Making of a Typing Test Application

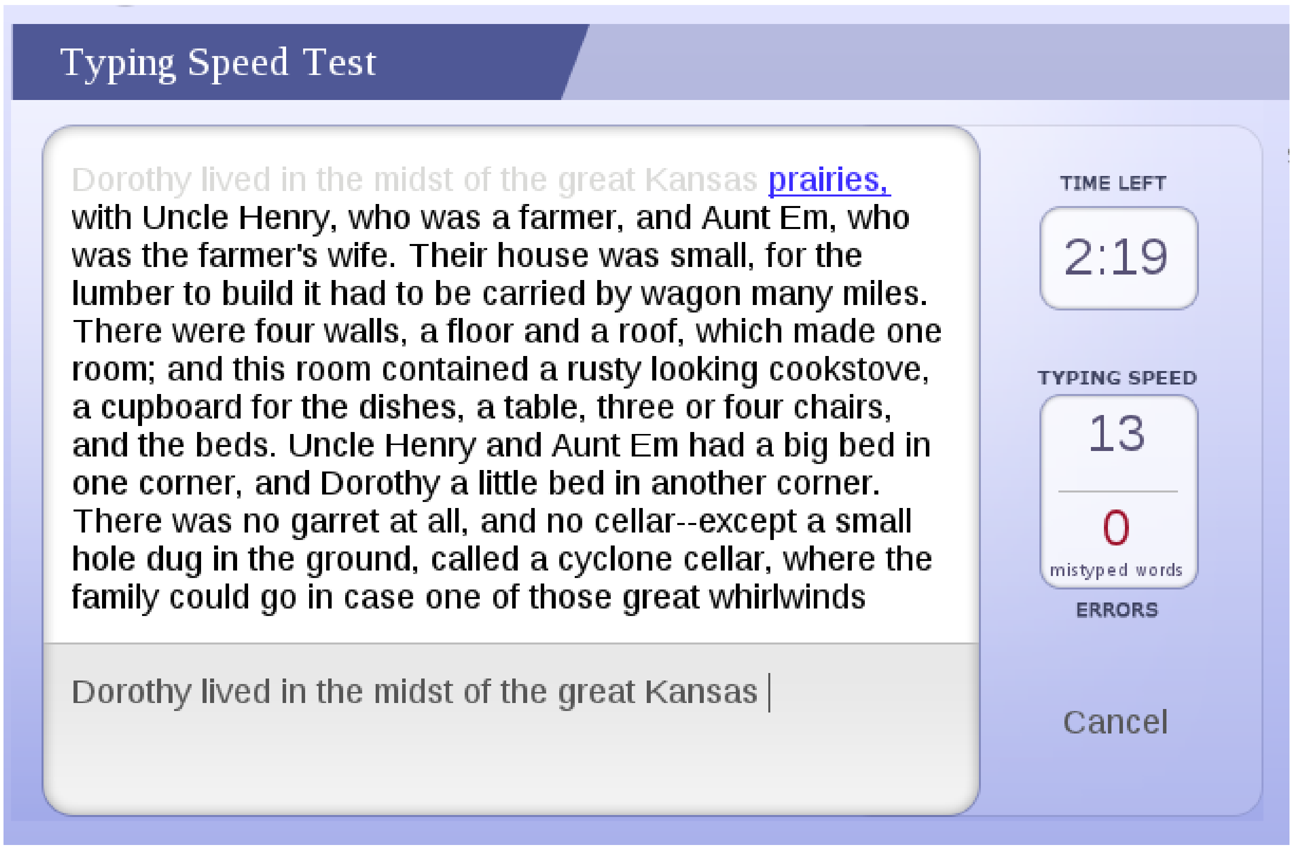
Index

1. Analysis of Requirements.

Ok, so we've got a task to make a simple typing test program similar to few existing examples, which would fit the purpose of assessing a user's speed and accuracy of typing. The program may be a web or a PC application, compatible with Windows operating system.

We have few more distinguishable requirements for a ready application:

- A typing exercise happens like: the user **sees** an exercise text and **retypes** the same in a special input editor pane.
- The user can **select** any out of few **sample texts** to type.
- The exercise is **time**-bounded.
- The user can **select the time limit** for the exercise before she or he begins typing.
- The exercise **ends** when either of the following events happens: (1) the user types the sample text **completely**; (2) the user chooses to **abort** the exercise; or (3) time **runs out**.
- If the exercise wasn't aborted, the user is then presented with **evaluation** of her or his typing speed and accuracy, measured in amount of characters or words typed wrongly.


Typing Speed Test

Dorothy lived in the midst of the great Kansas prairies, with Uncle Henry, who was a farmer, and Aunt Em, who was the farmer's wife. Their house was small, for the lumber to build it had to be carried by wagon many miles. There were four walls, a floor and a roof, which made one room; and this room contained a rusty looking cookstove, a cupboard for the dishes, a table, three or four chairs, and the beds. Uncle Henry and Aunt Em had a big bed in one corner, and Dorothy a little bed in another corner. There was no garret at all, and no cellar--except a small hole dug in the ground, called a cyclone cellar, where the family could go in case one of those great whirlwinds

Dorothy lived in the midst of the great Kansas

Pic 1: That is one of the example programs working. Features observable: two panes for sample text and the text to type (one which has the input focus), countdown time counter, typing speedometer and a mistyped words counter; the Cancel button.

As none of the above tells that the program should in any way authorize or remembers users and their performance, or have an extendable configuration, we may throw out considerations of making any kind of an application requiring persisting of a state. We can just produce a standalone PC application or a code running within the browser doing the same.

Given that, a good choice for me (fitting my development experience) is to make a Java application with a GUI based on Swing framework. This is also leaves a possibility to turn the desktop application into an applet running in a web browser window with a small effort.

However, to make this endeavor more interesting, I've chosen to set myself a set of extra requirements, not conflicting with the base requirements described above. They are:

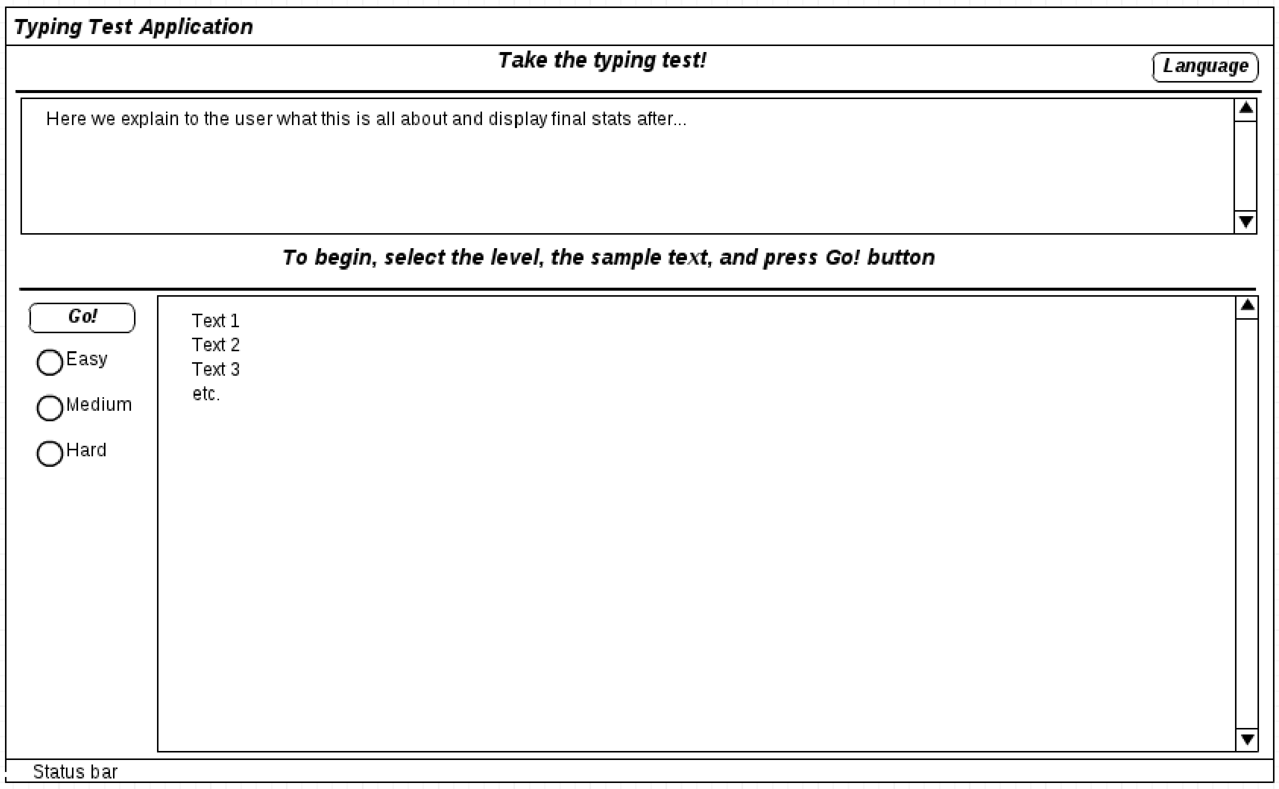
- The user's **statistics** will include: winning status (depends on whether the user got to the end of text in time, or didn't), time spent for the exercise, numbers of typed words and characters, numbers of mistyped (wrongly typed) words and characters.
- The program must recognize words in the text and be able to **confirm order and correctness** of words in the typed text. All the usual separator characters occurring in fiction book texts should be counted apart from words and considered separators.
- All kinds of the following is considered **mistyped separators**: missing characters, unexpected characters, misplaced characters.
- The application must be **forgiving** to character and letter interchanging which is allowed in the real typing practice according to common conventions. For example, it should not count a penalty for using Russian letter “e” instead of “ë”, or straight quotes (" ") instead of Guillemets (« ») or the literary quotation marks (“ ”).

