary ***ZKBindingClient*** extends the ***Observable*** class and generates the update events based on property listeners of related component. Every bindable component includes instance of this class.

The **`update(Observable, Object)`** method in the observer is invoked whenever some property of bound object is changed, even if the actual property is not watched. The method searches a local database for related ***ZKBindingRelation***. If the relation is found, the setter method for the property is gained and invoked with a given value.

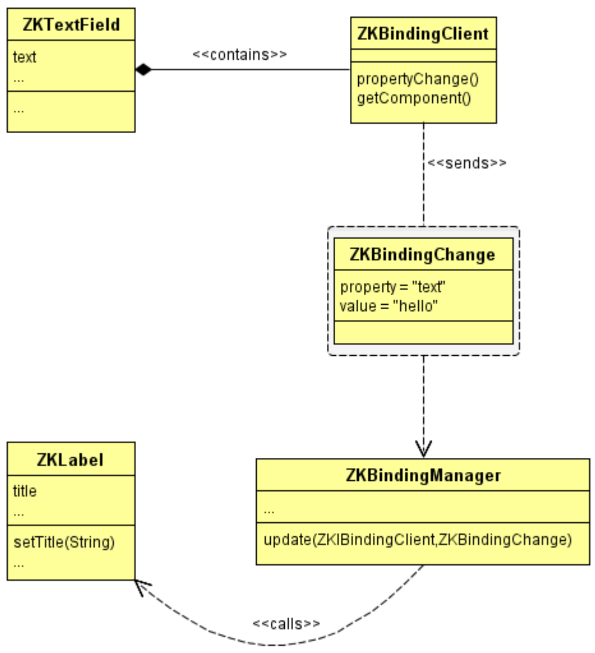


Figure 12: The binding diagram

There is a reflection used for the setter method invocation, likewise in ***ZKCanvas*** in chapter 6.3.1.2. The existence of the setter method is checked out in the **bind(…)** method for the first time. When the setter method does not exist, the **bind(…)** method returns **FALSE**.

There is a binding diagram example in Figure 12, showing how the binding sends the data.

6.6.2 Binding for complex components

Only three components from the framework belong to this group.

6.6.2.1 ZKTable data binding

There were ***ZKTable*** data sources described in chapter 6.3.2.1. Except of the source binding the table offers some properties binding (e.g. title) too, alike almost all other components.

6.6.2.2 ZKList data binding

ZKList offers the same data binding principle as ***ZKTable***. The data sources for ***ZKList*** were introduced in 6.3.3.1.

6.7 Tooltip formatter

The tooltip formatter package includes the ***ZKTooltipFormatter*** class and also some tags classes. ***ZKTooltipFormatter*** includes methods for global tooltips setting – background, border, delay etc. The tooltip formatter tool is independent to the rest of the package and can be used for general HTML code creation.

All the implemented tag classes implement the ***ZKITooltipTag*** interface. There is also another interface ***ZKITooltipContentTag***, which extends ***ZKITooltipTag***. Most of the implemented tags (e.g. ***ZKRawTag***, ***ZKTextTag***, ***ZKListTag***) implement ***ZKITooltipTagContent***, but ***ZKBreakTag*** and a few others do not have any content and implement the ***ZKITooltipTag*** interface only.

All the implemented tags are:

1) Break <BR>
2) Horizontal line <HR>
3) Text <SPAN>
4) List <UL>
5) Numbered list <OL>

The ***ZKGeneralTag*** class constructor has the name parameter, which makes possible to use it as an arbitrary tag. There is tag class ***ZKMainTag***, which has to be used as a parent tag of whole tooltip content. It can be created directly by constructor or get by the static **getHtmlTooltip()** method in ***ZKTooltipFormatter***.

6.8 Factories

6.8.1 Dialog factory

ZKDialogFactory currently has only five methods, which serve for showing an ordinary message or error message.

6.8.2 Icon factory

ZKIconFactory has a few methods for most common icons, as closer described in 5.8.3. All the icons are stored in the JAR with the framework and ***ZKIconFactory*** contains a method, which loads them into the ***ZKIcon*** instances.

6.8.3 Border factory

There are only few methods in the ***ZKBorderFactory*** class. This is because it should only supplement ***BorderFactory*** in standard Swing. All currently implemented methods are:

- **roundedBorder(Color)**
- **plainBorder(Color)**
- **titledBorder(Color)**

… and their overloaded variants.

```
public static CompoundBorder titledBorder(String title, Color color)
{
    return BorderFactory.createCompoundBorder(new TitledBorder(
        new LineBorder(color, 1, true), title), new EmptyBorder(1, 1, 1,
1));
}
```

Code 16: The ZKBorderFactory method example

As you can see in Code 16, all methods in ***ZKBorderFactory*** combine few methods from ***BorderFactory*** (e.g. add padding to standard ***LineBorder***).

