10. homework assignment; JAVA, Academic year 2014/2015; FER

This homework has two problems.

Problem 1.

Create new maven project: groupId **`hr.fer.zemris.java.studentVASJMBAG.hw10`**, artifactId **`jnotepadpp`**. 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.zemris.java.hw10.jnotepadpp` and any subpackages you need. Your application must be startable by method `main` located in class `JNotepadPP` in package `hr.fer.zemris.java.hw10.jnotepadpp`.

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 embedded an icon (see `getIconAt(index)`, `setIconAt(index)` 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). Then user holds mouse cursor over tab for some time, a tooltip with full document path should appear (see `get/setTitleAt(...)`, `get/setToolTipTextAt(...)`).

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

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++
```

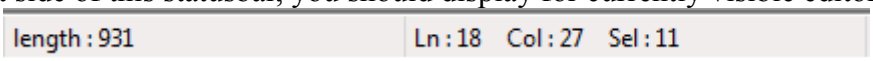
Problem 2.

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: toolbar must remain floatable so think how to support this. The statusbar should have an appropriate border.

Aligned with the left side of this statusbar, 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). The Sel display the length of current selection (if user selected anything); as selection grows, this field should be automatically updated.

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 statusbar, 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.

Hint: you can skip the localization of `JFileChooser` dialogs, and button labels which are displayed by `JOptionPane`.

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 menuitems should be disabled so that user can not activate them.

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, `JTextArea` offers methods:

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

This information can then be used to query the appropriate part of the document:

```
JTextArea.getDocument().getText(fromPos,toPos),
JTextArea.getDocument().remove(fromPos,toPos),
JTextArea.getDocument().insertString(...),
```

`Document` is the model for `JTextArea`; `JTextArea` is view which displays the content of the document and offers mechanism for document editing.

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 give following code as part of homework so please open a new "dummy" project that will allow you to experiment with following examples.

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.zemris.java.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);
                }
            });
        }
}
```

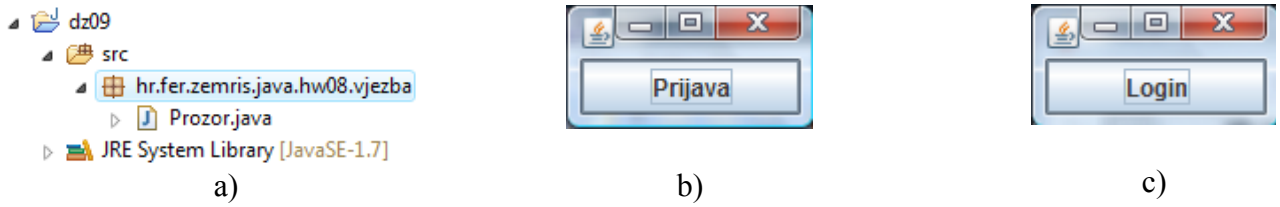
The state of Eclipse project is given on Figure 2.1a. The result when program is started by command:

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

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

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

is given on Figure 2.1c.



a) b) c)

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
```

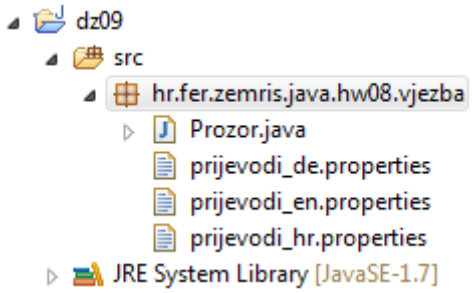
prijevodi_de.properties

