8th homework assignment; OPRPP1

Napravite prazan Maven projekt: u Eclipsovom workspace direktoriju napravite direktorij `hw08-0000000000` (zamijenite nule Vašim JMBAG-om) te u njemu oformite Mavenov projekt `hr.fer.oprpp1.jmbag0000000000:hw08-0000000000` (zamijenite nule Vašim JMBAG-om) i dodajte ovisnost prema `junit`. Projekt treba imati i `src/main/resources` direktorij (te `src/test/resources` direktorij) u koji ćete stavljati lokalizacijske datoteke (u prikladnu strukturu direktorija). Importajte projekt u Eclipse. Sada možete nastaviti s rješavanjem zadataka.

Problem 1.

Your task is to create a simple text file editor called `JNotepad++`. The name of this editor must be shown in window's title. `JNotepad++` must allow user to work with multiple documents at the same time. For this, you must use `JTabbedPane` component:

http://docs.oracle.com/javase/8/docs/api/javax/swing/JTabbedPane.html
http://docs.oracle.com/javase/tutorial/uiswing/components/tabbedpane.html

For this problem use the package `hr.fer.oprpp1.hw08.jnotepadpp` and any subpackages you need. Your application must be startable by method `main` located in class `JNotepadPP` in package `hr.fer.oprpp1.hw08.jnotepadpp`. Since the images shown in the remainder of this document were created during the period of several years, please do not take packages shown on them literally – update them to reflect the package prescribed at the beginning of this paragraph.

For text editing use `JTextArea` component. For each open document you will create a new instance of `JTextArea` for it; this component will be then (indirectly) added to the `JTabbedPane`. I say indirectly because you must wrap it into `JScrollPane` and you may add this `JScrollPane` into `JPanel` (or other containers) which will eventually be added into the `JTabbedPane`.

Your application must provide the following functionality to the user:

- creating a new blank document,
- opening existing document,
- saving document,
- saving-as document (warn user if file already exists),
- close document shown it a tab (and remove that tab)
- cut/copy/paste text,
- statistical info,
- exiting application.

All of those actions must be available from:

- menus (organize them as you see fit),
- dockable toolbar,
- keyboard shortcuts.

Please see:
http://docs.oracle.com/javase/8/docs/api/javax/swing/JToolBar.html
http://docs.oracle.com/javase/tutorial/uiswing/components/toolbar.html

For open/save file selection use standard Java build-in dialogs: `JFileChooser`. See:
http://docs.oracle.com/javase/8/docs/api/javax/swing/JFileChooser.html
http://docs.oracle.com/javase/tutorial/uiswing/components/filechooser.html

Each tab should have an embedded icon (see `getIconAt(index)`, `setIconAt(index, icon)` for `JTabbedPane`) which visually indicates if the document is modified (for example, small green diskette for unmodified, small red diskette for modified). Each tab should additionally display filename of document (only filename; not whole path) or text "`(unnamed)`" if document has no associated path. Then, if the user holds a mouse cursor over the tab for some time, a tooltip with full document path should appear (see `get/setTitleAt(...)`, `get/setToolTipTextAt(…)`), or text "`(unnamed)`" if document has no associated path.

For working with icons please use `javax.swing.ImageIcon` class. Put icon files (PNG images) into `hr.fer.oprpp1.hw08.jnotepadpp.icons` subpackage (but in `src/main/resources` folder, only java code goes to `src/main/java`). Then refresh Eclipse project so that Eclipse becomes aware of new files. To achieve platform and location independent loading, you **must not load** these images using `FileInputStream` (or similar API). Instead, let assume that you write a method in class `X` which is in package `hr.fer.oprpp1.hw08.jnotepadpp`. You should simply ask Java Virtual Machine to open appropriate input stream for you the same way as it opened the input stream used to load the class itself into memory – let JVM takes care of the specific location. The pseudocode of solution (i.e. the content of the above-mentioned method) is:

```
InputStream is = this.getClass().getResourceAsStream("icons/redDisk.png");
if(is==null) error!!!
byte[] bytes = readAllBytes(is);
close input stream
return new ImageIcon(bytes);
```

Please read:

https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html#getResourceAsStream-java.lang.String-

Observe that in this example, since the method is in class in package `hr.fer.oprpp1.hw08.jnotepadpp`, we defined a relative path to images as: go to icons subdirectory (package structure is reflected in directory structure) and then find given PNG file.

If user attempts to close the program, you must check if there are any modified but unsaved text documents. If there are, ask the user for each document if he wants to save the changes, discard the changes or abort the closing action. Simplest way to implement this is to set default closing operation to be `DO_NOTHING_ON_CLOSE` and then to register in window constructor your implementation of `WindowListener` so that you can be informed when user attempts to close the program (use method `windowClosing` of interface `WindowListener`). You can read about `WindowListener` interface here:

http://docs.oracle.com/javase/8/docs/api/java/awt/event/WindowListener.html

and about `WindowAdapter` class:

http://docs.oracle.com/javase/8/docs/api/java/awt/event/WindowAdapter.html

Use method `addWindowListener` to add an instance of an anonymous class that is derived from `WindowAdapter` and not directly from `WindowListener` (do you understand why is this convenient?). In your implementation of `windowClosing` method call a method that will do the required checking. If everything is OK, this method should end with a call to `dispose()` method which will close the window

and eventually the program. If user decides to abort closing, you must skip the call to the `dispose()` method. When user calls the "exit" action from menu, you should simply call again your method that will check the status of all documents and that will allow user to abort the closing.

For communication with user, please use `JOptionPane` and its methods `showMessageDialog` and `showConfirmDialog`. See:

http://docs.oracle.com/javase/8/docs/api/javax/swing/JOptionPane.html
http://docs.oracle.com/javase/tutorial/uiswing/components/dialog.html

Even better approach that will allow you to easily localize dialogs is to use `JoptionPane#showOptionDialog` (check the documentation).

When user requests statistical info on document, you should calculate:

- a number of characters found in document (everything counts),
- a number of non-blank characters found in document (you don't count spaces, enters and tabs),
- a number of lines that the document contains.

Calculate this and show an informational message to user having text similar to: "Your document has X characters, Y non-blank characters and Z lines.".

When opening and saving the files, always use UTF-8 code page.