6.8.4 Button factory

ZKButonFactory works similarly to ***ZKIconFactory***. All implemented methods such as **getOKButton(String)** takes the ***String*** as a parameter for the button title. The buttons icons are got from ***ZKIconFactory***.

6.8.5 Font factory

As declared above in chapter 5.8.6, ***ZKFontFactory*** makes an effort to provide the programmer more fonts than the standard set. All new fonts are included in JAR file with the framework, the same as the icons are.

```
ClassLoader cl = new ZKFontFactory().getClass().getClassLoader();

URL url = cl.getResource("resources/fonts/" + name.toLowerCase() + ".ttf");

if (url == null) throw
    new FileNotFoundException("Cannot find file with font '" + name + "'!");

GraphicsEnvironment ge = GraphicsEnvironment.getLocalGraphicsEnvironment();
return ge.registerFont(Font.createFont(Font.TRUETYPE_FONT,
url.openStream()));
```

Code 17: The ZKFontFactory.registerFont(…) method

ZKFontFactory contains an important method **registerFont(String)** which loads a font with specified name. The method extract is visible in Code 17.

This is the list of fonts, which are included in the framework:

- Am Sans light
- Eurofurence light
- Share regular
- Times sans serif
- Ritalin
- Gunny Handwriting
- GFS Bodoni
- GFS Artemisia
- Flatform
- TeXGyreCursor

All these fonts have been downloaded from http://www.ceskefonty.cz/ and are released under a royalty free license[9]. They all support Czech characters.

The ***ZKFontFramework*** contains ***String*** constants for every font mentioned above and also for standard fonts like Arial, Verdana or Times New Roman. There are also constants for font size (such as **SIZE_BIGGER**) and for a few colors.

6.9 Utilities

6.9.1 Memory file system – MFS

6.9.1.1 Basis of the filesystem

The MFS is based on classes ***FileSystem***, ***FileSystemProvider***, ***Path*** and a few others from ***java.nio*** package. These classes were introduced in JDK 7, together with the example (but fully functional) implementation of ZIP/JAR filesystem[10]. This allows the developer to implement a new FS with maximum compatibility with the current existing ones, so the operations can be processed easily.

All common operations supported are in MFS. It is fully compatible with any other filesystem implementing ***java.nio.file.Filesystem*** and related classes. It was advantageous to use abstract classes rather than interfaces while implementing ***ZKMemoryEntry*** and its successors – ***ZKMemoryFile*** and ***ZKMemoryDirectory***. ***ZKMemoryDirectory*** contains the ***HashMap*** of its entries. There is no limit of entries

[9] It means it can be used for everything including commercial usage.
[10] The ZIP FS implementation is basis of the ZIP functions of this framework – see ZIP files.

in one directory, so it should not be a problem while copying from any existing filesystem. The maximum level of entry is not set too.

ZKMemoryFile contains the ***ZKMemoryFileContent*** instance, which is holding the current file content. The data are stored in the ***DirectByteBuffer*** instance with default size of 5MB. When the buffer size is not satisfying, its size is doubled. It can lead to a pointless memory usage but it is an action against frequent resizing too. ***ZKMemoryFile*** supports basic ***InputStream***/***OutputStream***, which simply goes through the content array and reads/writes the data.

6.9.1.2 Basic functions

The FS allows the programmer to do the classical CRUD operations with all files stored in the memory. Files can be stored in directories; the programmer can get their input/outputstream, size or CRC. He/she can go through the file tree too.

While creating a new MFS instance, the programmer can determine max size of the FS. The default size is 50MB. The size is maximum size and it will not be allocated immediately after filesystem creation, but progressively as needed. ***IOException*** is thrown when setting new filesystem size and there is not enough space in the RAM.

```
Socket socket;
....
//inputstream from the socket
InputStream in = socket.getInputStream();
//outputstream to file
BufferedOutputStream out = new
BufferedOutputStream(Files.newOutputStream(mFile));

/* copy: */
byte[] buffer = new byte[1024];
int numRead;
while ((numRead = in.read(buffer)) != -1)
{
    out.write(buffer, 0, numRead);
}
out.close;
in.close;
```

Code 18: The ZKMemoryFileSystem example 2

In Code 18 the program receives data from socket (typically some Internet connection) and saves them into the memory file. Afterwards the programmer can also get the CRC and validate it against the source.

```
Map<String, Object> env = new HashMap<>();
env.put("size", 100 * 1024 * 1024);

//create new filesystem, name 'test'
FileSystem fs = FileSystems.newFileSystem(URI.create("mem:test"), env);
```

Code 19: The ZKMemoryFileSystem example 3

In Code 19 is set the maximum size of the filesystem to 100MB.

6.9.1.3 Pitfalls

1) The programmer should not forget about once he/she call the **`fs.close()`** method, all contained data are erased and the operation cannot be undone, because allocated memory is released.
2) The programmer should not forget to set max size of the filesystem. He/she can change the size additionally, but he/she cannot exceed the current max size.
3) When the programmer does not use the MFS from included JAR file, the **mem** scheme is unknown. In this case, he/she has to use method **`newFileSystem()`** directly from the ***`ZKMemoryFileSystemProvider`*** class.