```
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.zemris.java.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.zemris.java.hw08.vjezba`). So, its full name is `hr.fer.zemris.java.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);
```

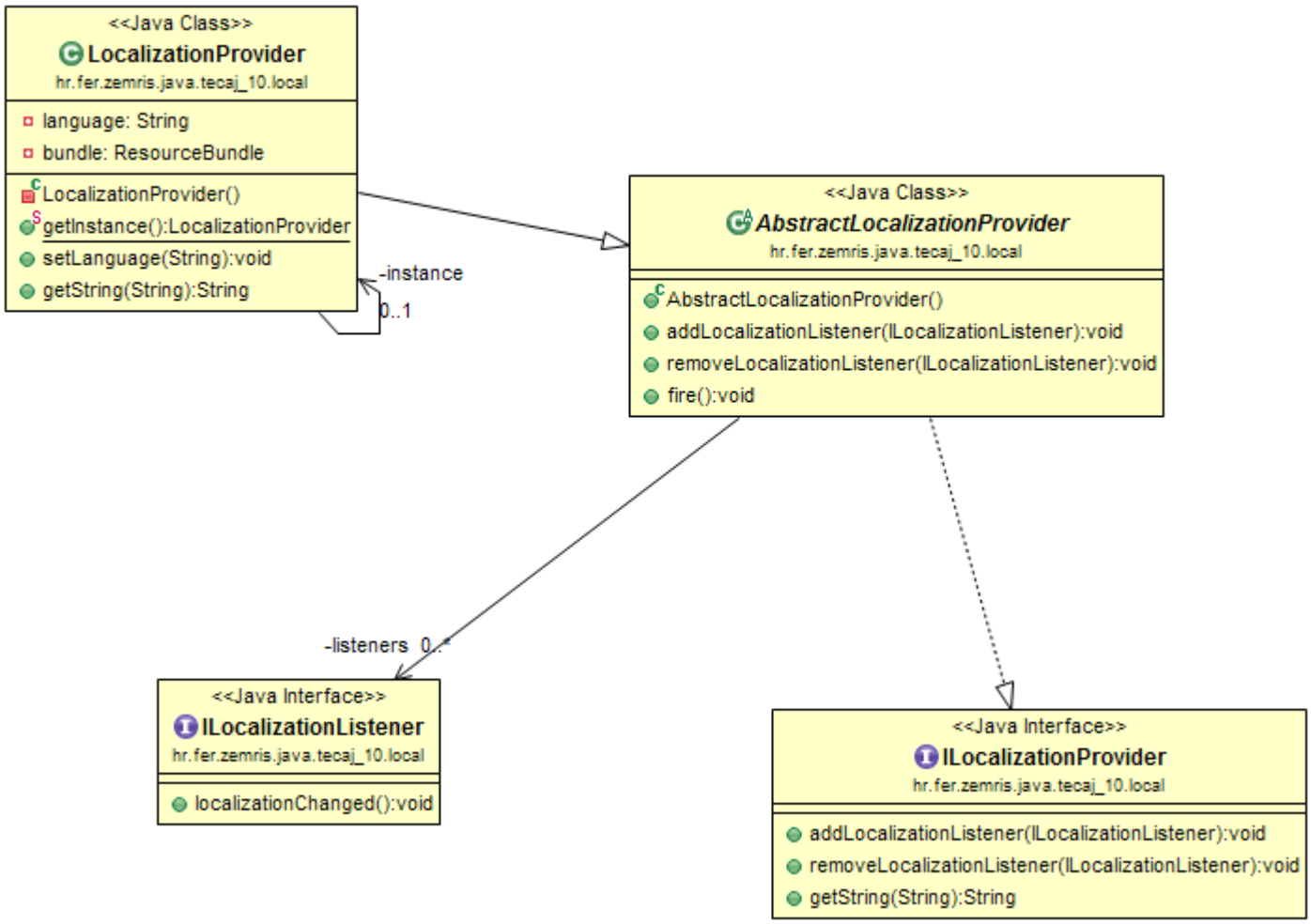
It takes a key and gives back the localization. Since we would like to support the dynamical change of

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.

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");
}
```