At all times, the path of currently selected document must be visible in window's title. To find out when the tab has changed, add appropriate listener to `JTabbedPane` component. Be careful to add this only once and not for each opened document. Here is a helpful example.

```
tabbedPane.addChangeListener(new ChangeListener() {
        public void stateChanged(ChangeEvent e) {
            System.out.println("Tab: " + tabbedPane.getSelectedIndex());
        }
    });
```

If user is currently editing `C:\example.txt`, the expected window title is:

```
C:\example.txt - JNotepad++
```

If user is editing new document (so the document does not have path information), expected window title is:

```
(unnamed) - JNotepad++
```

Do not enumerate new documents; so no `(unnamed1)`, `(unnamed2)`, …

Important: Implementation details

In order to solve this problem, you are expected to define two interfaces:

- **`MultipleDocumentModel`** represents a model capable of holding zero, one or more documents, where each document and having a concept of current document – the one which is shown to the user and on which user works.
- **`SingleDocumentModel`** represents a model of single document, having information about file path from which document was loaded (can be `null` for new document), document modification status and reference to Swing component which is used for editing (each document has its own editor component).

Both interfaces are *Subjects* for its state information and define appropriate registration/deregistration methods for its listeners (Observer design pattern).

Create class `DefaultSingleDocumentModel` as implementation of `SingleDocumentModel` interface (add private helper methods and properties and needed). It should define a constructor with two parameters: file path and text content; the constructor should create an instance of `JTextArea` and set its text content; reference to this text area must be returned from `getTextComponent()`. Further, the constructor should register a listener on `JTextArea` component's document model and use it to track its modified status flag (boolean property of `SingleDocumentModel`); each time this flag is modified, all registered listeners should be notified.

Create class `DefaultMultipleDocumentModel` by inheriting from `JTabbedPane` and implementing `MultipleDocumentModel`. This class should have a collection of `SingleDocumentModel` objects, a reference to current `SingleDocumentModel` and a support for listeners management. Add private helper methods and properties and needed. Ensure that in this class you track which tab is currently shown to user and update current `SingleDocumentModel` accordingly. Implement `loadDocument` to load specified file from disk, create `SingleDocumentModel` for it, add tab and switch to it. If there already is `SingleDocumentModel` for specified document, do not create new one, but switch view to this document. When loading new document or creating empty documents, add appropriate code/listeners which will ensure that tab icon tracks modification status of document (this is internal detail of `DefaultMultipleDocumentModel` class; outside code in not aware of this).

Create `JNotepadPP` as `JFrame` with menus and toolbars and an instance of `DefaultMultipleDocumentModel` where you look at its instances exclusively through `MultipleDocumentModel` interface. In `initGUI`, create this instance and add its visual component to the appropriate container. Implement all communication from your program (from menu actions etc.) with this object only through the methods declared in this interface. So if you were to add the instance directly into the frame, the code would look something like this:

```
MultipleDocumentModel mdm = new DefaultMultipleDocumentModel();
this.getContentPane().add(mdm.getVisualComponent(), ...-some-constraint-...);
```

In `JNotepadPP`, you will conceptually have menubar, toolbar, a single instance of `DefaultMultipleDocumentModel` and a single instance of statusbar. Let us repeat this: you are not allowed to create multiple statusbars (for example, for each document separate statusbar). For gui elements which must track and dynamically show information on current document, implement them to observe changes of current document, and on each change unregister appropriate listeners from "old" current document, register them on "new" current document and update presented information.

Here are the interfaces.

```
public interface MultipleDocumentModel extends Iterable<SingleDocumentModel> {
	JComponent getVisualComponent();
	SingleDocumentModel createNewDocument();
	SingleDocumentModel getCurrentDocument();
	SingleDocumentModel loadDocument(Path path);
	void saveDocument(SingleDocumentModel model, Path newPath);
	void closeDocument(SingleDocumentModel model);
	void addMultipleDocumentListener(MultipleDocumentListener l);
	void removeMultipleDocumentListener(MultipleDocumentListener l);
	int getNumberOfDocuments();
	SingleDocumentModel getDocument(int index);
	SingleDocumentModel findForPath(Path path); //null, if no such model exists
```

```
	int getIndexOfDocument(SingleDocumentModel doc);  //-1 if not present
}
```

`loadDocument`: path must not be `null`;

`saveDocument`: `newPath` can be `null`; if `null`, document should be saved using path associated from document, otherwise, new path should be used and after saving is completed, document's path should be updated to `newPath`.

`closeDocument`: removes specified document (does not check modification status or ask any questions).

`findForPath`: given path must not be `null`.

Please observe: `MultipleDocumentModel` can hold a single instance for each different path, but can have multiple "unnamed" documents (documents with no path associated). If `saveDocument` is called with `newPath` of some existing `SingleDocumentModel`, method must fail with appropriate exception. Your code which is calling this method should check that there is no other `SingleDocumentModel` for that path, and if there is, it should tell the user to close the other document first, and then try saving again.

`getVisualComponent`: returns the graphical component which is responsible for displaying the entire `MultipleDocumentModel`'s user interface. This component will be added into the hierarchy of graphical components which are displayed in window. If the `MultipleDocumentModel` is graphical component itself, it will simply implement the method as: `return this;`.

```
public interface SingleDocumentModel {
	JTextArea getTextComponent();
	Path getFilePath();
	void setFilePath(Path path);
	boolean isModified();
	void setModified(boolean modified);
	void addSingleDocumentListener(SingleDocumentListener l);
	void removeSingleDocumentListener(SingleDocumentListener l);
}
```

`setFile`: path can not be `null`.

```
public interface MultipleDocumentListener {
	void currentDocumentChanged(SingleDocumentModel previousModel,
SingleDocumentModel currentModel);
	void documentAdded(SingleDocumentModel model);
	void documentRemoved(SingleDocumentModel model);
}
```

`currentDocumentChanged`: `previousModel` or `currentModel` can be `null` but not both. Here the term "current" document means the one that the user is currently interacting with; you can also think about it as the "active" document.

```
public interface SingleDocumentListener {
	void documentModifyStatusUpdated(SingleDocumentModel model);
	void documentFilePathUpdated(SingleDocumentModel model);
}
```

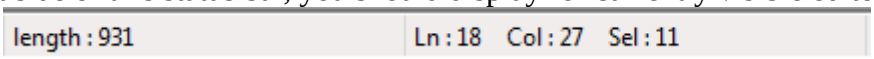
Problem 2.

