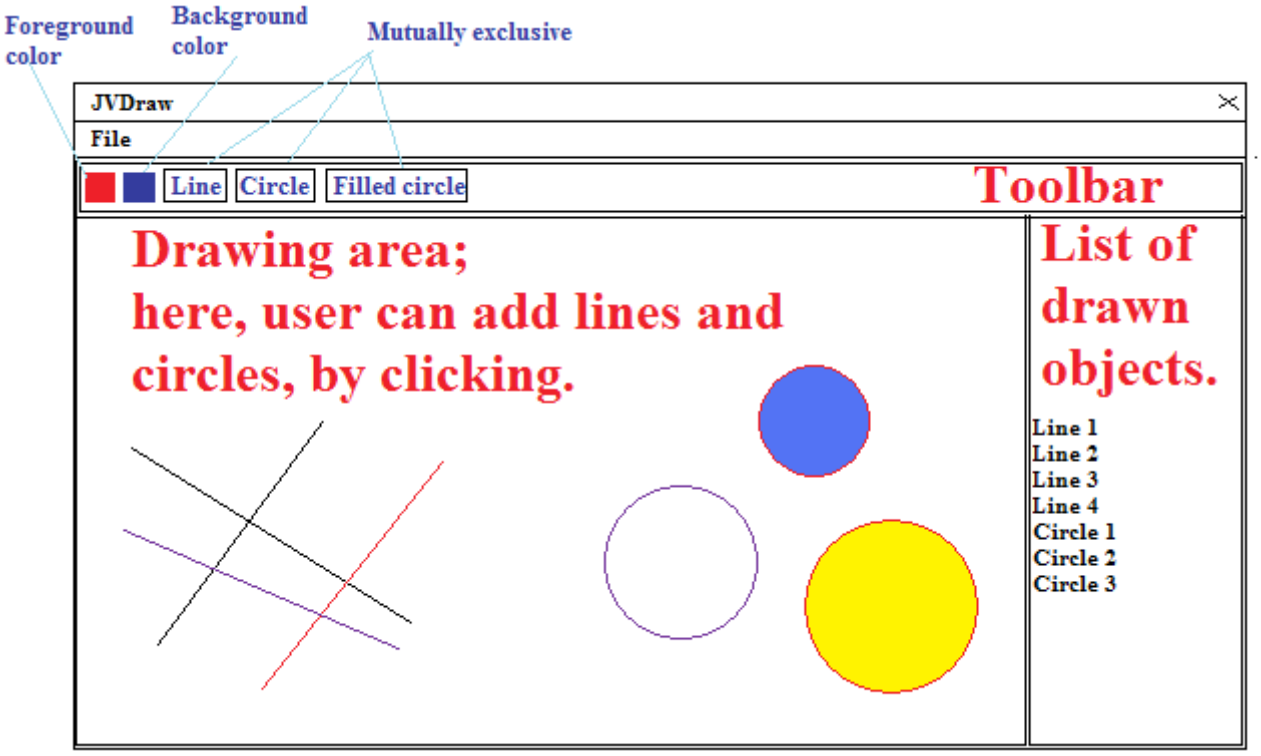
11th homework assignment; JAVA, Academic year 2013/2014; FER

As usual, please see the last page. I mean it! You are back? OK. Here we have 2 problems for you to solve.

Problem 1.

Important: please read the whole text of problem 1 before trying to solve it. Identify all the classes and interfaces you are required to create. Sketch a class diagram and relations among the classes – who extends who, who implements who, and who has a reference to who. Identify the design patterns used; label each class/interface according to its purpose in the design pattern. Once you understand the roles and the relationships among classes, start to implement you solution. Scan your drawings: you will have to upload them as well.