6.9.2 ZIP files

6.9.2.1 Basis of the system

Like the Memory file system – MFS, the ZIP tool stands on classes ***`FileSystem`***, ***`FileSystemProvider`***, ***`Path`*** from ***`java.nio`*** package. Oracle released an implementation of the ZIP/JAR filesystem, so it can be used freely now. Of course the programmer has an option to use this system directly, without this framework. On the other hand, the FW gives him an option to handle the ZIP files very simply and it also solves some difficulties.

The ZKZipFile class is an instance of ZIP file. It provides common operations for working with the file.

```
File file = randomFile();
ZKZipFile zip = ZKZipFile.create(file);
File randomFile = randomFile();
String pass = ZKCrypt.getHash(new Random().nextLong() + "");

zip.enableEncrypting(pass);
```

Code 20: The ZKZipFile example 2

In Code 20 there is shown creation of encrypted ZIP archive. The **`ZKZipFile.createEncrypted(File file, String password, boolean overwrite)`** method can be used for this too.

6.9.2.2 Pitfalls

1) The programmer should not forget to close the ZIP after his/her work, or the changes will not be written to a disk. In special case, the ZIP can be even damaged. The closed ZIP can be reopened later.
2) Encrypted ZIP files can be reopened only by this FW, because it uses a specific encryption method. On the other hand, non-encrypted ZIP archives are fully compatible with standard ZIP format (it can be opened by WinZip and other similar programs).

The programmer should notify difference between **`create()`** and **`open()`** methods: **`create()`** will create a **new** archive in any case; on the contrary **`open(filename, true)`** will create a new archive only if it does not exist. The **`open()`** method without ***createIfNecessary*** parameter will fail, when the archive does not exist. The **`openOrCreate()`** method can also be used.

6.9.3 Encryption

All functions use classes from ***`javax.crypto`*** and ***`java.security`*** packages. These packages give the programmer an option to use different types of ciphers, but usage is very difficult and requires knowledge of the issue. The framework makes it possible to use these functions without any knowledge, only by calling such a method.

```
inFile = Files.newInputStream(file);
outFile = new FileOutputStream(tempFile);

byte[] salt = Arrays.copyOf(encodeBase64(password).getBytes(), 8);

SecretKeyFactory factory =
SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
KeySpec spec = new PBEKeySpec(password.toCharArray(), salt, 65536,
STRENGTH);
SecretKey tmp = factory.generateSecret(spec);
SecretKey secret = new SecretKeySpec(tmp.getEncoded(), ALGORITHM);

Cipher cipher = Cipher.getInstance(ALGORITHM);
cipher.init(cryptMode, secret);

byte[] input = new byte[64];
int bytesRead;
while ((bytesRead = inFile.read(input)) != -1){
    byte[] output = cipher.update(input, 0, bytesRead);
    if (output != null) outFile.write(output);
}

byte[] output = cipher.doFinal();
if (output != null) outFile.write(output);
```

Code 21: The encrypting algorithm extract

There is the encrypting algorithm shown in Code 21. This algorithm is used for encrypting/decrypting files – the way depends on value of ***cryptMode*** parameter. This is an example of so-called stream ciphering, because it can encrypt/decrypt an InputStream/OutputStream directly, which is usable e.g. for socket communication.

6.9.3.1 Basic functions

The programmer can encrypt/decrypt files and strings.

There are also methods for creating hash of ***String*** (a one-way cipher) and for Base64 scheme encoding in ***ZKCrypt***. The encrypting was also implemented for ZIP files, so the programmer does not have to encrypt/decrypt his/her ZIP archive separately. Of course, he/she can combine ***ZKCrypt*** with others parts of the FW – e.g. he/she can decrypt some file directly to the memory file and stay safe.

6.9.3.2 Text and files encryption method

The encryption/decryption uses by default the AES-128 algorithm. Unfortunately, the standard Java release does not allow using stronger keys for the AES[11]. If the programmers computer has the JCE (Java cryptography extension) unlimited strength configured, he/she can use the AES-256 too.

All possible standard options are: AES-128, DES-64, 3DES-192. The AES method is recommended.

6.9.3.3 Base64

The Base64 can be used when the programmer has an array with raw data and he/she needs to use it as a ***String*** (sent through a text oriented socket etc.) – all data are converted to basic ASCII (7bit encoding). Retrieved ***String*** can be converted back to the original ***byte[]***, of course.

```
Socket socket;
BufferedWriter writer =
new BufferedWriter(new OutputStreamWriter(socket.getOutputStream()));
writer.write(ZKCrypt.encryptString("very secret text!", "the password"));
...
BufferedReader reader =
new BufferedReader(new InputStreamReader(socket.getInputStream()));
String message = ZKCrypt.decryptString(reader.readLine(), "the password");
```

Code 22: The ZKCrypt example 2

The example in Code 22 (very simplified compared to real usage) shows how to send an encrypted message through text-oriented channel.