Continue working in previous project. As part of this problem you will implement some additional functionality.

Subproblem 2.1

Add (visually) at the bottom of your editor a simple status bar. Be careful: tool bar must remain floatable, so think how to support this. The status bar should have an appropriate border. Status bar must be instantiated only once; do not create a different status bar for every opened document.

Aligned with the left side of this status bar, you should display for currently visible editor:



The `"length: "` displays the size of the document; `Ln:` displays the current line and `Col:` the current column in which is the caret (as user moves the caret, this information should be automatically updated). For purposes of displaying information to the user, enumerate lines and columns from 1 (not 0), so that when the caret is at the start of the document, user sees `Ln:1 Col:1`. The `Sel` display the length of current selection (if user selected anything); as selection grows, this field should be automatically updated. Check how this works when you change current document by clicking on tabs.

To receive information about caret, you can register appropriate listener on `JTextArea` (use `addCaretListener`).

Please update this information when user activates another editor (by clicking on its tab).

Aligned with the right end of the status bar, you must display a clock which shows current date and time. The format must be like: `2015/05/15 15:35:24`.

Subproblem 2.2

Implement the localization of this JNotepad++, as described in lectures. You must add a menu Languages/Jezici/Sprache, which when clicked shows menu items for a list of supported languages (add at least 3: Croatian, English, German) and instantly updates the whole GUI. The appropriate package for localization interfaces and classes (shown in lecture slides) is `hr.fer.oprpp1.hw08.jnotepadpp.local`.

Hint: you can skip the localization of `JFileChooser` dialogs. You can skip localization of button labels which are displayed by `JoptionPane`, or you can do the localization by using `showOptionDialog` (which is mandatory, if you aim for *excellent* grade for this homework).

When implementing localization, add `getCurrentLanguage` to `ILocalizationProvider`, and implement current language cacheing in `LocalizationProviderBridge` to enable notification dispatching if the bridge is reconnected to the source which had the language changed while the bridge was disconnected.

Subproblem 2.3

Add a menu Tools. Add submenu "Change case" with menu items:

- to uppercase
- to lowercase
- invert case

This tools act only on selected part of document; if no selection exists, this menu items should be disabled so that user can not activate them (observe: this menu items track some state from current document – try to minimize code duplication when implementing this).

Add submenu "Sort" with menu items "Ascending" and "Descending" which sort only the *selected lines* of text using rules of currently defined language. If user selection spans only part of some line, whole line is affected. Illustrative example which shows how the collators can be used for locale-sensitive string comparison:

```
Locale hrLocale = new Locale("hr");
Collator hrCollator = Collator.getInstance(hrLocale);
int r = hrCollator.compare("Češnjak", "Dinja");  // result is less than zero
```

See:

https://docs.oracle.com/javase/8/docs/api/java/util/Locale.html
https://docs.oracle.com/javase/8/docs/api/java/text/Collator.html

Add menuitem "Unique" which removes from selection all lines which are duplicates (only the first occurrence is retained).

To find out where some line starts and ends and to convert offset into line, if we know that we use `JTextArea` for editing, `JTextArea` offers methods:

```
int getLineOfOffset(int offset)
int getLineStartOffset(int line)
int getLineEndOffset(int line)
```

If we don't want to be explicitly dependent on the fact that we use `JTextArea` component (and we don't), we can obtain the same information from general `JTextComponent` (which is parent class of `JTextArea`) if we relay at least on the expectation that the document model will be of some known type. For example, `JTextComponent` by default stores information about the document in instances of class `javax.swing.text.PlainDocument` (which extends `javax.swing.text.AbstractDocument` which implements `javax.swing.text.Document`). Each `Document` stores content in hierarchy of objects which are `javax.swing.text.Element`. Each `Element` can have references to children which are again `Element`, and so on. This way, the document model can easily be used to represent documents with tree-based structure, such as HTML and similar. The reference to root element can be obtained by `Document#getDefaultRootElement()`.

The specificity of `PlainDocument` (see javadoc!) is the fact that the document representation is relatively shallow: it has a single root element, and the root element has one child element for each line. Each of those elements contain all characters for the line which element represent. Having this in mind, if we have a linear position (which we typically obtain from the caret):

```
JtextComponent c = someTextComponent();
int pos          = c.getCaretPosition();
```

we can use `Element#getElementIndex(pos)` on root element to find the child element index which

contains that position. In `PlainDocument`, that will also correspond to the line number (indexes from 0) since each child element is one line:

```
Document doc = c.getDocument(); // we assume it is instanceof PlainDocument
Element root = doc.getDefaultRootElement();
int row      = root.getElementIndex(pos); // zero-based line index...
```

The column can be calculated by subtracting the given position and the linear position at which the line starts:

```
int col = pos - root.getElement(row).getStartOffset(); // also zero-based...
```

Having the reference to document, you can also call some interesting methods (for this homework):

```
doc.getText(fromPos,toPos),
doc.remove(fromPos,toPos),
doc.insertString(...),
```

WHAT FOLLOWS IS ONLY ILLUSTRATIVE EXAMPLE OF LOCALIZATION FOR THOSE STUDENTS WHICH MISSED THE CLASS (OR THE CONCENTRATION). THIS CAN BE USED AS HELP FOR HOMEWORK (to implement requested i18n) BUT IN ITSELF IS NOT A PART OF THE HOMEWORK.

Warming up

We will start by a simple example. You don't have to upload the following code as part of your homework, so please open a new “dummy” project that will allow you to experiment with the following examples. This example assumes you are working with standard Eclipse project, not Maven project, so create a new Eclipse project for this example (again: this is not directly part of your homework so do not upload this additional project; however, some parts of the developed interfaces/classes you can copy in your actual homework; but take care to put non-java files into `src/main/resources` and only java files into `src/main/java` when copying files). Also, update package names appropriately to indicate actual homework number.