All classes developed as part of this problem must go into package `hr.fer.zemris.java.hw08.jvdraw` (and its subpackages, if necessary). We are developing a GUI application called *JVDraw*, which is a simple application for vector graphics. The main class (a class which is used to start the program) must be `hr.fer.zemris.java.hw08.jvdraw.JVDraw`.
When started, program will show an empty canvas. Sketch of the program is shown below.



Program allows user to draw lines, circles and filled circles. Program features menubar, toolbar, drawing canvas and object list.

Toolbar has five components: there are two `JColorArea` components and three mutually exclusive `JToggleButtons` (only one can be selected at any time).

JColorArea

Write the code for this component; extend it from `JComponent`. Override its method `getPreferredSize()` so that it always returns a new dimension object with dimensions 15x15. Add a property `Color selectedColor` and when painting the component, just fill its entire area with this color. Give it a constructor that accepts

the initial value for `selectedColor`. When user clicks on this component, component must open color chooser dialog and must allow user to select color that will become new selected color. See:

http://docs.oracle.com/javase/7/docs/api/javax/swing/JColorChooser.html

Wire all necessary listeners in components constructor – this is behavior that the component is responsible for and not any outside component. Considering the `selectedColor`, this component must behave as a Subject in Observer pattern. So define a new interface for observers:

```
public interface ColorChangeListener {
  public void newColorSelected(IColorProvider source, Color oldColor, Color newColor);
}
```

In `newColorSelected` method, source is the component that has access to color. Define interface:

```
public interface IColorProvider {
  public Color getCurrentColor();
}
```

Modify `JColorArea` so that it implements this interface and offers selected color through `getCurrentColor()` method. Equip `JColorArea` with methods for observer registration and deregistration:

```
public void addColorChangeListener(ColorChangeListener l);
public void removeColorChangeListener(ColorChangeListener l);
```

When user changes the selected color, notify all registered listeners about the change. **Add single component to the JFrame's bottom** (for example, derive it from `JLabel`). This component must be a listener on both `JColorArea` instances. At all times, it must display text like this:

"`Foreground color: (255, 10, 210), background color: (128, 128, 0).`"

In parentheses are given red, green and blue components of the color.

Mutually exclusive buttons

Read:

http://docs.oracle.com/javase/7/docs/api/javax/swing/ButtonGroup.html
http://docs.oracle.com/javase/7/docs/api/javax/swing/JToggleButton.html
http://docs.oracle.com/javase/tutorial/uiswing/components/button.html

You must offer user three tools: drawing of lines, drawing of circles (no filling) and drawing of filled circles (area is filled with background color, circle is drawn with foreground color). Objects are added using mouse clicks. For example, if the selected tool is a line, the first click defined the start point for the line and the second click defines the end point for the line. Before the second click occurs, as user moves the mouse, the line is drawn with end-point tracking the mouse so that the user can see what will be the final result. Circle and filled circle are drawn similarly: first click defines the circle center and as user moves the mouse, a circle radius is defined. On second click, circle is added.

Right side of the window should be occupied by a list of currently defined objects. Each object must have its automatically generated name (an example is shown in given illustration; however, feel free to implement it any way you like).

Devise appropriate class hierarchy for modeling geometrical shapes so that this list can treat all object

equally. Assume the top class to be `GeometricalObject`. When user double-clicks an object in this list, a property dialog must appear. Model the content of the dialog as a single `JPanel` and then use `JOptionPane.showConfirmMessage` giving the panel as the message. For lines, user must be able to modify start and end coordinate and line color. For (unfilled) circle, user must be able to modify center point, radius and color; for filled circle user must be able to modify center point, radius, circle outline and circle area color.
Define an interface `DrawingModel` as follows:

```
interface DrawingModel {

  public int getSize();
  public  GeometricalObject getObject(int index);
  public void add(GeometricalObject object);

  public void addDrawingModelListener(DrawingModelListener l);
  public void removeDrawingModelListener(DrawingModelListener l);

}
```

Define `DrawingModelListener` as this:

```
public interface  DrawingModelListener {
  public void objectsAdded(DrawingModel source, int index0, int index1);
  public void objectsRemoved(DrawingModel source, int index0, int index1);
  public void objectsChanged(DrawingModel source, int index0, int index1);
}
```

In `DrawingModel`, graphical objects have its defined position and it is expected that the image rendering will be created by drawing objects from first-one to last-one (this is important if objects overlap).
Interval defined by `index0` to `index1` is inclusive; for example, if properties of object on position 5 change, you are expected to fire `objectChanged(..., 5, 5)`.