- The text typing quality evaluation must be done **in real time**, allowing the user to instantly see if any mistake is made, giving her or him an opportunity to quickly fix it (a noticeable number of errors appears or changes in that case).
- To help users who are good at touch typing, a visual cue shall be provided, in form of **highlighting the next word** to type. This, naturally, involves a real-time recognition of where the user currently is through his typing exercise.
- The functions depending on tracking of the user's progress should be reasonably **fool-proofed**, they must provide a help for those who are typing a text with a sporadic typo here and there, and stay safe from failure in case anyone just crazily bounces on the keyboard: a stream of gibberish on input must be profoundly recognized as such.
- As a little modification to the original rules, I'd like to put a little style of game into this application and instead of explicit setting of the time to do an exercise, offer the user a possibility to choose between of three **levels**: “Easy”, “Medium”, and “Hard”, thus picking a speed of the “virtual competitor” whose progress will tick during the test. Two progress bars side by side will allow to see instantly who is having an upper hand at the moment, the virtual competitor, or the user. The tempo of the competitor's progress is defined by the picked game level, the more serious the level, the greater the expected typing speed is.
- In addition, the interface should have optional **Russian localization**, because I feel best at fast typing Russian texts, and it's against any logic to fit in Russian texts and not to include Russian localization.

2. The GUI design.

Let's first draft how the UI should look. I use http://creately.com online tool for sketching of UI mock-ups. As it's evident from the given application example and from studying the requirements that the application should have at least two distinct screens: one to set up the exercise and another to run it.

So the next picture represents my idea of what the initial screen of the application should look like:


Typing Test Application
Take the typing test!
Language
Here we explain to the user what this is all about and display final stats after...
To begin, select the level, the sample text, and press Go! button
Go!
Easy
Medium
Hard
Text 1
Text 2
Text 3
etc.
Status bar

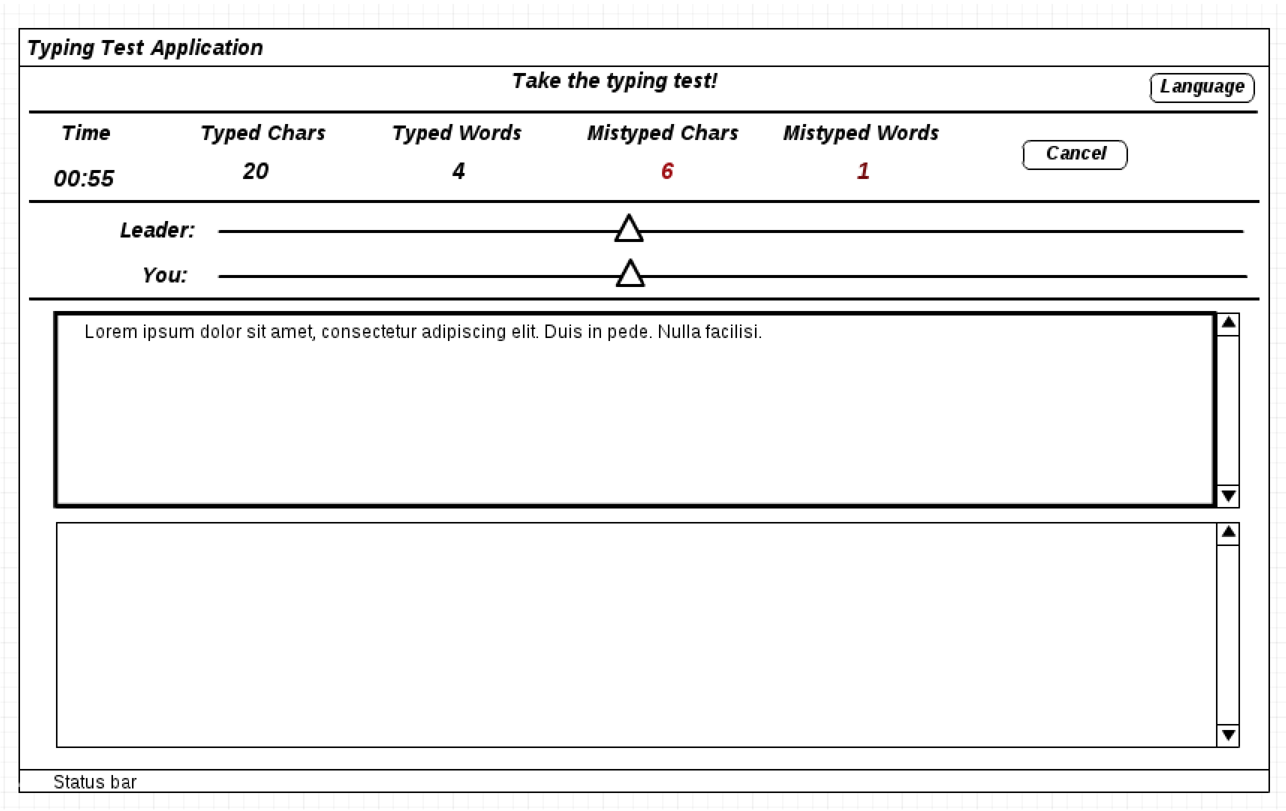
Pic 2: A sketch of the UI screen for the typing exercise setup.

This initial screen has four functional zones: the header with the language toggle button aligned to right edge; The information text pane to be filled with suitable content, depending on the current state.

Next, the bottom right corner is occupied by a scrollable list of available sample texts. The user is supposed to click on one of them (single line selection mode should be supported) and then pick the “game level” in the bottom left pane (any one radio button is selected at any time). Then she or he clicks the “Go!” button, and it takes her or him to the exercise screen.

Finally, the status bar text area is reserved for better look, convenience of resizing the window, and as a space for displaying various messages, including debugging.

In the exercise mode, the screen should look like at the next picture.



Pic 3: A sketch of the UI screen for the exercise mode.

As we can see, I prefer to reuse the upper part of the initial screen with the language switch button, but the lower part is significantly different. Going top to bottom, it has:

- The set of display **gauges**, ticking in the real time. Their meaning is self-explained by the title headers. Rightmost in this line is the **Cancel** button, also bound to Escape key; pressing it will pause the exercise and offer the user a choice to either continue with typing or go back to the initial screen.
- Two **progress bars** showing current progress of the virtual leader (or competitor, but the word Leader is better for brevity), and the user's. Alas, the sketching web tool didn't have a primitive for a progress bar, so I've used a slider here, but in reality I want a progress bar.
- The **sample text** pane. It displays the selected sample text and has the following sub-features:
    - User can't select or copy the text from it.
    - The next word the user is expected to type, is highlighted here.
    - The content is automatically scrolled, so that the next word the user is expected to type is always visible.
- The **typing text** pane. This is where the user is typing the text. She or he must not be able to paste any text in (yet why forbid text copying or cutting from it)? It's also useful to let user have a possibility to slide the divisor between the upper and lower panes manually for convenience.

3. Getting down to work... (making the GUI view part).

As this guide to building the application is not a formal paper for some waterfall style project, let's be agile and have the ground turned with our development before further elaboration of the design is provided.

Note: I'm using a Linux-based system, but the Java applications are inherently cross-platformed, so in the end we'll have a Windows-ready executable code.

However, all outputs listed here will give away my platform...

We need the following tool set:

1. Java Development Toolkit (JDK) from Oracle (http://www.oracle.com/technetwork/java/javase/downloads/index.html). At the moment of writing, the latest release is 8u25, so I've used this. Follow the instructions for installation to your target platform. To check that your installation is okay, use command:

```
$ java -version

java version "1.8.0_25"
Java(TM) SE Runtime Environment (build 1.8.0_25-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.25-b02, mixed mode)
```

2. Maven build and integration tool (http://maven.apache.org/download.cgi). Again, get the latest version and install on your platform as described. After installation and environment setup, you should have an okay version command output:

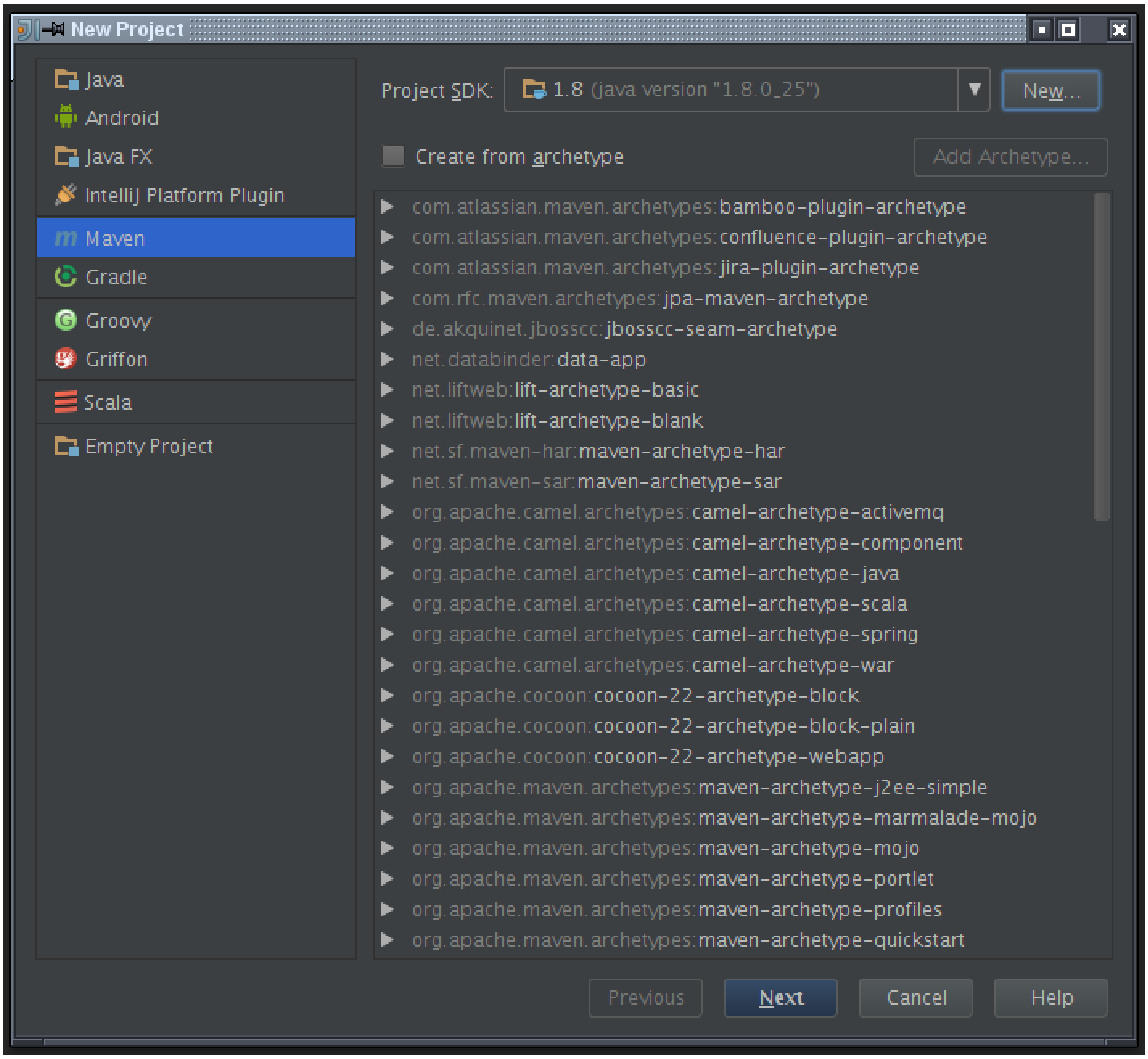
```
$ mvn -version

Apache Maven 3.2.5 (12a6b3acb947671f09b81f49094c53f426d8cea1; 2014-12-
14T20:29:23+03:00)

Maven home: /usr/local/apache-maven

Java version: 1.8.0_25, vendor: Oracle Corporation

Java home: /usr/local/jdk1.8.0_25/jre

Default locale: en_US, platform encoding: UTF-8

OS name: "linux", version: "2.6.32-431.11.2.el6.x86_64", arch: "amd64",
family: "unix"
```

3. A suitable Java development IDE. I think the IntelliJ Idea is great and has all the features we need in its Community edition, free for non-commercial use. Go grab it at http://www.jetbrains.com/idea/download/.

4. To keep the changes of your code stored, you may need to employ any of the popular version control systems, like Git or Mercurial, but I won't describe how to install and use them.

We are going to make a maven-built project which uses a Swing framework interface designed and instrumented in Idea's **UI Designer** tool. This approach is controversial, because it hides part of the UI content creation code from the developer and requires IntelliJ libraries to be attached to the project, but I'm fine with it, because it allows a quick and easy interactive drawing and property setting for the UI widgets.

Run IntelliJ idea and create an new Maven project in *File – New Project* menu. Pick an SDK for the project (the JDK you've installed). Don't select any archetype, for this kind of application we need the default structure.


New Project

Java
Android
Java FX
IntelliJ Platform Plugin
Maven
Gradle
Groovy
Griffon
Scala
Empty Project

Project SDK: 1.8 (java version "1.8.0_25")

Create from archetype

```
com.atlassian.maven.archetypes:bamboo-plugin-archetype
com.atlassian.maven.archetypes:confluence-plugin-archetype
com.atlassian.maven.archetypes:jira-plugin-archetype
com.rfc.maven.archetypes:jpa-maven-archetype
de.akquinet.jbosscc:jbosscc-seam-archetype
net.databinder:data-app
net.liftweb:lift-archetype-basic
net.liftweb:lift-archetype-blank
net.sf.maven-har:maven-archetype-har
net.sf.maven-sar:maven-archetype-sar
org.apache.camel.archetypes:camel-archetype-activemq
org.apache.camel.archetypes:camel-archetype-component
org.apache.camel.archetypes:camel-archetype-java
org.apache.camel.archetypes:camel-archetype-scala
org.apache.camel.archetypes:camel-archetype-spring
org.apache.camel.archetypes:camel-archetype-war
org.apache.cocoon:cocoon-22-archetype-block
org.apache.cocoon:cocoon-22-archetype-block-plain
org.apache.cocoon:cocoon-22-archetype-webapp
org.apache.maven.archetypes:maven-archetype-j2ee-simple
org.apache.maven.archetypes:maven-archetype-marmalade-mojo
org.apache.maven.archetypes:maven-archetype-mojo
org.apache.maven.archetypes:maven-archetype-portlet
org.apache.maven.archetypes:maven-archetype-profiles
org.apache.maven.archetypes:maven-archetype-quickstart
```

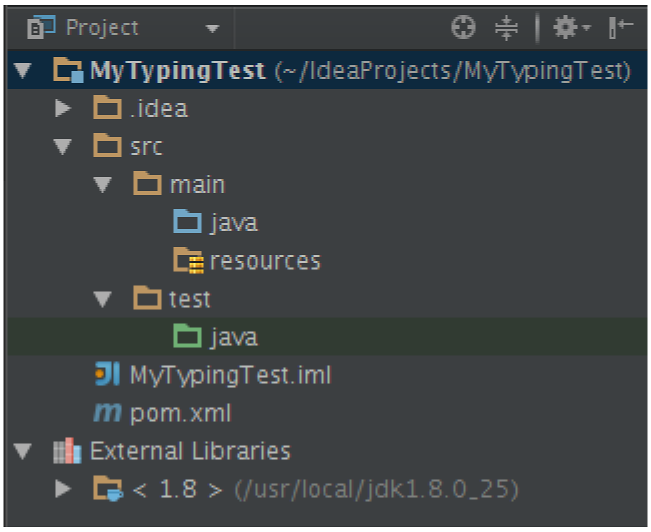
Pic 4. Creating a new Maven project in IntelliJ Idea.

Specify the required groupId, artifactId and version for your project. I've used:

```
<groupId>org.krasnyansky</groupId>
<artifactId>typetest</artifactId>
<version>1.0-SNAPSHOT</version>
```

Agree to enable automatic import of maven project (in the pop-up that appears at that stage, or later, at *Settings – Build, Execution, Deployment – Build Tools – Maven – Importing* – [v] Import Maven Projects Automatically). This way, changes in your .pom file will trigger Maven imports.

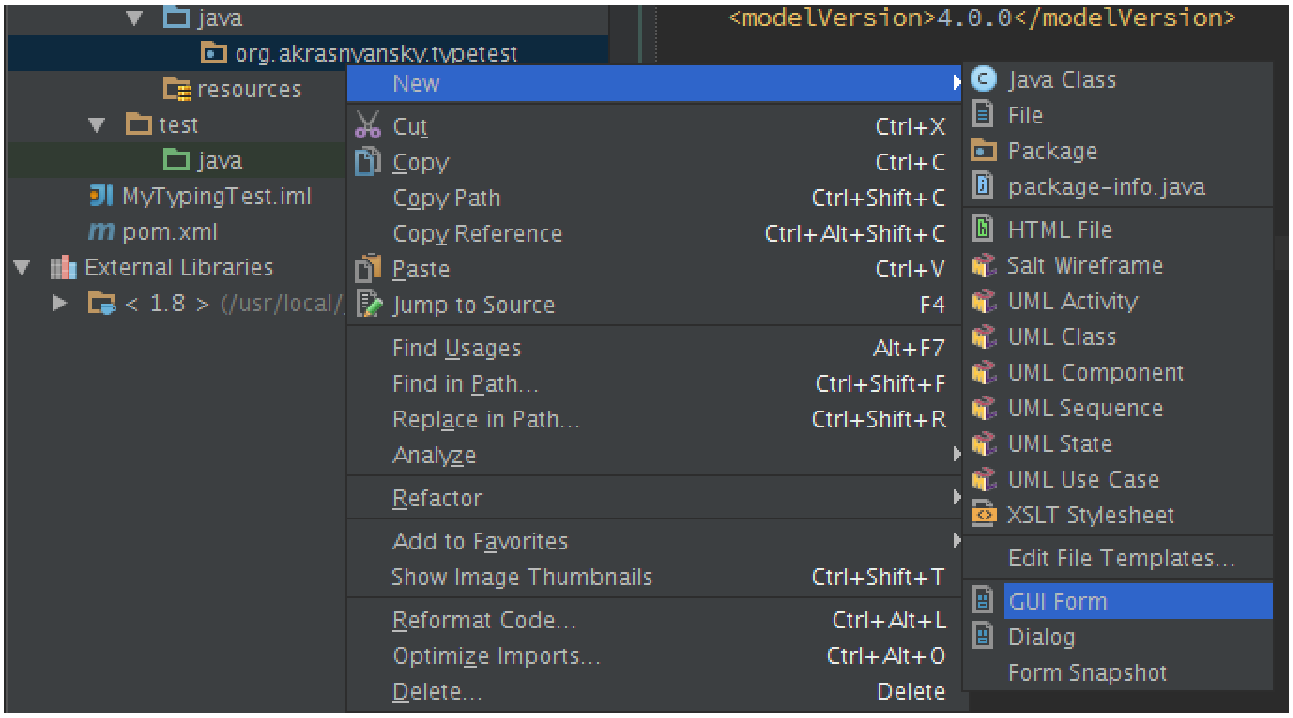
A bare bone project's structure looks like:



Pic 5. An empty project.