Lets say you need to support several languages for your user interface. Let's start by Croatian and German. Imagine you have to add a button with which a user could start a log-in procedure. Here is a simple example. Copy it and try it.

```
package hr.fer.oprpp1.hw08.vjezba;

import java.awt.BorderLayout;
import java.awt.HeadlessException;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.SwingUtilities;
import javax.swing.WindowConstants;

public class Prozor extends JFrame {

	private static final long serialVersionUID = 1L;
	private String language;

	public Prozor(String language) throws HeadlessException {
		this.language = language;
		setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
		setLocation(0, 0);
		setTitle("Demo");
		initGUI();
		pack();
	}

	private void initGUI() {
		getContentPane().setLayout(new BorderLayout());

		JButton gumb = new JButton(
			language.equals("hr") ? "Prijava" : "Login"
		);
		getContentPane().add(gumb, BorderLayout.CENTER);

		gumb.addActionListener(new ActionListener() {
			@Override
			public void actionPerformed(ActionEvent e) {
				// Napravi prijavu...
			}
		});
```

```
	}

	public static void main(String[] args) {
		if(args.length != 1) {
			System.err.println("Očekivao sam oznaku jezika kao argument!");
			System.err.println("Zadajte kao parametar hr ili en.");
			System.exit(-1);
		}
		final String jezik = args[0];
		SwingUtilities.invokeLater(new Runnable() {
			@Override
			public void run() {
				new Prozor(jezik).setVisible(true);
			}
		});
	}
}
```

The state of Eclipse project is given on Figure 2.1a. The result when program is started by command:

```
java -cp bin hr.fer.oprpp1.hw08.vjezba.Prozor hr
```

is given on Figure 2.1b. The result when program is started by command:

```
java -cp bin hr.fer.oprpp1.hw08.vjezba.Prozor en
```

is given on Figure 2.1c.

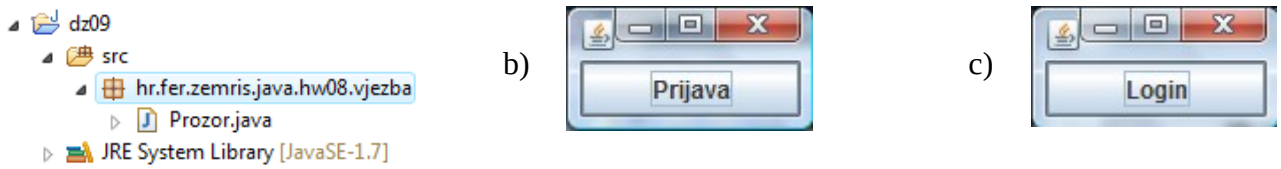


Figure 2.1

In given solution the key elements are: the `Prozor` class which has a member variable `language` which holds the identifier for currently selected language and method `initGUI()` which creates components using the information on currently selected language:

```
public class Prozor extends JFrame {

	private String language;

	private void initGUI() {

		JButton gumb = new JButton(
			language.equals("hr") ? "Prijava" : "Login"
		);

	}
}
```

One of the problems with this solution is that it is not easily extendable. What if we want to add several additional languages? What if we need the same translation on more than one place – will we hard-code the translation on each of those places? What then if the translation is wrong? And, last but not least important, are we happy with the need to recompile entire code just because we changed the some translation or

corrected a typo?
Today, translations are typically implemented by translation-externalization. In code, for each translation we need we define a new key. This key can be a string or a numerical value which uniquely identifies the needed translation. More often then not, keys are strings. Then, for each translation, a text file is prepared (so called translation bundle) in which each line contains a translation for a single key. Lets define that the translation for the text “Login/Prijava/Anmelden/...” will be stored under the key “login”. Now, for each supported language we can create a new text file associating the key with the translation.

prijevodi_hr.properties

```
login = Prijava
logout = Odjava
```

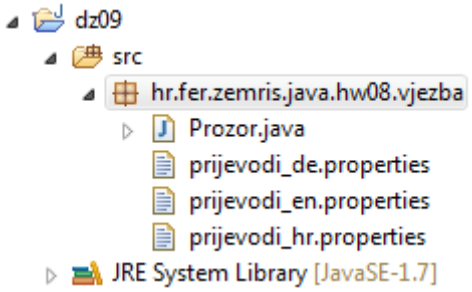
prijevodi_de.properties

```
login = Anmelden
logout = Abmeldung
```

prijevodi_en.properties

```
login = Login
logout = Logout
```

We will place those file in the same package in which we have the `Prozor` class. All translation files are named on the same way: we have the common part (`prijevodi`), underscore, language identifier and the “.properties” extension.



Eclipse will copy these files into the `bin` directory so that, when the program is started, it can locate these files in its *classpath* (which includes the `bin` directory but does not include the `src` directory). Please be warned that if you change localization files in `bin` directory outside of Eclipse, Eclipse will overwrite it with copies from `src` directory. Also, if you modify localization files in `src` directory outside of Eclipse, Eclipse will not be aware of the modifications until you refresh your project (F5) and outdated copy will exist in `bin` directory which will be used when you launch your application.

Now we can modify the previous code so that it uses the translations we prepared. Don't worry, Java has all the required classes already in place – you just have to learn to use them. Please modify the code as shown below:

```
private void initGUI() {
      getContentPane().setLayout(new BorderLayout());

      Locale locale = Locale.forLanguageTag(language);
      ResourceBundle bundle =
             ResourceBundle.getBundle("hr.fer.oprpp1.hw08.vjezba.prijevodi", locale);

      JButton gumb = new JButton(
            bundle.getString("login")
```

```
        );

        ...
}
```

Java expect us to represent localization information as an instance of `Locale` class. This class has a static factory method `forLanguageTag` which accepts language identifier (hr/en/de/...) and returns the required object. Once we have the `Locale` object, we can request the `ResourceBundle` that represents our translations for the given language. `ResourceBundle` is named just like a class: it has a name (in our case `prijevodi`) and a package in which it resides (in our case: `hr.fer.oprpp1.hw08.vjezba`). So, its full name is `hr.fer.oprpp1.hw08.vjezba.prijevodi`. In our example, resource bundle represents a set of translations. Which translations should be used is determined by the second argument: `Locale` object. When we call a method *`getBundle`* and provide a full resource name and the `Locale` object, an appropriate file will be accessed on the disk, opened and loaded into the memory. The object we obtain this way behaves as a map: we call a method `bundle.getString` with a key and we get the translation associated with that key.