Implement the central component as a component derived from the `JComponent`; name it `JDrawingCanvas`. Register it as a listener on the `DrawingModel`. Each time it is notified that something has changed, it should call a `repaint()` method. When in process of adding a new object by mouse, after a final click occurs (so you know all parameters for the object), dont call the `repaint()`; instead, add the newly created object to the `DrawingModel` – it will fire `objectsAdded(...)` method and then you will call the `repaint()` as a response to that notification.

When creating the object list component, use `JList` with a custom made list model. Develop a `ListModel` (for this you can extend your model from abstract class `AbstractListModel` which already implements listener registration/deregistration/notification functionality) named `DrawingObjectListModel`. This model must not have its own list of object – drawing object are stored in `DrawingModel`. Implement `DrawingObjectListModel` to be an object adapter for the `DrawingModel`: it must store a reference to the `DrawingModel` and implement all methods in such a way that the information is retrieved from `DrawingModel`. Please note that you will want to make `DrawingObjectListModel` a listener on `DrawingModel`, so that, when user defines a new object by clicking in `JDrawingCanvas`, the `DrawingObjectListModel` can get a notification that the model has changed and that it can re-fire necessary notifications to it's own listeners (to the `JList` that shows the list of currently available objects).

Menus

Under the File menu you are required to implement several actions.

You must provide *Open*, *Save* and *Save As* actions. Each of these actions reads or writes a text file with extension `*.jvd`. An example for this file is given below.

```
LINE 10 10 50 50 255 255 0
LINE 50 90 30 10 128 0 128
CIRCLE 40 40 17.7 0 0 255
FCIRCLE 40 40 17.7 0 0 255 255 0 0
```

In file there is one row per object; space is used as element separator. Definitions of lines start with `LINE`, of unfilled circles with `CIRCLE` and of filled circles with `FCIRCLE`. The meaning of numbers is:

```
LINE x0 y0 x1 y1 red green blue
CIRCLE centerx centery radius red green blue
FCIRCLE centerx centery radius red green blue red green blue
```

For `FCIRCLE`, first three color components define outline color and last three color components fill color.

You also must provide an export action. When user selects export, you must ask him which format he wants (offer: JPG, PNG, GIF) and then ask him to select where he wants to save the image. You can make this in two steps, or you can immediately open save dialog with allowed extension `jpg`, `png` and `gif` and then determine what user selected by inspecting the file extension after the dialog is closed. Look at the objects in `DrawingModel` and find out the bounding box (the minimal box that encapsulates the whole image). Create an `BufferedImage` and the export procedure as follows:

```
BufferedImage image = new BufferedImage(
    box_width, box_height, BufferedImage.TYPE_3BYTE_BGR
);
Graphics2D g = image.createGraphics();
 … draw objects …
g.dispose();
File file = …;
ImageIO.write(image, "png", file);
Tell-user-that-images-is-exported.
```

For file formats (second argument) you must provide strings “png”, “gif” or “jpg”.

Lets see a simple example. Lets assume that the document model holds just a single line: (10, 100) to (100, 20). The bounding box is a box whose top-left corner is in (10, 20); it has width 100-10=90 and height 100-20=80. You would create an image 90x80 pixels. When drawing objects, you would translate all coordinates left for 10 pixels and up for 20 pixels. The idea is that the produced rendering corresponds to the interior of the bounding box, so that, if all of your object have x-coordinate grater than 100, you wont get 100 pixels of blank image. If you determine that bounding box has negative coordinates, this procedure will shift objects to the right so that exported image will again be OK.