[11] Standard Java restrictions are very strong. Stronger keys can be enabled only by downloading and installing a special JAR file from Oracle to every single machine, where the encrypting should start to work. There is no way how to distribute it with the framework.

6.9.3.4 Pitfalls

1) The password must be >=8 chars, to secure power of the encryption.
2) The programmer can use even more algorithms, but its functionality depends on his/her JDK distribution (JCE configuration).

6.9.4 File utilities

6.9.4.1 Basic functions

The file utilities class currently contains only two methods:

- ✓ CRC calculation of files (including memory files).
- ✓ Getting file content (including memory files) as a String.

6.9.5 Localization

6.9.5.1 Basis of the system

The ***`ZKLocalization`*** class uses standard Java ***`ResourceBundle`*** class.

6.9.5.2 Basic functions

The ***`ZKLocalization`*** class contains the following methods:

- ✓ **`getString(String key, Object… params)`** – this method calls the **`String.format()`** method internally
- ✓ **`getCurrentLocale()`** – gets current set Locale instance
- ✓ **`setCurrentLocale(Locale locale)`** – sets current Locale and loads related language file

The class also contains constants for the following languages: ***CZECH***, ***ENGLISH***, ***GERMAN*** and ***SLOVAK***.

6.9.6 Download

The download tool was described in chapter 5.9.5. The buffer size is set to 1kB.

6.9.6.1 Basic function

The main function of this class is very simple. Main advantage of this class is in 4 events methods, which are invocated in certain situations:

1) **`onStart(bytesTotal)`** – gives a size of downloaded file. There is a small catch in here – see 6.9.6.2.
2) **`onProgress(bytesDownloaded, percent)`** - invoked when some progress (1kB) is done.
3) **`onFinish(bytesDownloaded)`** - invoked when the file is already downloaded.
4) **`onFail(exception, bytesDownloaded)`** – when something went wrong.

```
ZKFileDownloader downloader = new ZKFileDownloader(new URL(addr)){
    @Override
    protected void onStart(long bytesTotal){
    }

    @Override
    protected void onFinish(long bytesDownloaded){
System.out.println("AVG: " +
ZKFileDownloader.formatSpeed(this.getAverageSpeed()) + " MAX: " +
ZKFileDownloader.formatSpeed(this.getMaxSpeed()));
    }

    @Override
    protected void onProgress(long bytesDownloaded, double percent){
    }

    @Override
    protected void onFail(Exception exception, long bytesDownloaded){
    }
};
downloader.start();
```

Code 23: The ZKFileDownloader example

Code 23 is an example of inline usage. The programmer should use it this way exceptionally, only when he/she has to download a single file.

```
new ZKFileDownloader(new URL(addr))
{
   ... ...
}.start();
```

Code 24: The ZKFileDownloader inline usage

Of course he/she does not have to keep the instance of the downloader and he/she can call the **`start()`** method directly, as shown in Code 24.

The **`formatSpeed()`** and **`formatSize()`** methods (used in **`onFinish())`** should help the programmer to format the received values properly.

6.9.6.2 Pitfalls

1) Unfortunately, the server does not have to provide the size of downloading file. Rarely the programmer can get int ***-1*** in events **`onStart()`** (as *bytesTotal*) and **`onProgress()`** (as *percent*).
2) The programmer should remember, that if he/she throws an exception from any of events method, the exception is distributed through the **`start()`** method and it will cause failure of download process.

6.9.7 EXIF

6.9.7.1 Basis of the system

Because the EXIF reading is a hard discipline, the framework uses a third party library (14) for this purpose. The original library can read EXIF from JPEGs, TIFFs and also some RAW formats. It contains support for all widely used camera brands.

The FW allows using only JPEGs but the programmer can use also the original library directly, if necessary.

6.9.7.2 Basic functions

The first instance of the ***ZKExif*** class is created and then the programmer can use getters (**getModel(), getExposureTime()**, **getAperture()**, **getFocalLength()**, etc.). He/she can also read the metadata by a universal getter **getTag()** with public constants such as **TAG_EXPOSURE_TIME**, **TAG_APERTURE** etc.

6.9.7.3 Pitfalls

1) The ***ZKExif*** class was tested with photos from Canon, Nikon, Sony, Pentax, Samsung, Casio, Fujifilm, Kodak and Panasonic and also with photos from smartphones from Apple, Sony Ericsson and LG. Unfortunately, not all of manufacturers put the EXIF to its JPEGs, so (especially for smartphones) the program cannot rely on existence of any EXIF data in the photo.
2) The programmer can read the EXIF from JPEG, which has no EXIF data stored. In that case every getter will return **null** value and **toString()** method will return only the name of the file.