Please try this example. Run the program with arguments hr, then de, then en and see that it works. Select another language – prepare its translation file and check that program works correctly.

Everything works OK? The method `ResourceBundle.`*`getBundle`* performs quite a lot of work for you. Unfortunately, this means that it is slow. It will try to cache the results, so it will try to serve you the same object if you call it multiple times with the same object. However, we will try to devise a solution in which we won't have to to call it multiple times – just to be on the safe side. The problem that arises in our current solution is that the i18n is implemented in single `JFrame`. What should we do if we have multiple windows? Should each window try to load its own `ResourceBundle`? How should we communicate the information on the selected language to all of those frames? And, finally, how can we achieve a dynamic change of language while the program is running? We will need a little help from the Singleton design pattern and the Observer design pattern. You have learned about the latter already; now please read about the former:

http://en.wikipedia.org/wiki/Singleton_pattern

What we aim at is the code like the following one:

```
public class Singleton {
    private static final Singleton instance = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return instance;
    }
}
```

Do you understand what does it do and how it guarantees that only a single instance of a class will be created?

We will use the singleton design pattern to store the information on the selected language and the loaded resource bundle.

Please look at the following class diagram.

We will define an interface named `ILocalizationProvider`. Object which are instances of classes that implement this interface will be able to give us the translations for given keys. For this reason there is a declared method

```
String getString(String key);
```

It takes a key and gives back the localization. Since we would like to support the dynamical change of selected language, here we also utilize the Observer pattern: each `ILocalizationProvider` will be automatically the Subject that will notify all registered listeners when a selected language has changed. For that purpose, the `ILocalizationProvider` interface also declares a method for registration and a method for de-registration of listeners. Although it is not shown in the class diagram, add:

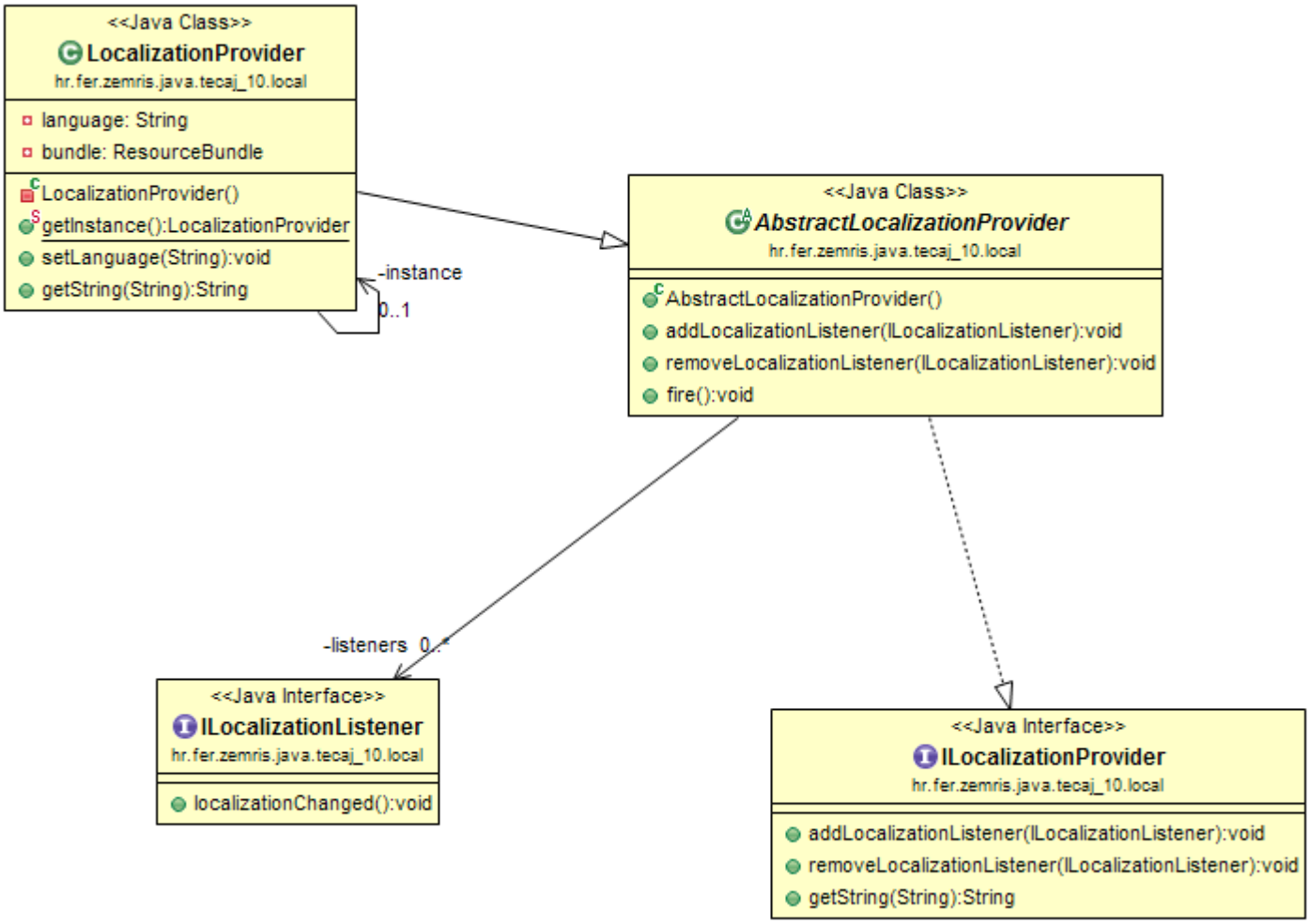
```
String getLanguage();   // returns current language
```

into this interface as well.

A listener is modeled with `ILocalizationListener` and has a single method:

```
void localizationChanged();
```

which will be called by the Subject when the selected language changes.



The `AbstractLocalizationProvider` class implements `ILocalizationProvider` interface and adds it the ability to register, de-register and inform (`fire()` method) listeners. It is an abstract class – it does not implement `getString(...)` method.

Finally, `LocalizationProvider` is a class that is singleton (so it has private constructor, private static instance reference and public static getter method); it also extends `AbstractLocalizationProvider`.