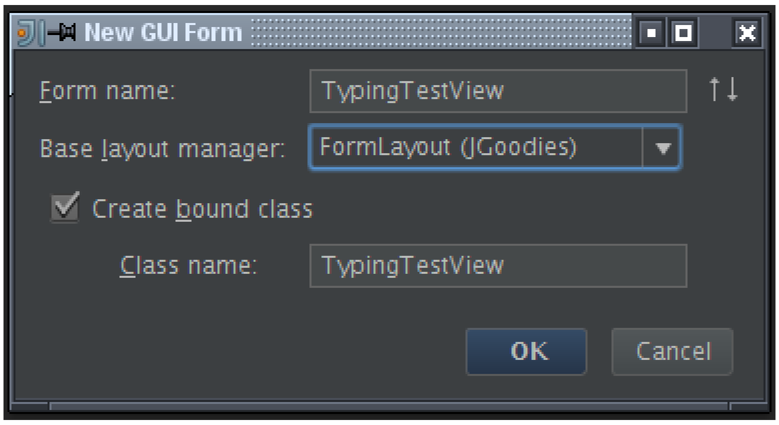
Next, create a package in `src/main/java` folder. I named it `org.akrasnyansky.typetest`, yours may have a different name.

Right-click the new package and create a GUI form in it:



Pic 6. GUI Form in the pop-up menu tree.

Next up, choose a name for the form, for the bound Java class (best to have the same name) and a base layout manager. I recommend using *FormLayout (JGoodies)*, as it provides the highest flexibility in widgets composition at the form.



Form name: TypingTestView

Base layout manager: FormLayout (JGoodies)

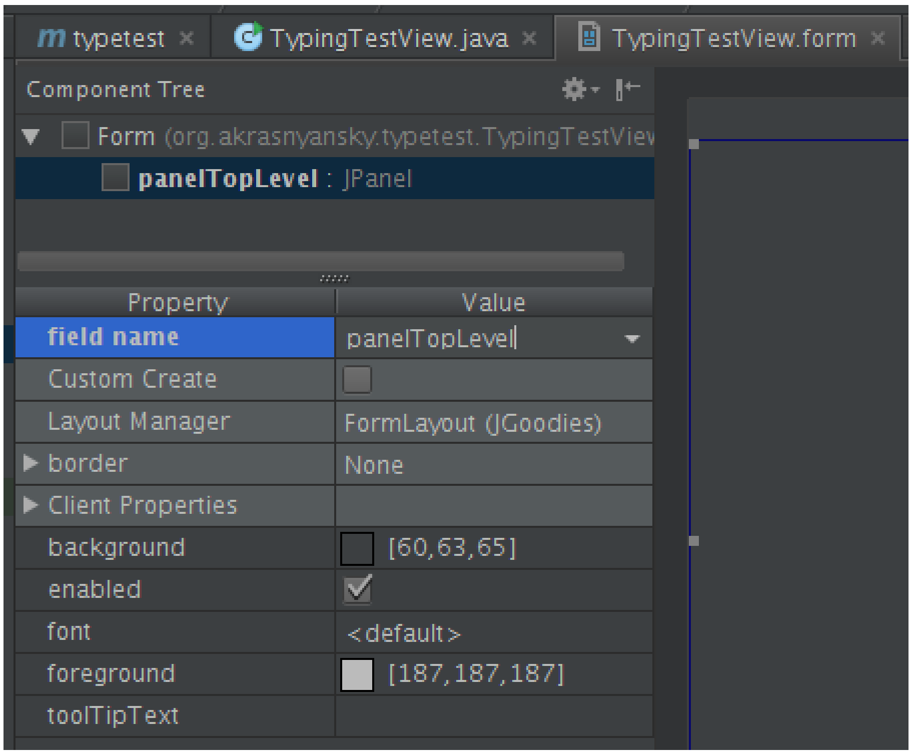
☑ Create bound class

Class name: TypingTestView

Pic 7. Form creation setup.

After you do that, you have an empty form and a supporting class with the same name, which we'll make responsible for the view tier of the application.

At the scratch start, you have a `Form` bound to `TypingTestView` class and a single `JPanel` inside it. This `JPanel` component will be the root of Swing component container hierarchy, so you have to give a name to it. In the form editor, type in the new name into the *field name* editor (let it be **panelTopLevel**):



Form (org.akrasnyansky.typetest.TypingTestView

panelTopLevel : JPanel

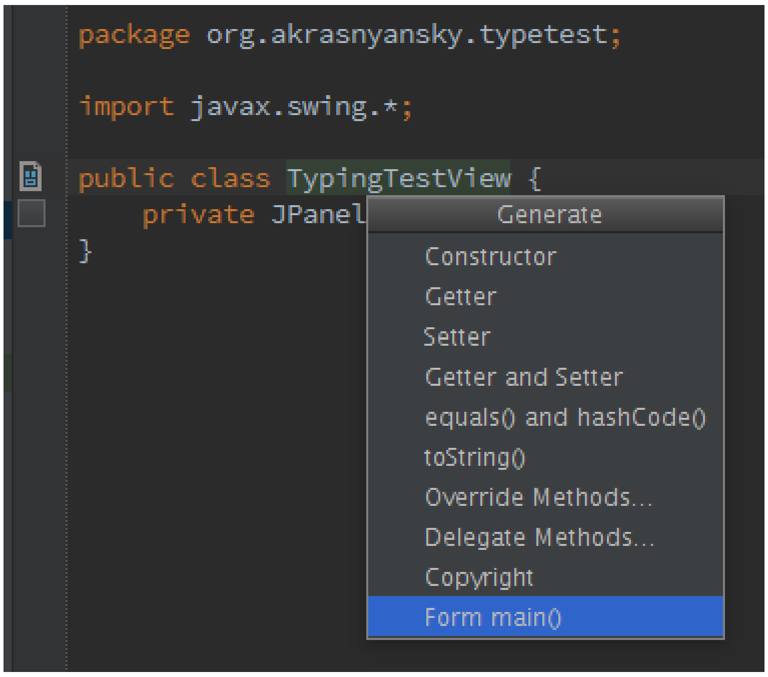
Property	Value
field name	panelTopLevel
Custom Create	☐
Layout Manager	FormLayout (JGoodies)
border	None
Client Properties	
background	■ [60,63,65]
enabled	✓
font	<default>
foreground	□ [187,187,187]
toolTipText	

Pic 8. Editing a property of JPanel.

Your supporting class' code then becomes:

```
package org.akrasnyansky.typetest;
import javax.swing.*;
public class TypingTestView {
	private JPanel panelTopLevel;
}
```

Idea can create the basic boilerplate code for starting up the visual application for you with the embedded code generation wizard. Go to the class code, right-click the class name (or press *Alt+Insert*) and pick a *Form main()* option:



```
package org.akrasnyansky.typetest;

import javax.swing.*;

public class TypingTestView {
    private JPanel
}
```

Pic 9. Code auto-generation.

The class now has the basic initialization code, and is now runnable:

```
package org.akrasnyansky.typetest;
import javax.swing.*;
public class TypingTestView {
      private JPanel panelTopLevel;
      public static void main(String[] args) {
            JFrame x_frame = new JFrame("TypingTestView");
            x_frame.setContentPane(new TypingTestView().panelTopLevel);
            x_frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            x_frame.pack();
            x_frame.setVisible(true);
      }
}
```

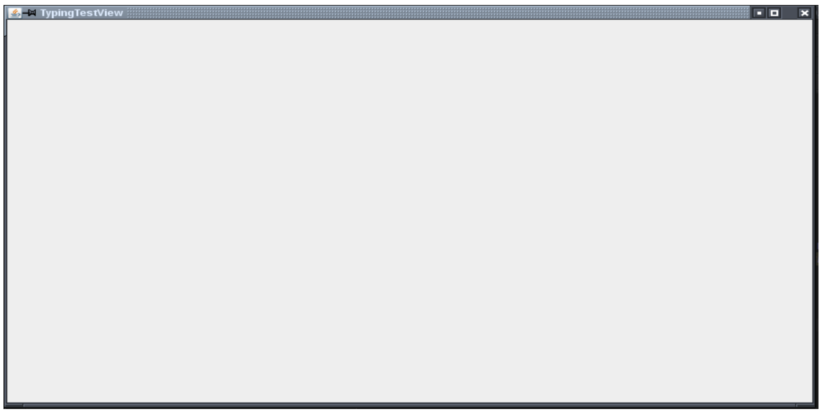
We need two more steps to set the stone rolling. In the form editor, select `panelTopLevel`, and check *"Show expert properties"* at the bottom, so that *preferredSize* property becomes visible. Change it from the default `[0, 0]` to a reasonable size for all our components to place, for example `[850, 520]`.