And finally, you must provide “Exit” action that will check to see if `DrawingModel` has changed since the last saving; if so, you must ask user if he wants to save the image, cancel the exit action or reject the changes.

Problem 2.

Today, application internationalization (or so called i18n) is simply a must-have. As part of this homework you will bring this ability into your own JNotepad++ program. It is a simple text editor offering File: New, Open, Save, Save As, Exit and Edit: Cut, Copy, Paste, Delete selection actions. Ensure that when user tries to close it, program will ask the user what to do if there are unsaved modifications in document.

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);
			}
		});
	}
}
```

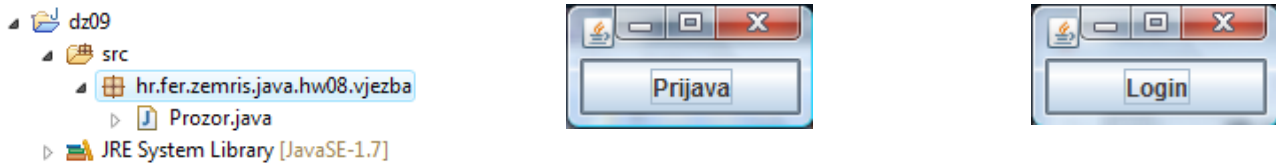
The state of Eclipse project is given on Figure 2.1a. The result when program is started by command:

```
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
```

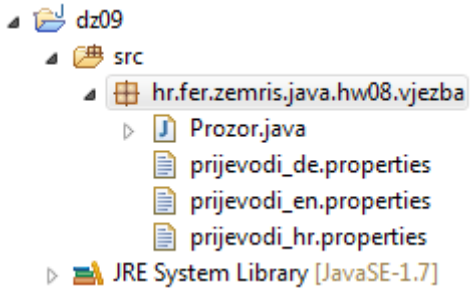
prijevodi_de.properties