Constructor sets the `language` to “en” by default. It also loads the resource bundle for this language and stores reference to it. Method `getString` uses loaded resource bundle to translate the requested key.

Implement all of those classes and interfaces and check that your program still works. At this point, the creation of the button should be performed as this:

```
JButton gumb = new JButton(
	LocalizationProvider.getInstance().getString("login")
);
```

In method `main()` you should set the requested language:

```
public static void main(String[] args) {
	if(args.length != 1) {
		System.err.println("Očekivao sam oznaku jezika kao argument!");
		System.err.println("Zadajte kao parametar hr ili en.");
		System.exit(-1);
	}
	final String jezik = args[0];
	SwingUtilities.invokeLater(new Runnable() {
		@Override
		public void run() {
			LocalizationProvider.getInstance().setLanguage(jezik);
			new Prozor().setVisible(true);
		}
	});
}
```

As show in this example, you should also delete a language argument from `Prozor` constructor and its private member variable `language` – i18n is now handled by the `LocalizationProvider` class.

More warming up

Now it is time to allow a dynamical change of language while the program is running. While creating the button, remember the reference to it in a member variable so you can access it even after the `initGUI` method is finished.

Add a menu bar, add menu “Languages” and add three menu items: “hr”, “en”, “de”. Implement action listeners for each of these menu items so that when clicked, they will call:

```
LocalizationProvider.getInstance().setLanguage("en");
```

(or “hr” or “de”). Now register an instance of anonymous class in Prozor constructor as listener for localization changes:

```
LocalizationProvider.getInstance().addLocalizationListener(...);
```

Implement the method `localizationChanged` as this:

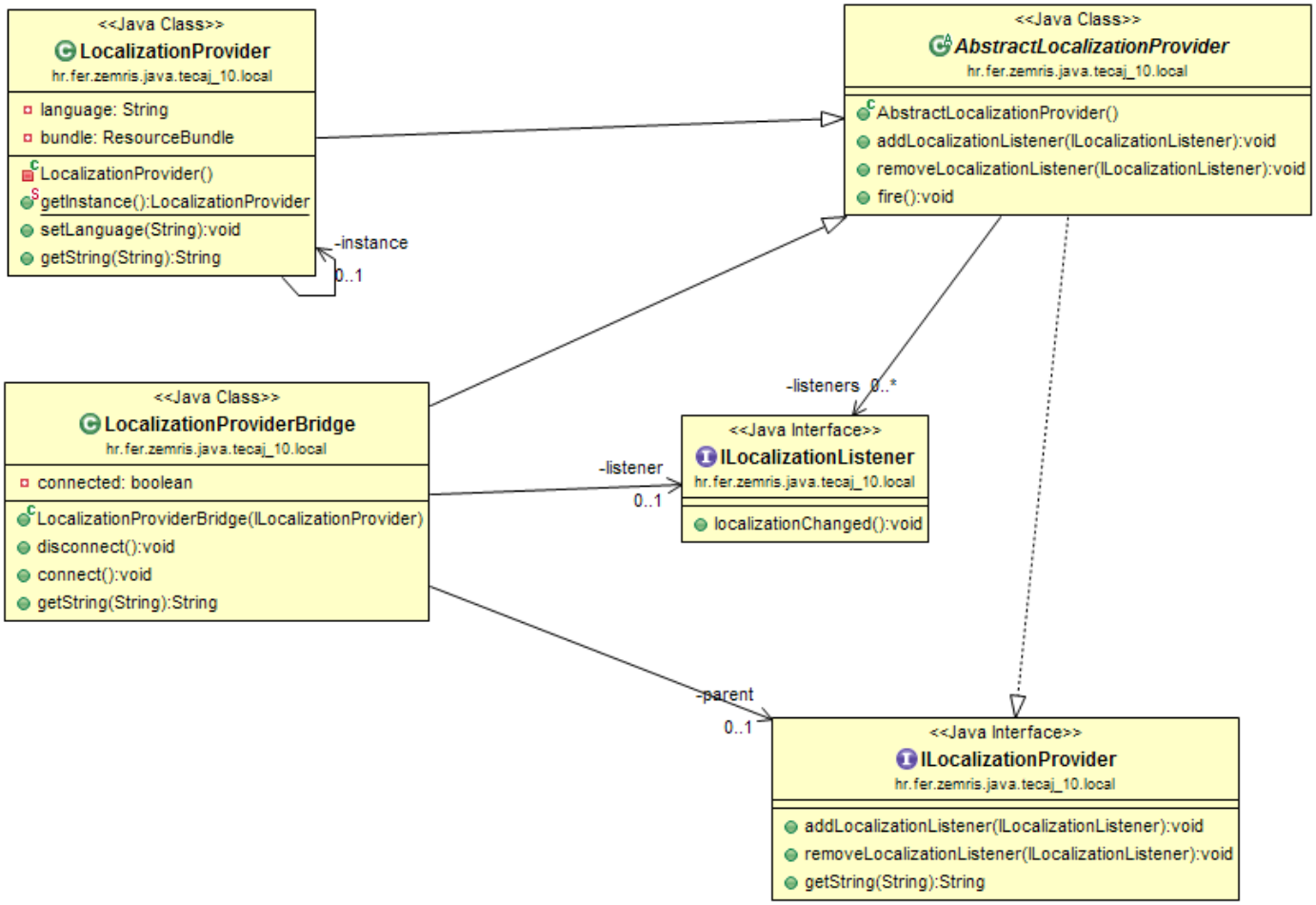
```
public void localizationChanged() {
  button.setText(LocalizationProvider.getInstance().getString("login");
}
```

Now start the program; as you select different languages from menu, the text shown on button should automatically change. Try it. If it does not work, go back and try to fix it. It it important that you make it work because the rest this problem depends on that.

And even more warming up

OK. Our previous solution is one step in right direction, one step in wrong direction. Right direction is: we have i18n working. Wrong direction is: a frame registers itself for a notifications on a `LocalizationProvider`. This has unfortunate consequence that `LocalizationProvider` holds a reference to this frame, so if we dispose the frame, garbage collection will still be unable to release its memory because the frame is not garbage yet – there is someone who holds the reference to it. In a program which opens and closes multiple frames, this can become a significant issue.

A solution of this problem is given on following class diagram.