Next, edit your *pom.xml* file and add there JGoodies dependency (and dependencies section which is still missing):

```
<dependencies>
      <dependency>
            <groupId>com.jgoodies</groupId>
            <artifactId>forms</artifactId>
            <version>RELEASE</version>
      </dependency>
```

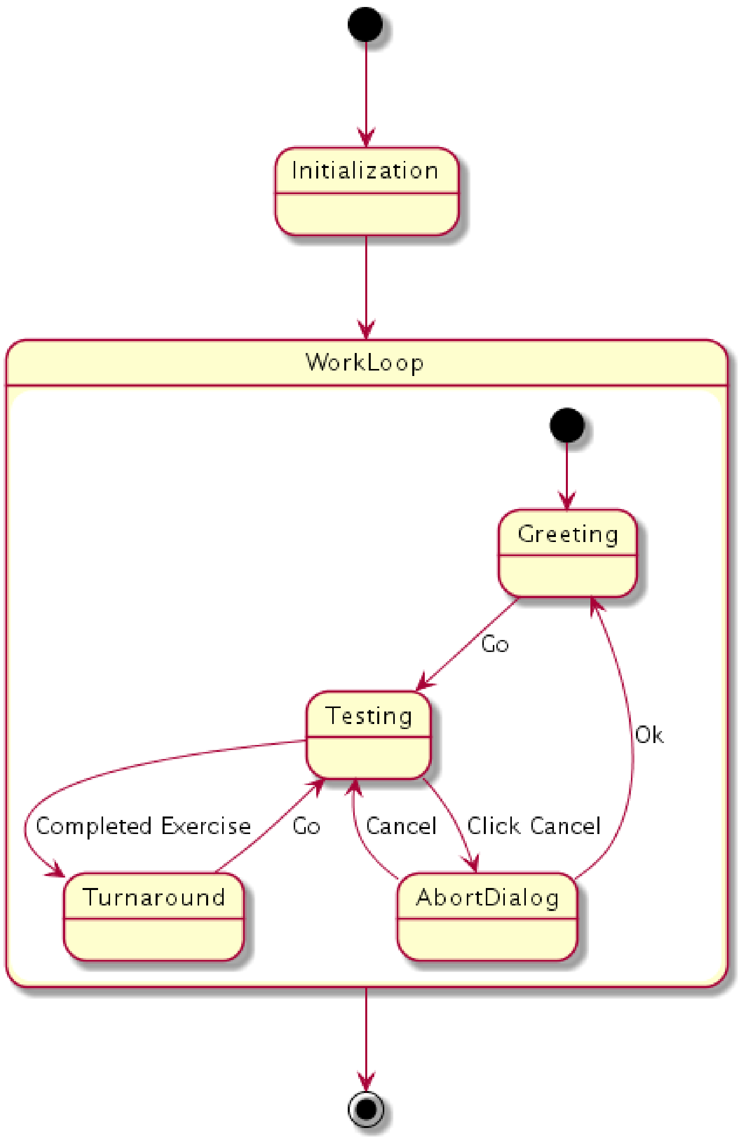
```
</dependencies>
```

After that, you can right-click and “Run...” `TypingTestView` class, and it will show you the empty form:



Pic 10. The new form.

Hooray! :-) Having this initial success, let's move on with creating the user interface. First, I'm laying down the life cycle of the program in form of a state diagram:



Pic 11. State diagram for Typing Test application.

Initialization phase is required to non-interactively create the program and load resources, while in the interactive work loop the user first goes to the *Greeting* screen (see pic. 2), then she or he begins *Testing* exercise (see pic. 3), which may be either completed or aborted from *AbortDialog* (which also has the way back to unpause and continue the exercise). In the *Turnaround* phase, reachable upon a successful completion of *Testing*, the user can see the exercise summary and can go back to *Testing*. Any state can be immediately terminated with window close cross button, and no special tear-down is required.

We will reuse the widgets used in *Greeting* and *Turnaround* phases, but the content for *Testing* is different, and we should be able to swap visibility between them. Swing allows to change visibility mode of a part of GUI by means of a special container layout manager called *CardLayout* (refer to A Visual Guide to Layout Managers). We'll use it for one container with interchangeable content, and we'll use *JGoodies FormLayout* everywhere else throughout the container hierarchy.

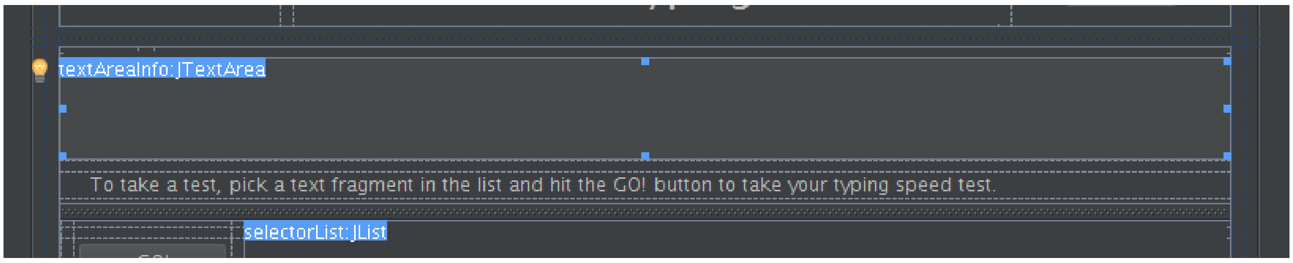
Please regard the following **component hierarchy tree**, which includes the containers of the application and the most important widgets (for the entire tree, load the source of TypingTest application attached to this description). Underlined are the components to which we bind listeners (described later). Each line begins with a name of the component, and in parentheses there are class name for the component and the assigned layout manager's name, where applicable):

- `Form (TypingTestView)`
    - `panelTopLevel (JPanel, FormLayout)`
        - `panelTopHeader (JPanel, FormLayout)`
            - `buttonToggleLanguage (JButton)`
        - `panelCardContent (JPanel, CardLayout)`
            - `panelInitialContent (JPanel, FormLayout)`
                - `splitterInitial (JSplitPane)`
                    - `panelResultsAndPrompt (JPanel, FormLayout)`
                        - `scrollPaneInfo (JScrollPane)`
                            - `textAreaInfo (JtextArea)`
                    - `panelGoAndSelector (JPanel, FormLayout)`
                        - `buttonGo (JButton)`
                        - `radioLvlEasy (JRadioButton)`
                        - `radioLvlMedium (JRadioButton)`
                        - `radioLvlHard (JRadioButton)`
                        - `scrollPaneTextSelector (JScrollPane)`
                            - `selectorList (JList)`
            - `panelWorkingContent (JPanel, FormLayout)`
                - `buttonCancelTest (JButton)`
                - `labelCharTypos (JLabel)`
                - `labelWordsTypos (Jlabel)`
                - `progressBarLeader (JProgressBar)`
                - `progressBarYou (JProgressBar)`
                - `splitterWorking (JSplitPane)`
                    - `scrollPaneExample (JScrollPane)`
                        - `textAreaExampleText (JTextArea)`
                    - `scrollPaneTyping (JscrollPane)`
                        - `textAreaTyping (JtextArea)`
        - `labelStatus (JLabel)`

In context of `panelCardContent`, there are two “card names” defined, `panelInitialContent` has name **cardInitial** in its *Card Name* property, while `panelWorkingContent` has the same as **cardWorking**. Either one is active at the same time.

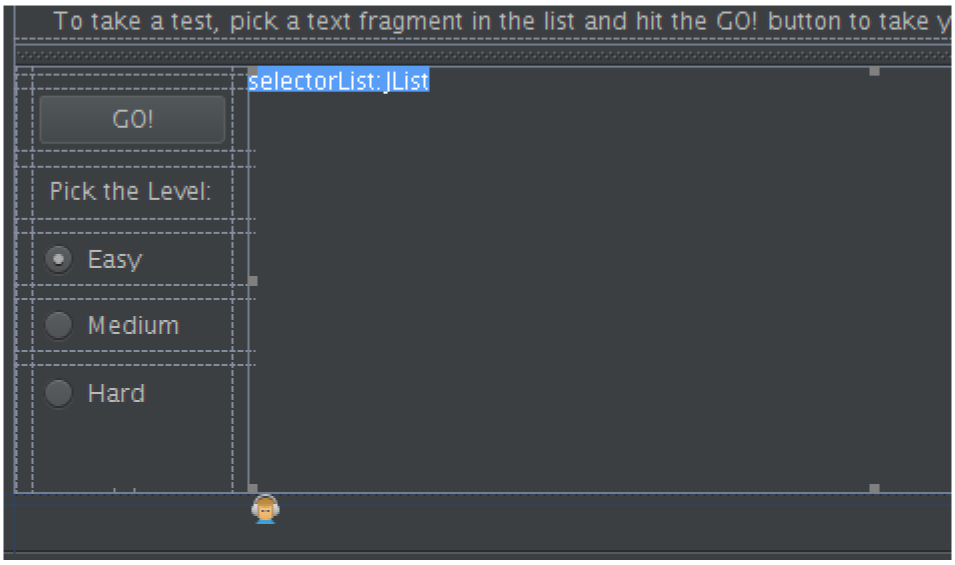
One more binding between components, served by Swing framework behind the scenes, is joining of three radio buttons (`radioLvlEasy`, `radioLvlMedium`, `radioLvlHard`) into a radio group `groupLevelPick` (associated with each of the button's property *Button Group*). Once we fill that in at the property editor, we may stop caring about only one of the radio buttons being selected at any time.