```
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

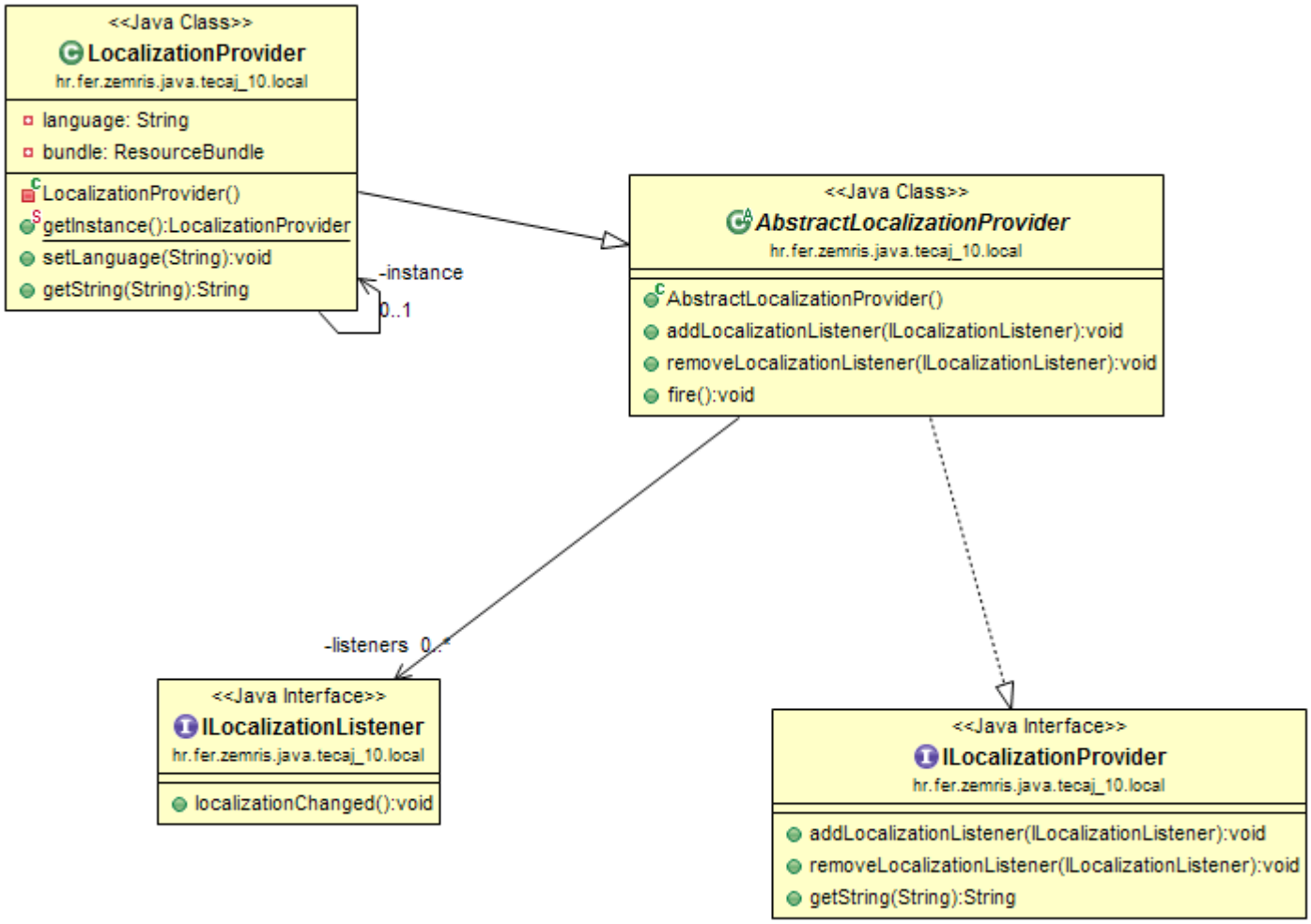
```
String getString(String key);
```

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");
}