6.9.8 JSON converter

6.9.8.1 Basic functions

The class can load the properties file and save it into JSON file or back to properties file (but alphabetically sorted).

```
Path source = new File("trunk/testdata/messages-en.properties").toPath();
Path dest = randomFile().toPath();

ZKJsonConverter converter = new ZKJsonConverter(source, dest);
converter.execute();
```

Code 25: The ZKJsonConverter example 1

Code 25 converts the properties file to the JSON format. The program can also use **saveAsProperties()** and **saveAsJson()** methods, but the **load()** method has to be called first in that case!

```
<property name="srcFilename" value="messages_en.properties" />
<property name="destFilename" value="messages-en.json" />

<taskdef name="convert"
classname="cz.zcu.kiv.framework.utils.jsonconverter.ZKJsonConverter"
classpath="lib/fw_0.1.jar" />
<target name="convertToJSON"
<convert srcfile="${srcFilename" destfile="${destFilename}" saveasjson="true
</target>
```

Code 26: The ZKJsonConverter example 2

Code 26 is an example of use in ANT script.

6.9.9 Sound factory

6.9.9.1 Basis of the system

The ***ZKSoundFactory*** class is based on 3rd party library Javazoom JLayer library (15). The class uses a small subset of the library abilities and can only start playing a selected sound.

The example of usage is very simple.

```
ZKSoundFactory.alarm();
```

Code 27: The ZKSoundFactory example

Code 27 shows how to play an "ALARM" sound. All available methods are non-blocking so the program execution continues immediately after the playback starts.

6.9.9.2 Pitfalls

1) The JLayer library (15) is quite old. Although the tests confirmed its functionality, the malfunction with some special modern MP3 cannot be fully eliminated.
2) ***ZKSoundFactory*** does not offer playback control or callbacks. It just simply plays the file.

6.9.10 Czech comparator

```
new Locale("cs", "CZ")
```

Code 28: The Czech Locale creation

The ***ZKCzechComparator*** class uses **Locale** created like in Code 28.

```
TreeSet<String> set = new TreeSet<>(ZKCzechComparator.getInstance());
```

Code 29: The ZKCzechComparator example

By Code 29 the programmer can set the comparator of new ***TreeSet*** to a Czech one. All inserted strings are then sorted properly.

6.9.11 Console

6.9.11.1 Basis of the system

The ***ZKConsole*** class is based on direct work with streams (***InputStream*** and ***PrintStream***). It uses the ***Scanner*** and ***Console*** classes for some operations too.

6.9.11.2 ZKConsole

The class offers overloaded methods for reading and printing to console, such as **print()**, **println()**, **read()**, **readLine().** It contains methods for users answering, such as **waitFor()** and **waitForEnterKey()** too.

The class is prepared for the program arguments handling. The programmer can use the **setProgramArgs()** method if he/she needs to use the program arguments in other place then the **main()** method. The **getProgramArgs()** will return **List<ZKCommand>.**

6.9.11.3 ZKCommandParser

This class offers a command parsing. It’s useful both for both program arguments parsing and console input parsing.

```
String line = "comm1 comm2 \"comm3_1 comm3_2\" comm4    comm5 \"comm6_1 comm6_2\" comm7 comm8\" comm9   ";

ZKCommandParser parser = new ZKCommandParser(System.in);

List<String> res = new ArrayList<>();
for (ZKCommand command : parser.parseLine(line))
{
    res.add(command.getText());
}
```

Code 30: The ZKCommandParser example

Code 30 parses a given line into a list of commands.

6.9.11.4 Pitfalls

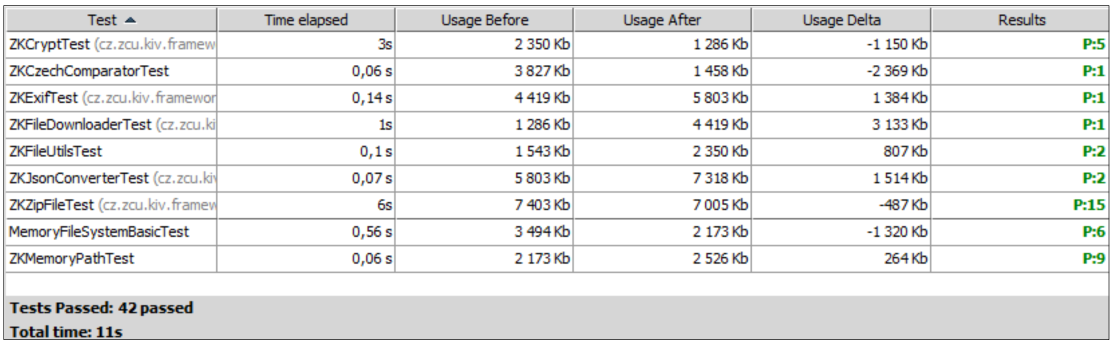
The **readPassword()** method unfortunately still depends on the ***Console*** class. When the console is unavailable, the method throws ***RuntimeException***. This is caused by Java has not an option to read chars from console without displaying them (except the **readPassword()** method in ***Console***).