In this tree, the main **output components** are various labels with either static or updatable text (not described here for brevity, because their usage is obvious); `textAreaInfo` belonging to the `panelInitialContent`, where the program displays the rules and test results:


textAreaInfo: JTextArea
To take a test, pick a text fragment in the list and hit the GO! button to take your typing speed test.
selectorList: JList

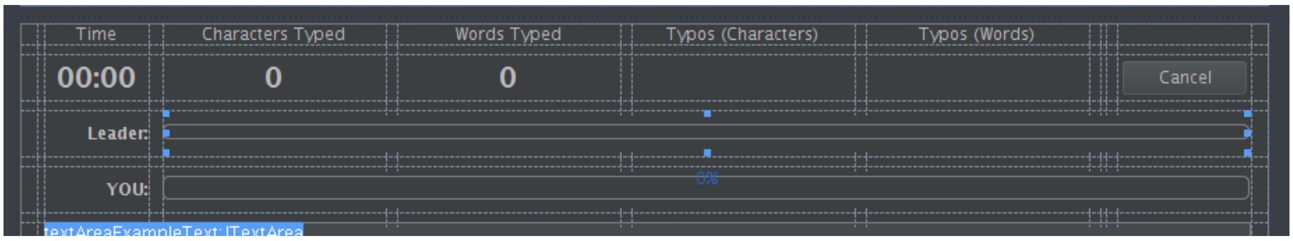
Pic 12. `textAreaInfo` in the form editor.

Next is the `selectorList` in the same pane, it serves for both output (presenting the list of sample texts) and for input (selecting one of them):


To take a test, pick a text fragment in the list and hit the GO! button to take y
selectorList: JList
GO!
Pick the Level:
Easy
Medium
Hard

Pic 13. `selectorList, buttonGo,` and radio buttons in the form editor. The visual cue looking like the head in headphones means that a listener is defined for the selected component.

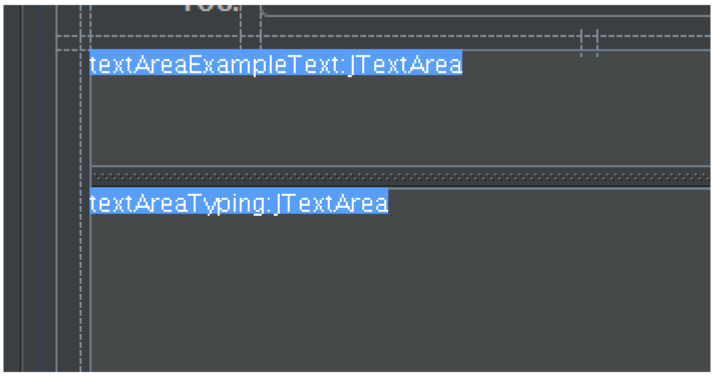
Inside `panelWorkingContent` we have four big output features. First, this is couple of progress bars we'll bind to progress percentage of the virtual leader and the user:



Pic 14. Top of the `panelWorkingContent` with labels and progress bars.

Note that numerical labels under Typos (Characters) and Typos (Words) are not visible, but they are there. This is just a result of setting their *visible* property to false, but they are still selectable in the component tree.

Finally, there are two more `JTextAreas` put into this pane, the upper is for displaying the sample text and the lower is for user's input. Note the grip handle in-between, it's a result of being both components put inside the same `JSplitPane` (`splitterWorking` in this case).

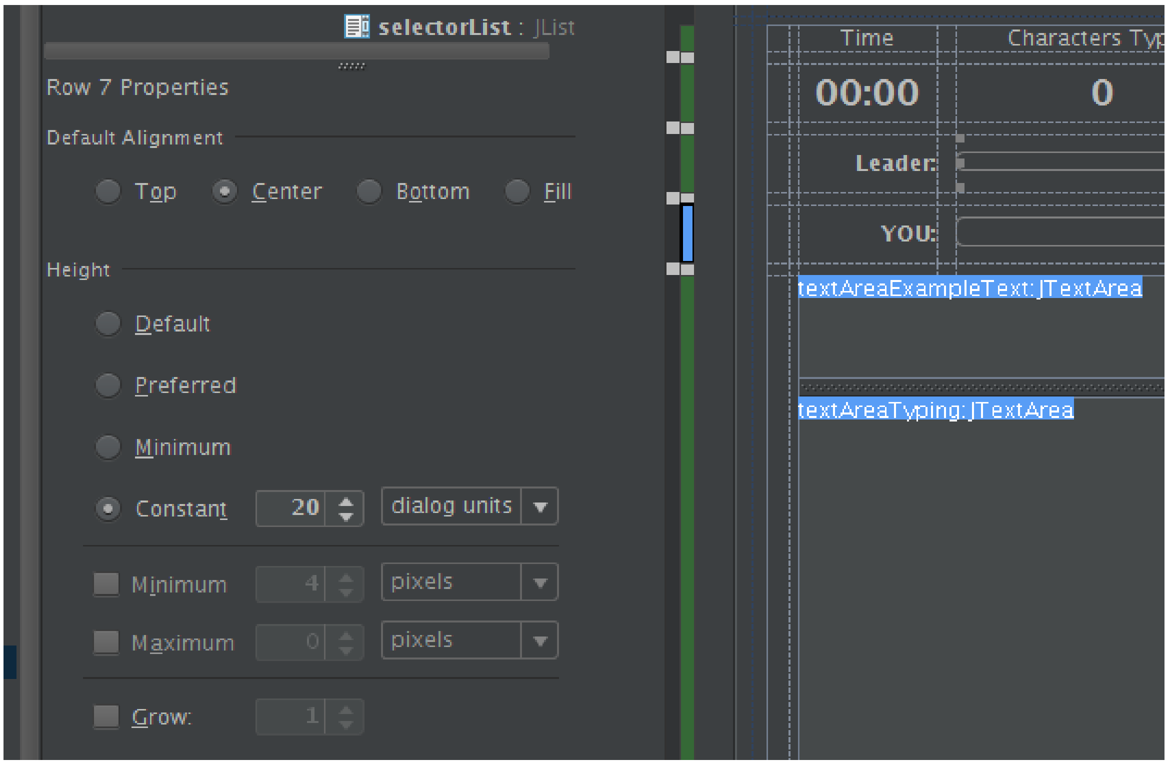


Pic 15. textAreaExampleText and textAreaTyping.

Also please go back to component tree description and note that all big textual components are surrounded by individual `JScrollPanes`. If we also set *autoscrolls* property to true for these components, this lets Swing to handle appearance and disappearance of scroll bars automatically whenever the user resizes the form or the content size is changed.

The **input components** include already mentioned `selectorList`, `goButton`, radio buttons (see pic. 13) `toggleLanguageButton`, `textAreaTyping`.

Few points on how *JGoodies FormLayout* is used to design a component layout. The idea of this layout manager is placing components in cells of a grid spanning the parent pane. Components may occupy few adjacent cells and align differently with relation to the cell boundaries (to edges, to center, fill by horizontal or vertical axis). Grid elements are rows and columns visible along the edges of the form editor when you select a parent pane or any child component. If you want a constant size of a row, this is defined like at the next picture:


selectorList : JList

Row 7 Properties

Default Alignment

Top [✓ Center] Bottom Fill

Height

Default

Preferred

Minimum

[✓ Constant] 20 dialog units

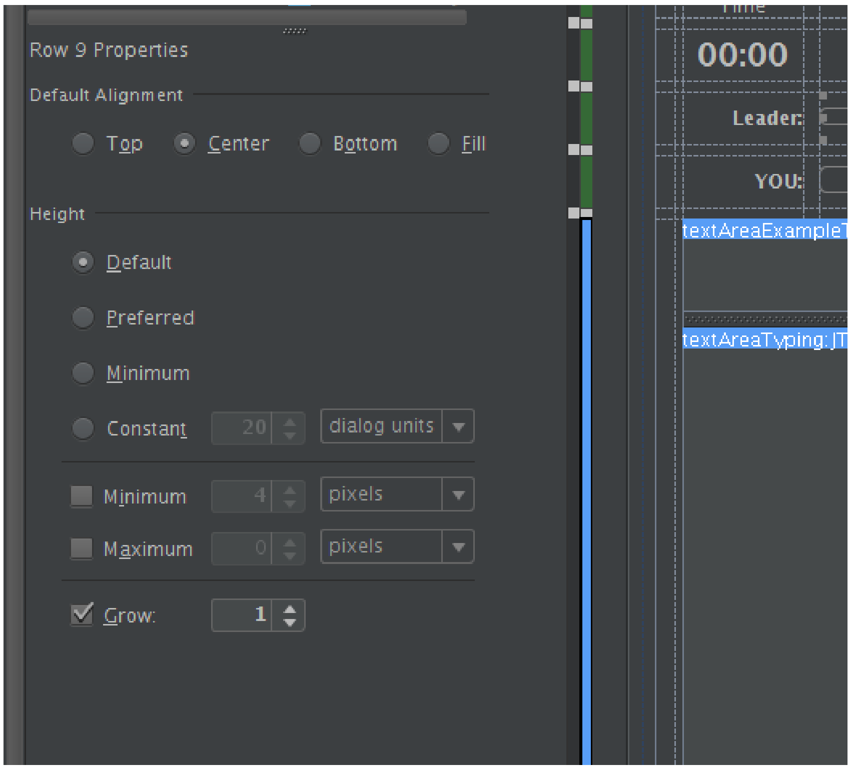
Minimum 4 pixels

Maximum 0 pixels

Grow: 1

Pic 16. A constant size row in FormLayout.