```

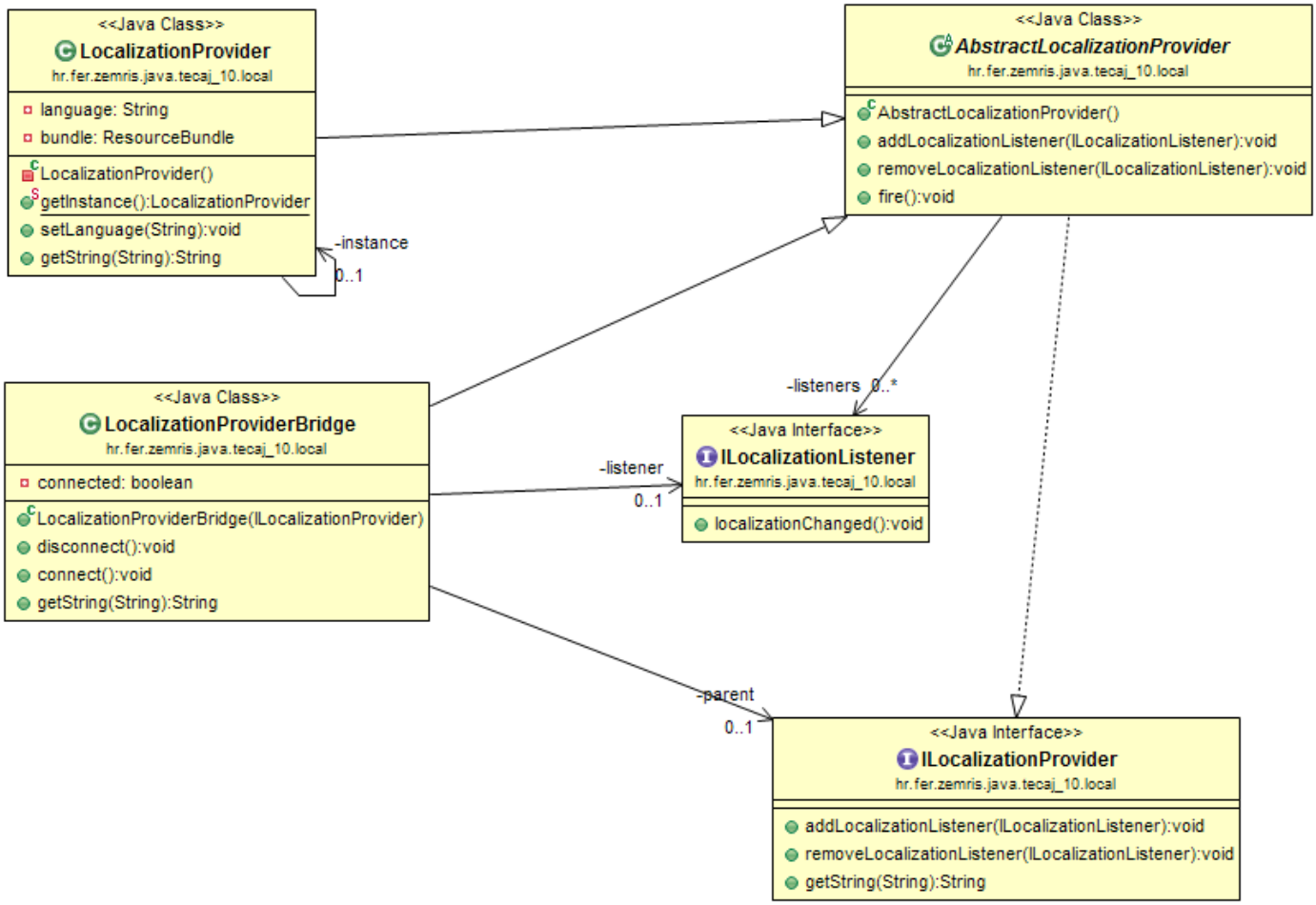
Now start the program; as you select different languages from menu, the text shown on button should automatically change.

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.