7 Project testing

7.1 Utilities testing

Utilities' testing was described in 5.10.2.

There were 42 tests methods implemented for the utilities package. The test code coverage of package is 72% of classes and 69% of code lines.

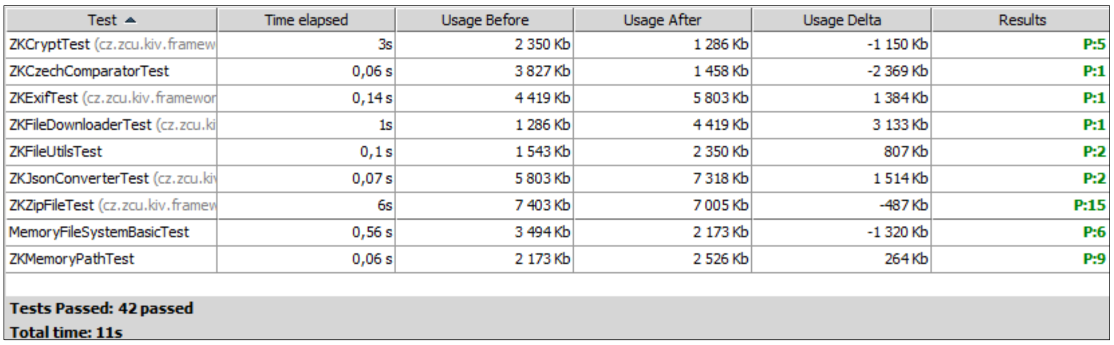


Test ▲	Time elapsed	Usage Before	Usage After	Usage Delta	Results
ZKCryptTest (cz.zcu.kiv.framew	3s	2 350 Kb	1 286 Kb	-1 150 Kb	P:5
ZKCzechComparatorTest	0,06 s	3 827 Kb	1 458 Kb	-2 369 Kb	P:1
ZKExifTest (cz.zcu.kiv.framewor	0,14 s	4 419 Kb	5 803 Kb	1 384 Kb	P:1
ZKFileDownloaderTest (cz.zcu.ki	1s	1 286 Kb	4 419 Kb	3 133 Kb	P:1
ZKFileUtilsTest	0,1 s	1 543 Kb	2 350 Kb	807 Kb	P:2
ZKJsonConverterTest (cz.zcu.ki	0,07 s	5 803 Kb	7 318 Kb	1 514 Kb	P:2
ZKZipFileTest (cz.zcu.kiv.framew	6s	7 403 Kb	7 005 Kb	-487 Kb	P:15
MemoryFileSystemBasicTest	0,56 s	3 494 Kb	2 173 Kb	-1 320 Kb	P:6
ZKMemoryPathTest	0,06 s	2 173 Kb	2 526 Kb	264 Kb	P:9

Tests Passed: 42 passed
Total time: 11s

Figure 13: Utilities tests result

All tests passed successfully as shown in



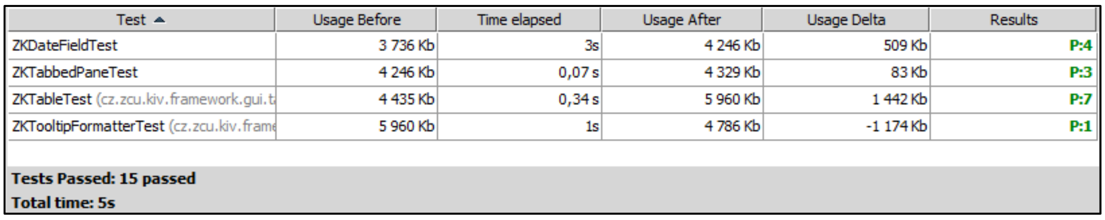
Test ▲	Time elapsed	Usage Before	Usage After	Usage Delta	Results
ZKCryptTest (cz.zcu.kiv.framew	3s	2 350 Kb	1 286 Kb	-1 150 Kb	P:5
ZKCzechComparatorTest	0,06 s	3 827 Kb	1 458 Kb	-2 369 Kb	P:1
ZKExifTest (cz.zcu.kiv.framewor	0,14 s	4 419 Kb	5 803 Kb	1 384 Kb	P:1
ZKFileDownloaderTest (cz.zcu.ki	1s	1 286 Kb	4 419 Kb	3 133 Kb	P:1
ZKFileUtilsTest	0,1 s	1 543 Kb	2 350 Kb	807 Kb	P:2
ZKJsonConverterTest (cz.zcu.ki	0,07 s	5 803 Kb	7 318 Kb	1 514 Kb	P:2
ZKZipFileTest (cz.zcu.kiv.framew	6s	7 403 Kb	7 005 Kb	-487 Kb	P:15
MemoryFileSystemBasicTest	0,56 s	3 494 Kb	2 173 Kb	-1 320 Kb	P:6
ZKMemoryPathTest	0,06 s	2 173 Kb	2 526 Kb	264 Kb	P:9

Tests Passed: 42 passed
Total time: 11s

Figure 13. The table columns contains name of the test, time from start to an end of the test, memory usage during the test and its result.