In contrast, the row which can automatically pull away when the form is resized, has the tick against its *Grow* property in the editor. In a resizeable form you must have one such row at each hierarchy level:


Row 9 Properties

Default Alignment

Top [✓ Center] Bottom Fill

Height

[✓ Default]

Preferred

Minimum

Constant 20 dialog units

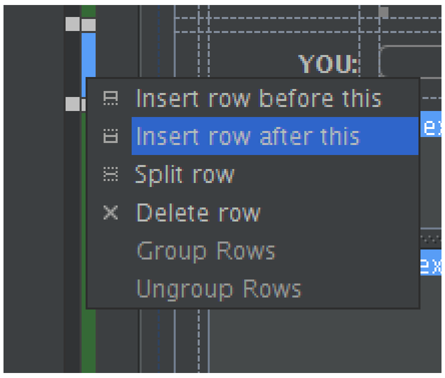
Minimum 4 pixels

Maximum 0 pixels

[✓ Grow:] 1

Pic 17. A resizeable row. Column properties are set the same way.

And this is how you modify grid layout with a pop-up menu appearing by a right-click on a row:



Pic 18. A grid layout control pop-up.

Finally, you can always check and adjust things in the XML file describing the source of your form. If you right-click anywhere in the form editor and choose *Form Source* menu item, you go there.

4. Adding behavior to the view class.

Let's go down to `TypingTestView` class source. It now has few code lines and many declarations of objects bounded to components we've named in the form editor. Most of them we'll be involved in the application's life cycle.

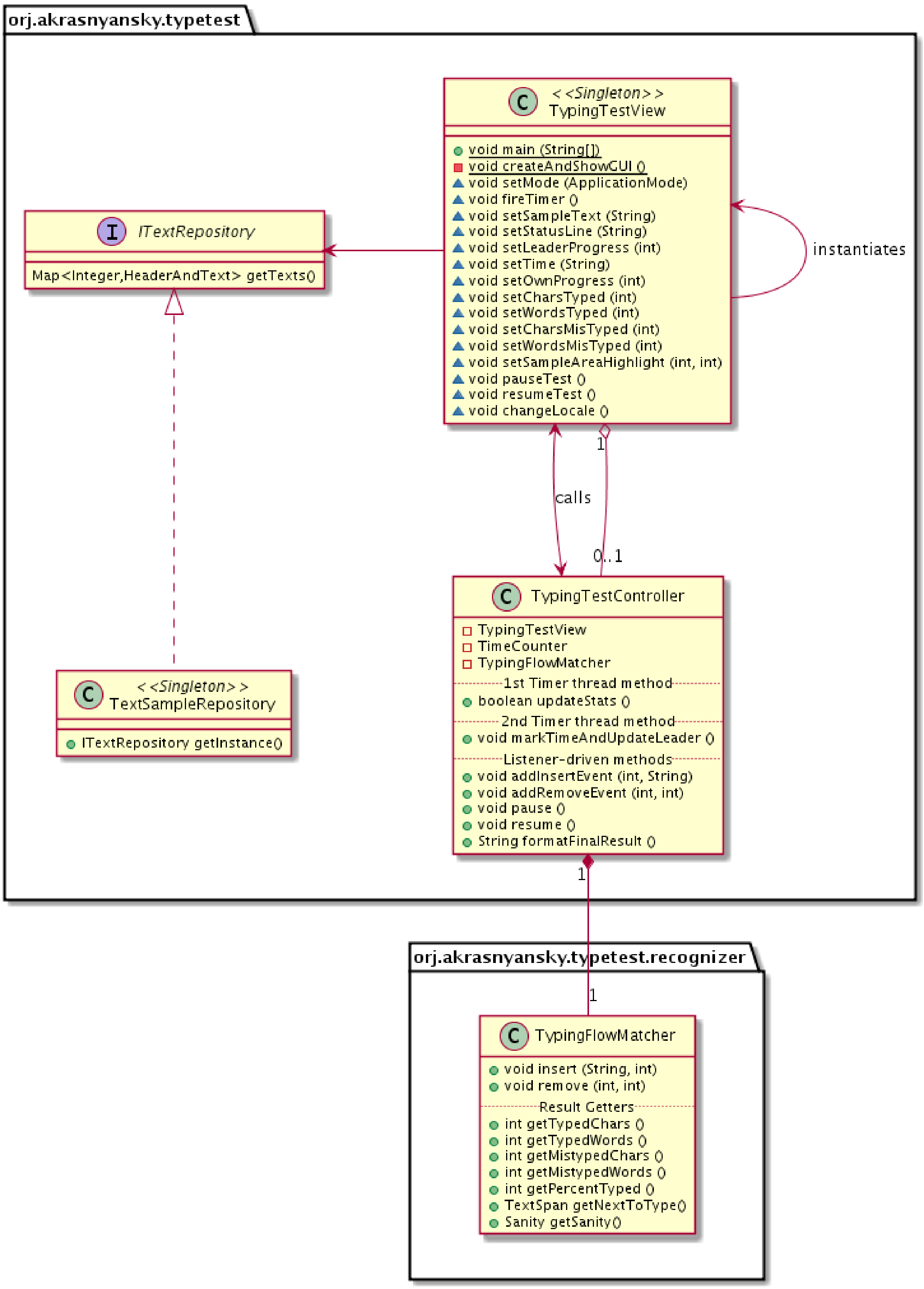
Firstly, we have to made a modification to the static `main()` method. Code generator of IntelliJ Idea put the form creation code straight into this method, but it is not always okay from the perspective of how Swing API handles visual components creation and our need to initialize them with some pre-loaded content. This is not safe to do in the application's main thread as long as components being modified have been realized, meaning, painted on screen. Such behavior is known to cause deadlocks.

It's a common recommendation to run all the code related to application startup on the so called *Event Dispatch Thread* (EDT from now on). This is a special thread which handles interface elements update events and commands, and there's a pattern to run a code on it. Let's move the frame creation code in a separate method called `createAndShowGUI()` and change the `main()` method to:

```
public static void main(String[] args) {
    // Init the GUI
    SwingUtilities.invokeLater(new Runnable() {
        @Override
        public void run() {
            createAndShowGUI();
        }
    });
}
```

Such way, we tell Swing to `createAndShowGUI()` at next opportunity on the EDT. We will massively use the same invocation method throughout the `TypingTestView` code, you'll see it.

Now let's regard the overall class structure of the application:



Pic 19. The class diagram for the primary classes of TypingTest application.

Note: The full class diagram is too large to fit in the document without much distortion, and is attached to the set of documentation as a separate graphical file **TypingTest-class.png**

I leave the view class **`TypingTestView`** with the responsibilities to interact with the user.

`TextSampleRepository` class implementing interface `ITextRepository` (for ease of unit testing) is a service class for loading sample texts from embedded resources into memory. It is used to fill the selection list for sample text.

`TypingTestController` is created specifically for each typing exerise in *Testing* state (see pic 11). It accumulates typing inputs from the user and serves periodic requests from the view to update visual components.

`TypingFlowMatcher` is a gateway class in a separate package that operates on two model representations of texts: one for the sample text and another for the text the user is entering. By requests from the controller, it computes the difference measures between two texts and returns numbers ready for displaying.

We will review specifics of the three latter classes in the next chapter, for now it's enough to know that we want to use the following interface for extracting the headers and texts themselves from a sort of source:

```
public interface ITextRepository {
	public static class HeaderAndText {
		public final String header;
		public final String text;
		public HeaderAndText(String p_header, String p_text) {
			header = p_header;
			text = p_text;
		}
	}
	Map<Integer,HeaderAndText> getTexts();
}
```

And we call the following methods of a working **controller** to queue in new user edits:

```
public void addRemoveEvent(int offset, int length);
public void addInsertEvent(int offset, String text);
```

We always need the **text repository**, so let's have it a final field, created once the application runs:

```
private final ITextRepository textRepository = TextSampleRepository.getInstance();
```

A controller will be created each time the user begins her or his exercise.