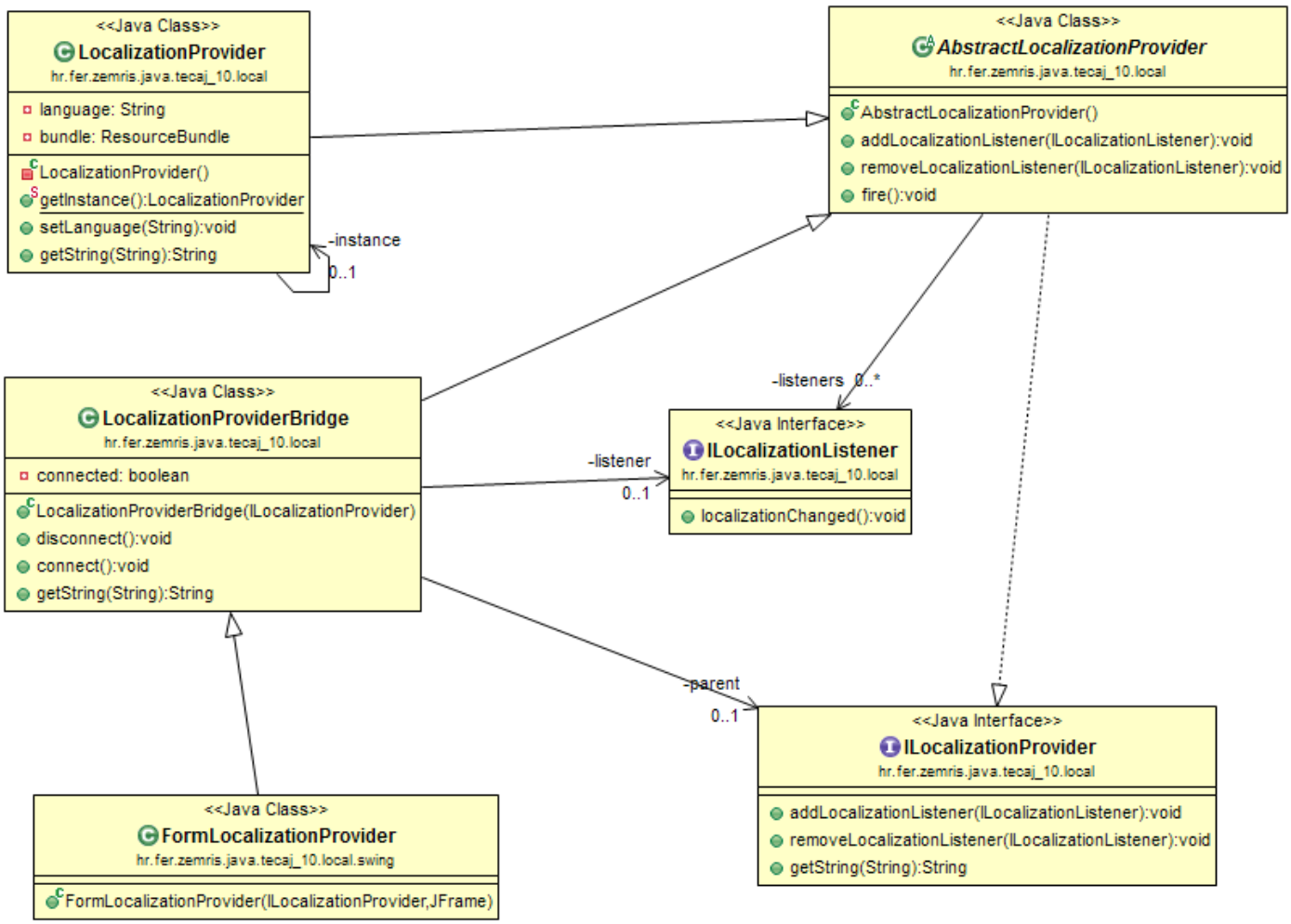
A solution of this problem is given on following class diagram.



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(...);
```