7.2 GUI automatic testing

Automatic testing of GUI was described in 5.10.1.1.



Test ▲	Usage Before	Time elapsed	Usage After	Usage Delta	Results
ZKDateFieldTest	3 736 Kb	3s	4 246 Kb	509 Kb	P:4
ZKTabbedPaneTest	4 246 Kb	0,07 s	4 329 Kb	83 Kb	P:3
ZKTableTest (cz.zcu.kiv.framework.gui.t	4 435 Kb	0,34 s	5 960 Kb	1 442 Kb	P:7
ZKTooltipFormatterTest (cz.zcu.kiv.fram	5 960 Kb	1s	4 786 Kb	-1 174 Kb	P:1

Tests Passed: 15 passed
Total time: 5s

Figure 14: GUI unit tests result

There were implemented tests as described in 5.10.2. All test passed successfully as shown in Figure 14.

7.3 GUI manual testing

Most of the GUI was tested manually, as described in 5.10.1.2. A demo application was implemented and all created components were used. The application was able to simulate real usage of application template and components.

The manual tests were performed by author of this thesis and two other people. Each component has defined functionality and the testers were checking defined points. All tests were successful.

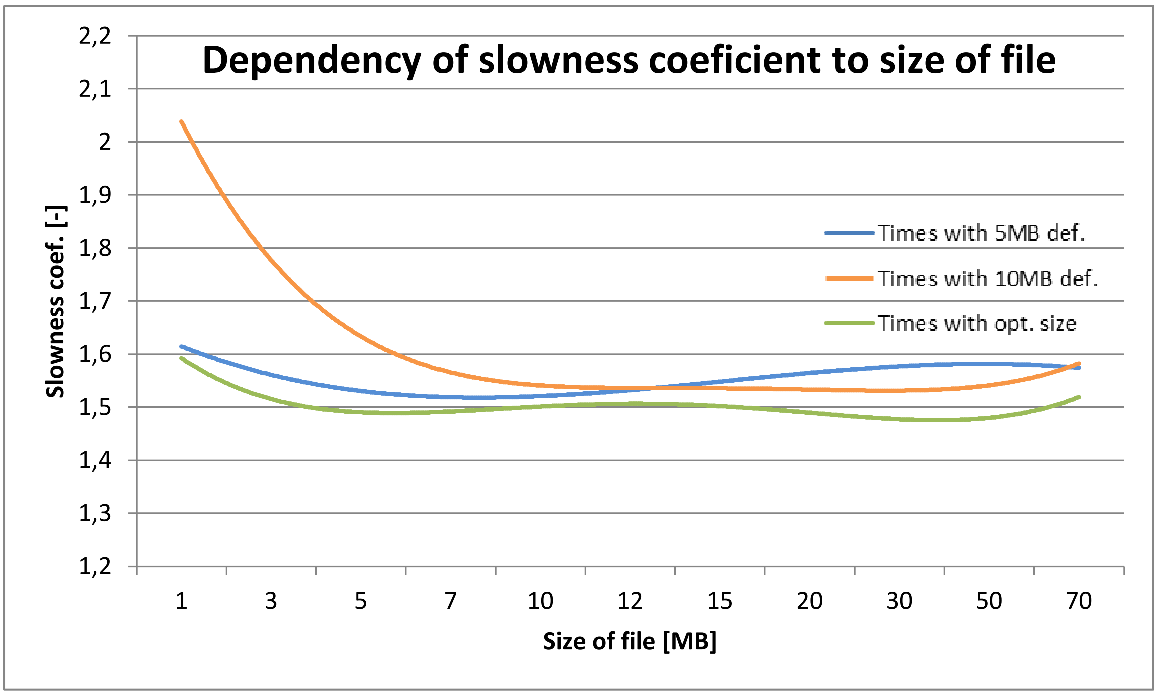
7.4 MFS performance test

The MFS described in 5.9.1 is one of the most interesting parts of the framework. It is also designed for very specific usage and it has to be known its performance before programmer uses it.

Designed performance test compares speed of few operations sequence between hard drive (Windows filesystem) and MFS. Test sequence consists of:

1) Create file of defined size, write byte by byte
2) Move created file to another location in the filesystem
3) Copy the file to another location in the filesystem

Size of created file is variable. Every sequence is done 10 times and then an average time is calculated. Every test is repeated 3 times and then the final coefficient for current file size is calculated as an average of three partial results.



Graph 1: Dependency of slowness coeficient to size of file

The orange line in Graph 1 shows an impact of MFS indirect costs. For files smaller than 5MB the writing is very quick and the indirect costs are relatively big part of result time. Allocating buffer is quite expensive operation. For buffer size set to 10MB, MFS has good results for files exceeding this buffer size.