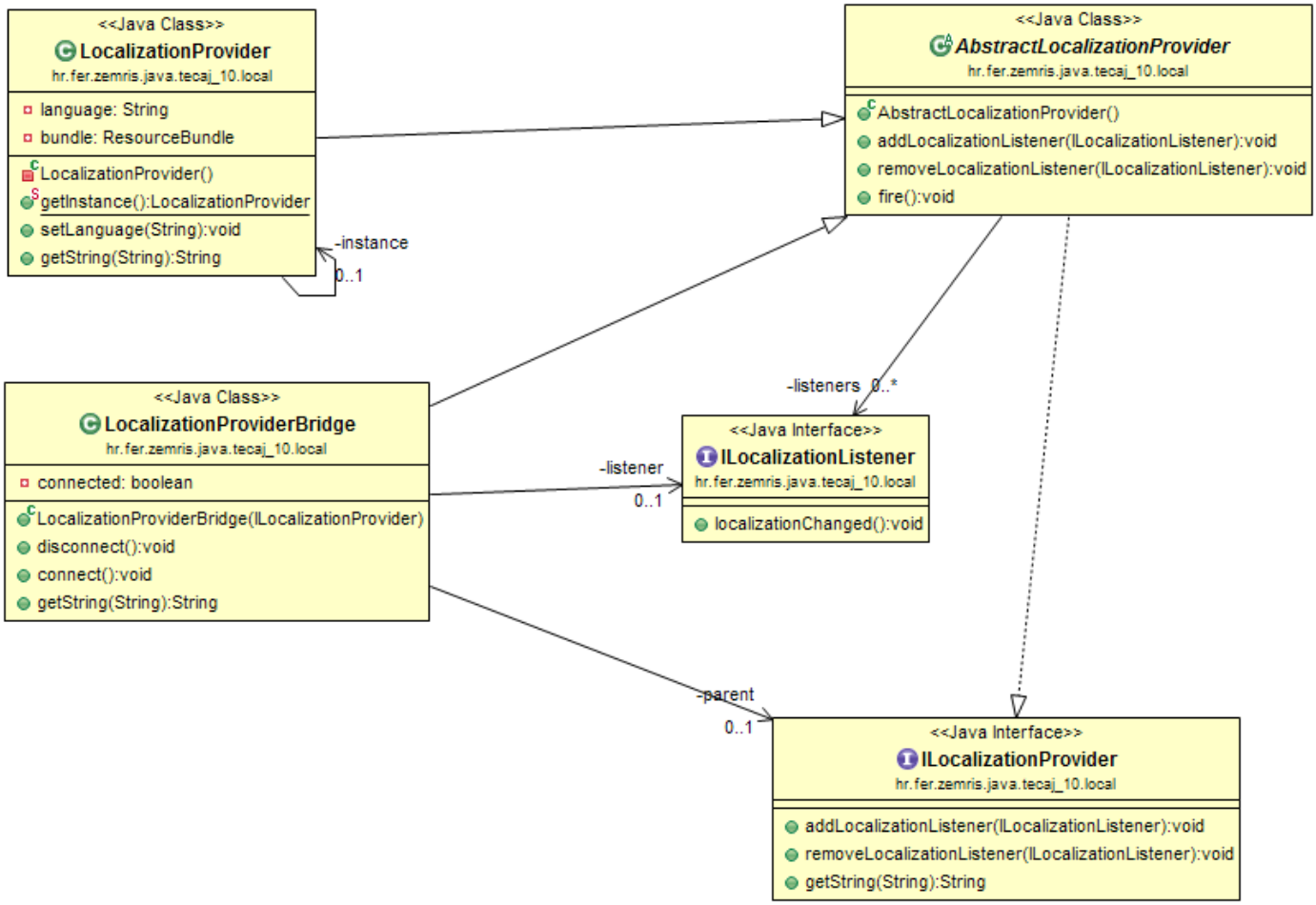
Now start the program; as you select different languages from menu, the text shown on button should automatically change.

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.