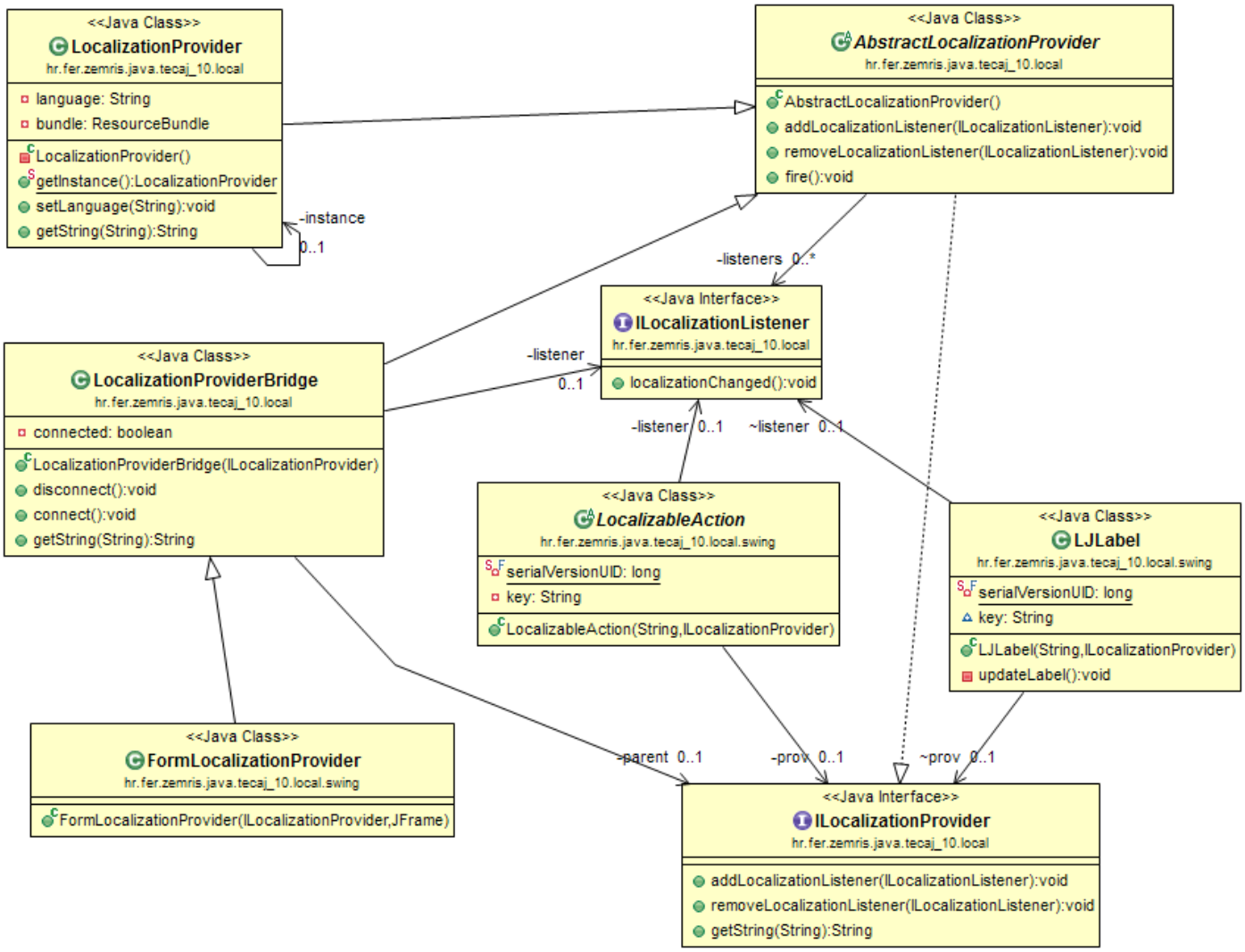
Change the way you construct button to this:

```
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.

Your job

Implement all of the previously described i18n infrastructure as part of your JNotepad++. You must add a localization to your notepad: a Languages/Jezici/Sprache menu and a selection of languages (at least two: Hrvatski/Croatian and Engleski/English).

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). Document your code!

In order to solve this homework, create a blank Eclipse Java Project and write your code inside. Once you are done, export project as a ZIP archive and upload this archive on Ferko before the deadline. Do not forget to lock your upload or upload will not be accepted.

Equip the project with appropriate `build.xml`. You must support two run targets. Target `run1` must start the solution of the first problem: a `JVDraw` program. Target `run2` must start a fully localizable version of `JNotepad++`.

You **are not required** to create any unit tests in this homework, except for one test (test whatever you want – it is important that there exists at least one test so that build process which involves generation of reports does not fail). Of course, you are not forbidden to utilize unit tests.

Before uploading, please make sure that your project can be started from console by ant and that the i18n works when started that way!

You will have to **upload two separate** files:

1. A ZIP archive with your homework.
2. A PDF file with class diagrams which you created **before** you stared programming 1. part of this homework: the idea is to spend some time and think about how you will design your application, and then to start coding. Do not generate this diagrams after your homework is done.

Since parts of this homework have been developed during several years, on various images and throughout this document you will find in some package names `hw08` (or similar) or that the project is named `dz09`. **You must ignore this and use subpackage `hw11`**, and your project name must be `HW11-`*yourJMBAG*.

The deadline for uploading and locking this homework is May, 26th 2014. at 08:00 AM (morning!).