The blue line shows us performance of MFS with default buffer size set to 5MB. Good results are achieved even for small files, because allocation of smaller buffer does not take so long time. On the contrary, for file sizes about 30MB, the buffer size 10MB gives us better results.

The green line in Graph 1 shows the optimal performance of MFS. The filesystem was created with parameter of default file size, which has caused that no memory reallocation was necessary while writing to the file. On the other hand, this method can be used only when we know size of the file in advance.

The conclusion is the MFS is not suitable for quick files operations, because it is about 1.5 times slower (in best case) than standard disk FS. On the other hand, the reason to use the MFS is different from the speedup (described in 5.9.1) and it fulfills its mission. We can optimize performance of MFS by setting size of copied file in advance.

8 Conclusion

In the frame of this thesis, I created list of required functions. According to that, I analyzed current situation on the field of GUI building and I compared proposed functionality with existing libraries. I decided to build new framework named KIV framework, which will satisfy all described requirements. I chose the Swing library as a basis of my framework.

I designed and described details of all parts of the framework. All designed parts were implemented and discussed afterwards. The whole framework was tested, partially by JUnit tests and partially by manual (visual) testing. MFS tools performance was tested and compared to an ordinary filesystem.

All requirements were satisfied and fully usable solution was implemented. The framework was released under the GNU LGPL license and it can be used by everyone.

There are many ideas that can be implemented in the future, such as integration of some useful components (e.g. JSyntaxPane, JFreeChart), implementation of new components (e.g. login dialog, about dialog) or implementation of XML to GUI generator.

References

1. Walrath K, Campione M, Huml A, Zakhour SB. The JFC Swing Tutorial: A Guide to Constructing GUIs: Sun corporation; 2004.
2. Oracle. The Swing tutorial. [Online]. [cited 2013 feb 20. Available from: http://docs.oracle.com/javase/tutorial/uiswing/start/about.html.
3. Darwin IF. Java Cookbook: O'Reilly; 2007.
4. Foudation A. Apache Pivot. [Online]. [cited 2013 mar 25. Available from: http://pivot.apache.org/about.html.
5. ZEUS. ZEUS Java Swing Component Library. [Online]. [cited 2013 mar 26. Available from: http://sourceforge.net/projects/zeus-jscl/.
6. BUOY. BUOY. [Online]. [cited 2013 mar 26. Available from: http://buoy.sourceforge.net/AboutBuoy.html.
7. CookSwing. CookSwing: XML to Java Swing GUI. [Online]. [cited 2013 mar 26. Available from: http://cookxml.yuanheng.org/cookswing/index.html.
8. Tödter K. JCalendar Java Bean. [Online]. [cited 2013 22 04. Available from: http://www.toedter.com/en/jcalendar/.
9. Factory Patterns. [Online]. [cited 2013 04 23. Available from: http://delphi.about.com/od/course/a/delphi_oop26.htm.
10. Patton R. Software testing: Pearson Education; 2005.
11. Schildt H. Java The Complete Reference: Oracle Press; 2011.
12. Tenouk.com. [Online]. [cited 2013 05 04. Available from: http://www.tenouk.com/visualcplusmfc/visualcplusmfc8_files/image023.png.
13. Tips4java. Wrap Layout. [Online]. [cited 2013 03 20. Available from: http://tips4java.wordpress.com/2008/11/06/wrap-layout/.
14. Noakes D. Metadata extractor. [Online]. [cited 2013 04 22. Available from: http://drewnoakes.com/code/exif/.
15. Javazoom. [Online]. [cited 2013 22 04. Available from: http://www.javazoom.net/javalayer/javalayer.html.

List of acronyms

GUI	– Graphical user interface
API	– Application interface, an interface which the others use to access the library
JFC	– Java Foundation Classes, see (1) for details
JDK	– Java Development Kit, a set of tools for Java Development
JNI	– Java Native Interface
XUL	– XML User interface Language
JAR	– Java archive, a file with the framework release
HTML	– Hypertext Markup Language, formatting language, mostly used at Internet pages
CRC	– Cyclic Redundancy Check, a unique calculated code, used for error – detecting
EXIF	– Exchangeable Image file Format
GPS	– Global Positioning System, a system for determining position on the Earth surface
JSON	– Javascript Object Notation, a text-based standard for human-readable data
Java EE	– Java Enterprise edition, a Java edition used in server applications (incl. the web app)
ANT	– a software tool for automated building of Java applications
WAV	– Waveform Audio File Format, an uncompressed audio format
MP3	– MPEG audio layer 3, a compressed audio format
CRUD	– Create, Read, Update, Delete
WYSIWYG	– What You See Is What You Get, class of UI designers, where the result is shown immediately
JVM	– Java Virtual Machine, a virtual machine, which executes a Java program

Attachment 1: Content of included CD

- this document in PDF and DOCX format
- complete sources of the project
- JAR file with compiled project, ready to be used
- demo application, contained in the JAR file (run the JAR)