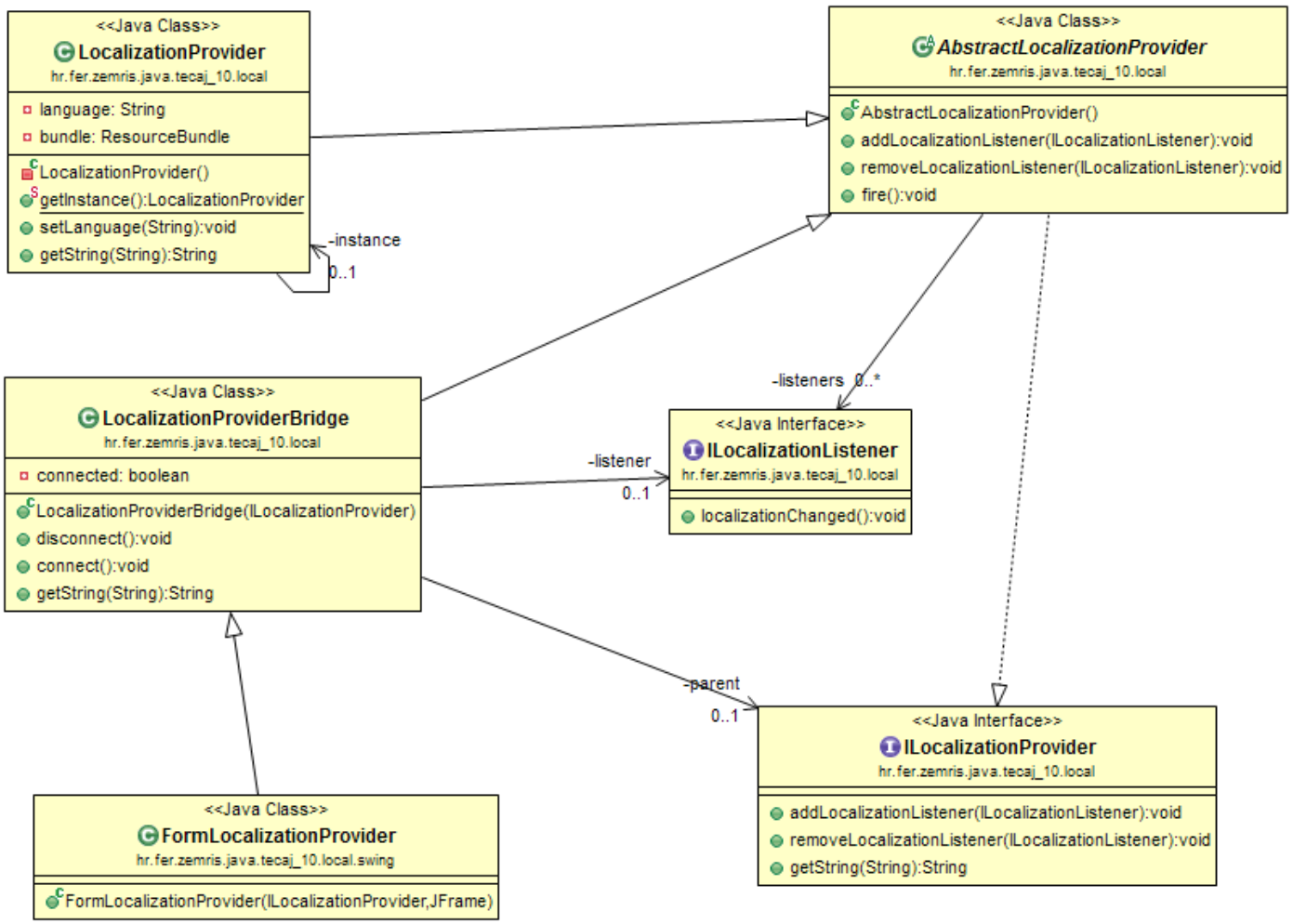
A solution of this problem is given on following class diagram.



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.

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

Now create `FormLocalizationProvider` (see 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` we will create we will add an instance variable of this type and in its constructor create it. 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 frame opens, `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 wont hold any reference to it and the garbage collector will be able to free frame and all of its resources (is frame is disposed).