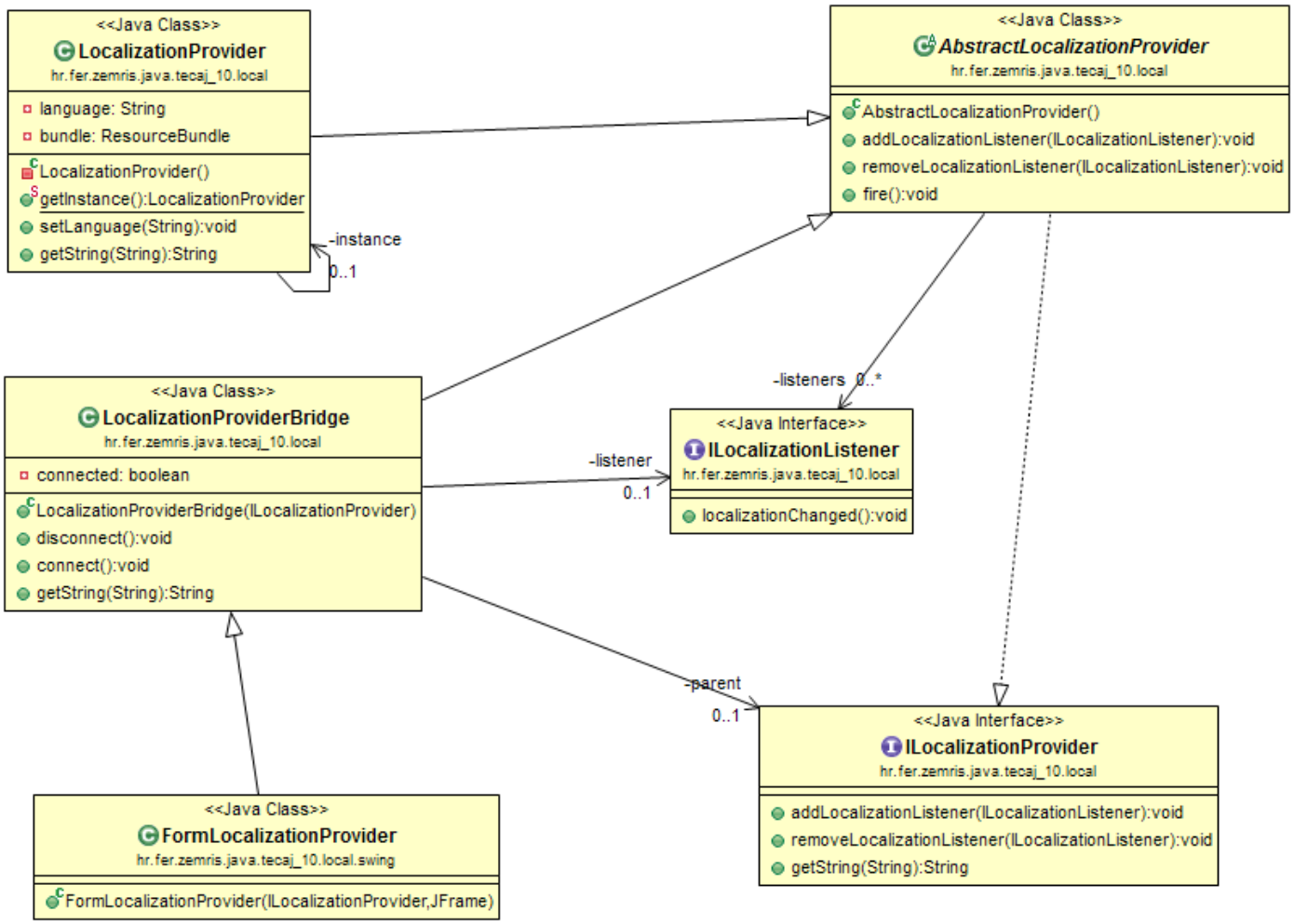
First step is to add a new class: `LocalizationProviderBridge` which is a decorator for some other `IlocalizationProvider`. This class offers two additional methods: `connect()` and `disconnect()`, and it manages a connection status (so that you can not connect if you are already connected). Here is the idea: this class is `ILocalizationProvider` which, when asked to resolve a key delegates this request to wrapped (decorated) `ILocalizationProvider` object. When user calls `connect()` on it, the method will register an instance of anonimous `ILocalizationListener` on the decorated object. When user calls `disconnect()`, this object will be deregistered from decorated object. Although it is not shown in the class diagram, add:

```
private String currentLanguage;
```

into `LocalizationProviderBridge` class. Use this field to keep track of the its parent's language, while bridge is connected to the parent. If bridge is disconnected, when user calls `connect()`, you must check if parent still uses the same language, and if that is not the case, update bridge's current language, and force bridge to notify its listeners that the language was changed.

The `LocalizationProviderBridge` must also listen for the localization changes on its parent so that, when it receives the notification, it will notify all of the listeners that are registered on that bridge.

Now create `FormLocalizationProvider` (see the following class diagram).



`FormLocalizationProvider` is a class derived from `LocalizationProviderBridge;` in its constructor it registeres itself as a `WindowListener` to its `JFrame;` when frame is opened, it calls `connect` and when frame is closed, it calls `disconnect`. Now for each `JFrame` instance we create, we will add an instance variable of this type (`flp` in code below), and initialize it in `JFrame` constructor. The code should look like this:

```
public class SomeFrame extends JFrame {

  private FormLocalizationProvider flp;

  public SomeFrame() {
    super();
    flp = new  FormLocalizationProvider(LocalizationProvider.getInstance(), this);
  }

}
```

Now, when the frame is created, `flp` will register itself to decorated localization provider automatically;

when frame closes, `flp` will de-register itself from the decorated localization provider automatically so that it won’t hold any reference to it, and the garbage collector will be able to free frame and all of its resources (when frame is disposed).