In frames written this way, we won't explicitly register ourselves on singleton object but where needed, on `flp` object. So now 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(...);
```

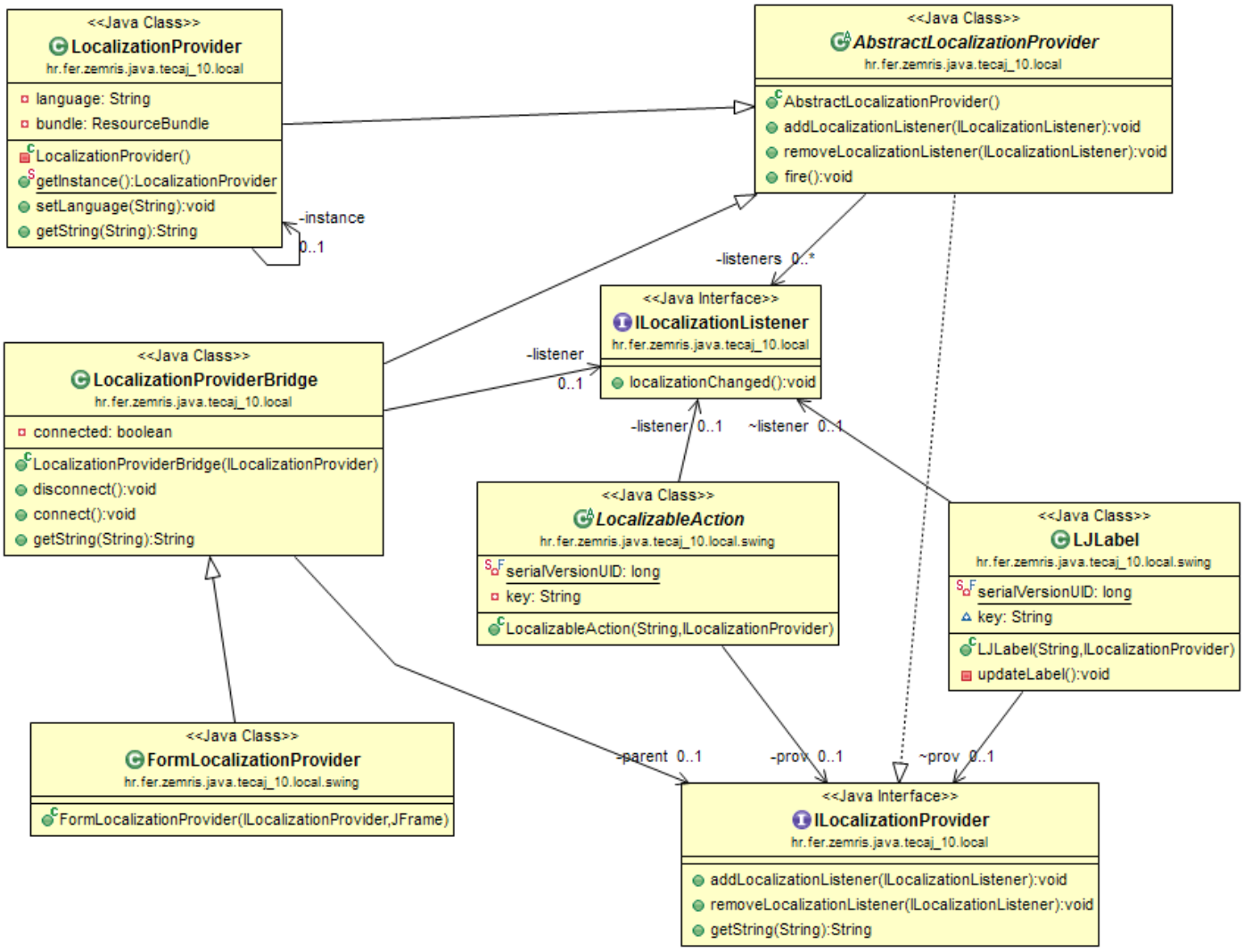
Change the way you construct button to this:

```
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 will 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 anonimous 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, action will notify all interested listeners about the change and 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 of 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.

Important notes

You must create a single ZIP archive containing all projects which you have created as part of this homework (each in its own folder), and than upload this single ZIP. ZIP arhive must have name `HW10-`*yourJMBAG*`.zip`.

All of the classes should have appropriate javadoc. No test are required (but are strongly advised, where appropriate).

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 which is not part of Java standard edition (Java SE) unless explicitly allowed or provided by me. You can use Java Collection Framework classes and its derivatives. Document your code!

Upload final ZIP archive to Ferko before the deadline. Do not forget to lock your upload or upload will not be accepted. Deadline is May 23nd 2015. at 07:00 AM.