In frames written this way, we won't explicitly register new GUI components on singleton object; instead, we will register them on `flp` object. Add this code in `Prozor`. Then remove from `Prozor` constructor localization listener you previously added:

```
LocalizationProvider.getInstance().addLocalizationListener(...);
```

and register it instead on `flp`.

```
flp.addLocalizationListener(...);
```

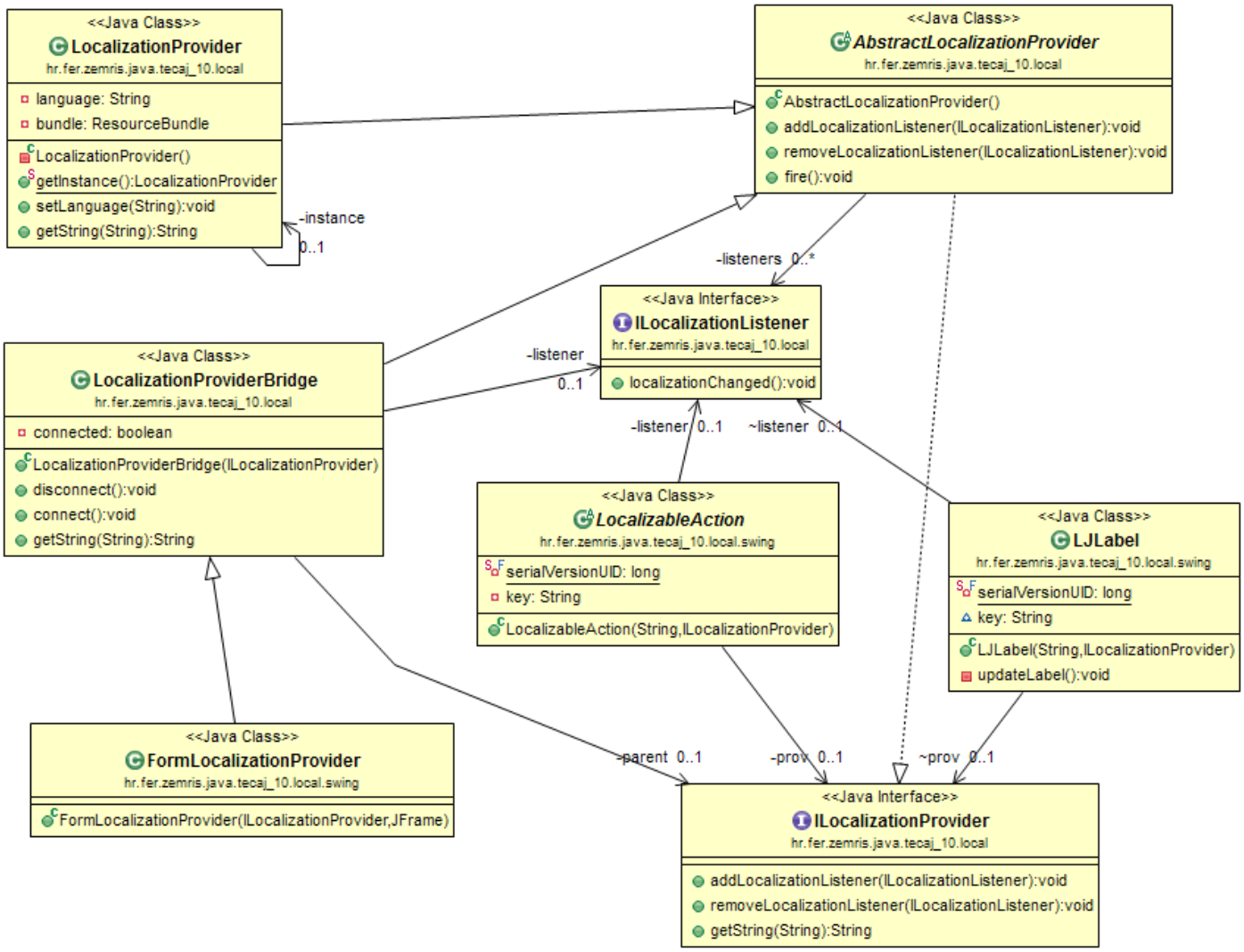
Change the way you construct button to this:

```
JButton gumb = new JButton(
      flp.getString("login")
);
```

Run the program and ensure that it still works correctly, and that you can dynamically change languages.

And even more more warming up

There is one thing left for you to do before I state the actual problem. We are creating buttons by giving them text to display on them? I hope you still remember that this is a bad thing to do. What we need are localized actions and localized Swing components which are not based on actions. Please take a look at the following class diagram.



We have a class `LocalizableAction`. This class extends the `AbstractAction` class (not shown on the diagram) and defines a single constructor:

```
public LocalizableAction(String key, ILocalizationProvider lp);
```

Please observe that the first argument is not a text for action (so called action-name); it is a key. The second argument is a reference to localization provider. In all our examples, we create actions in frames, so the second argument will be the `flp` reference.

In `LocalizableAction` constructor you must ask `lp` for translation of the `key` and then call on `Action` object `putValue(NAME, translation)`. You must also register an instance of anonymous class as a listener for localization changes on `lp`. When you receive a notification, you must again ask `lp` to give you a new translation of action's `key` and you must again call `putValue(NAME, translation)`. Since this method changes action's properties, the action itself will notify all interested listeners about the change, and then all

GUI components (buttons, menu items) will automatically refresh itself.

For GUI components that are not based on actions, you can prepare each component on similar way. For example, on previous class diagram there is `LJLabel` component which extends `JLabel`; it has a constructor just like the class `LocalizableAction` and it does the same: it registers itself on given `lp` and when it receives a notification, it translates the `key` again and sets the translation as a new text that it displays.

Now make a final change in your demo program:

```
JButton gumb = new JButton(
      new LocalizableAction("login", flp) { … }
);
```

Start the program and make sure that it works.

Please note. You can consult with your peers and exchange ideas about this homework *before* you start actual coding. Once you open you IDE and start coding, consultations with others (except with me) will be regarded as cheating. You can not use any of preexisting code or libraries for this homework (whether it is yours old code or someones else). You can use Java Collection Framework and other parts of Java covered by lectures; if unsure – e-mail me. Document your code!

All source files must be written using UTF-8 encoding. All classes, methods and fields (public, private or otherwise) must have appropriate javadoc.

You are not required to unit test code in this homework.

When your **complete** homework is done, pack it in zip archive with name `hw08-0000000000.zip` (replace zeros with your JMBAG). Upload this archive to Ferko before the deadline. Do not forget to lock your upload or upload will not be accepted.