Inside the mentioned `createAndShowGui()` method, I delegate the application's initialization code to a special method called **`initApp()`**:

```
private static void createAndShowGUI() {
	TypingTestView x_view = new TypingTestView();
	x_view.initApp();
	JFrame x_frame = new Jframe("Typing Test");
	x_frame.setContentPane(x_view.panelTopLevel);
	x_frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
	x_frame.pack();
	// Now centering the frame at the screen, resizing if necessary
	// … (omitted here) …
	x_frame.setVisible(true);
}
```

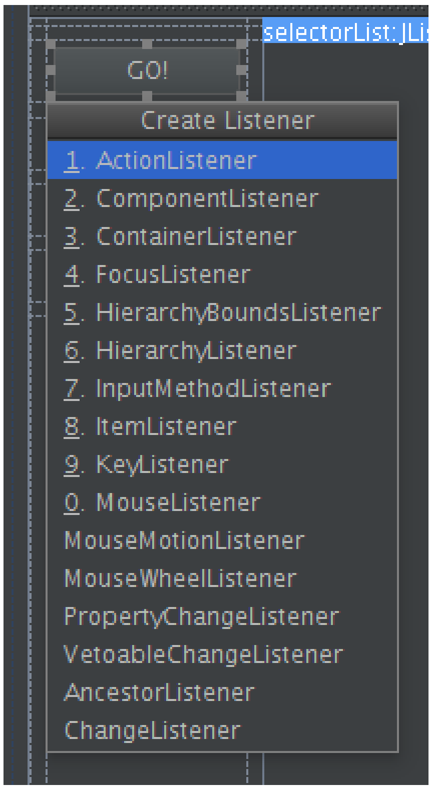
We also need to abstract life cycle changing events in a method **`setMode()`**, which coordinates switching between them:

```
void setMode(ApplicationMode p_mode) {
	mode = p_mode;
	CardLayout layoutOuter = (CardLayout) panelCardContent.getLayout();
	switch (p_mode) {
		case GREETING:
			initPicking();
			layoutOuter.show(panelCardContent, "cardInitial");
			break;
		case TESTING:
			initTest();
			adjustSplittersAuto();
			layoutOuter.show(panelCardContent, "cardWorking");
			textAreaTyping.requestFocus();
			break;
		case TURNAROUND:
			initPicking();
			initResults();
			layoutOuter.show(panelCardContent, "cardInitial");
	}
}
```

We will review various init methods a bit later.
Note that **`ApplicationMode`** is externalized as a simple java enum:

```
public enum ApplicationMode {
	GREETING,
	TESTING,
	TURNAROUND
}
```

Now, in `initApp()` method, we define component properties, listeners and start up one of the timer-based tasks. Let's begin with a simple thing – **creating a listener** for "Go!" button. First, select the button in the form editor, right-click, select *"Create a Listener"* option and pick **ActionListener** as the kind you need:



Pic 20. Making a listener.

This will create a code stub with `actionPerformed()` method:

```
buttonGo.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {

    }
});
```

Which we fill in with our `setMode()` method call:

```
buttonGo.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        setMode(ApplicationMode.TESTING);
    }
});
```

The same way we add a `ListSelectionListener` listener code for **`selectorList`**, to make sure that "Go!" button cannot be pressed when nothing is selected in the list (see the full code to check).

This is also nice to make numbers for how many typos measured in characters and words the user is having, disappear when such numbers are zero. Normally, the typer should not make mistakes, and it's ergonomic to have these red labels hidden when she or he hadn't yet made any mistake.

There are labels **labelCharTypos** and **labelWordTypos**, whose *text* property will later be assigned from a controller. To make themselves responsible for their own visibility, let's add a PropertyChangeListener to each:

```
labelCharTypos.addPropertyChangeListener(new PropertyChangeListener() {
    @Override
    public void propertyChange(PropertyChangeEvent evt) {
        if (PROPERTY_TEXT.equals(evt.getPropertyName())) {
            if ("0".equals(evt.getNewValue())) {
                SwingUtilities.invokeLater(new Runnable() {
                    @Override
                    public void run() {
                        labelCharTypos.setVisible(false);
                    }
                });
            } else {
                SwingUtilities.invokeLater(new Runnable() {
                    @Override
                    public void run() {
                        labelCharTypos.setVisible(true);
                    }
                });
            }
        }
    }
});
```

with the constant declared as

```
private static final String PROPERTY_TEXT = "text";
```

As you can see, it checks that if the new text value is string “0”, then hide the label and show otherwise.

A little bit complex behavior is tied to the test **Cancel** button. I want the user to be able to access the Abort dialog either by clicking the button or by pressing Escape key on her or his keyboard. To make it possible, I needed to define a simple Action class and work with key bindings for the component container. This is how.

Let's first declare an ActionAbort as a class extending AbstractAction. In its working method it pauses the test controller, creates a modal JOptionPane with a system-native look, a question, and Ok and Cancel options to choose between. If clicked Ok, it will call a switch to *Greeting* mode, if clicked Cancel, it will resume the controller.

```
private class ActionAbort extends AbstractAction {
    @Override
    public void actionPerformed(ActionEvent e) {
        pauseTest();
        Object[] options = {"Ok", "Cancel"};
        Component source = (Component) e.getSource();
        Component rootContainer = SwingUtilities.getRoot(source);
        int selectOpt = JOptionPane.showOptionDialog(rootContainer, "Abort the
typing test and go back to text selection?",
                "Test Abort",
                JOptionPane.OK_CANCEL_OPTION,
JOptionPane.QUESTION_MESSAGE, null, options, options[0]);
        if (selectOpt == 0) {
            setMode(ApplicationMode.GREETING);
        } else {
            resumeTest();
        }
    }
}
```

This action makes sense when the typing test is running and the input focus is held by any of the components which are in the children tree for `panelWorkingContent`, so, in `initApp()` we add:

```
// Actions
panelWorkingContent.getActionMap().put(ABORT_KEY, new ActionAbort());
```

Actions in the action maps can actually be keyed by anything, so here's:

```
private static final Object ABORT_KEY = new Object();
```

Now in the same method we can associate the button with the action:

```
buttonCancelTest.addActionListener(panelWorkingContent.getActionMap().get(ABORT_
KEY));
```

And add the appropriate key bindings for **Esc** and **Shift**+**Esc** combinations:

```
// Key bindings
InputMap imap =
panelWorkingContent.getInputMap(JPanel.WHEN_ANCESTOR_OF_FOCUSED_COMPONE
NT);
imap.put(KeyStroke.getKeyStroke(KeyEvent.VK_ESCAPE, 0), ABORT_KEY);
imap.put(KeyStroke.getKeyStroke(KeyEvent.VK_ESCAPE, InputEvent.SHIFT_MASK),
ABORT_KEY);
```

This is almost it. One thing we want to do is to prevent the user from **cheating**. Look, it's just so easy to select the text in the upper text pane and Ctrl+V it into the lower pane, et voilà! A fantastic typing speed! You are the champion! :-)

To make that impossible, let's first disable “focusability” of `textAreaExampleText`: in the form editor, find that component, and uncheck it's *focusable* property. This way, it can't be selected in the form, and we are fine with this.

However we cannot do the same with `textAreaTyping`, all we need there is to prevent the user from pasting a text in, allowing only letter-by-letter typing. This is done in `initApp()` method so:

```
ActionMap typingActionMap = textAreaTyping.getActionMap();
typingActionMap.get("paste").setEnabled(false);
typingActionMap.get("paste-from-clipboard").setEnabled(false);
```

Two more things related to the visual stuff, that are too trivial for this review, but best to be mentioned:

Firstly, adjusting **position of splitters** in the initial and working modes to best proportions. Done as a call to special method `adjustSplittersAuto()` from inside `createAndShowGUI()` after the form is “packed”, but before is made visible.

Secondly, I wanted the **look and feel** of the form to differ from the default one, if possible, and thus wrote method `trySetNimbus()`, which is called from `main()` just before calling `createAndShowGUI()`. It will try to set Nimbus L&F via Swing's `UIManager`, if that is available at runtime.

Finally, the `initApp()` contains a call to `setMode()` method, telling it to set the *Greeting* mode at the start.

There will be one more important addition here, but let's back track to it afterwards, in the section where we review Controller handing.

To finish this section, let's describe `initPicking()`, `initTest()` and `initResults()` private methods, which do the work for `setMode()`.

`initPicking()` is there to fill `textAreaInfo` with a string describing for the user the purpose of the program, to reinitialize `selectorList` with captions of sample texts, appending a length of each text in characters to the end of each caption:

```
DefaultListModel<String> listModel = new DefaultListModel<String>();
Map<Integer, ITextRepository.HeaderAndText> textMap = textRepository.getTexts();
for (int i = 0; i < textMap.size(); i++) {
      int charLength = textMap.get(i).text.length();
      listModel.add(i, textMap.get(i).header + " (" + charLength + " characters)");
}
selectorList.setModel(listModel);
selectorList.requestFocus();
```

initResults() just relies on Controller to fill the same textAreaInfo with a bunch of text that is appropriate for test completion.

But **`initTest()`** is where we actually have some code full of meat. To remind you, this method is called when the user clicks “Go!” and we switch to *Testing* mode.

At first piece, the “game level” is determined by finding which radio button is selected:

```
    // Determining the level
    final LeaderLevel lvl;
    if (radioLvlEasy.isSelected()) {
        lvl = LeaderLevel.NORM;
    } else {
        if (radioLvlMedium.isSelected()) {
            lvl = LeaderLevel.GOOD;
        } else {
            lvl = LeaderLevel.PROFI;
        }
    }
```

Next, the `textAreaTyping` is cleared (obvious, won't quote); and a new instance of **`TypingTestController`** is created, using the text from the repository and the just found level:

```

// Create the test controller and bind to the current view
String sampleText = textRepository.getTexts().get(textIndex).text.trim();
testController = new TypingTestController(sampleText, this, lvl);
```

After we fill the upper part of the pane with the same sample text, two more things are done, related to the Controller.

One is binding of the **stream of text edit events** that happen when the user types the text, to the input methods of Controller. To do that, we assign a custom `DocumentFilter` object to the `Document` object underlying `textAreaTyping` (please read Text Component Features and The Text Component API articles for better detail). It overrides `insertString()`, `remove()` and `replace()` methods of the `DocumentFilter` to react on text mutation at a high level:

```
// Associate the typing text editor with the controller
((AbstractDocument) textAreaTyping.getDocument()).setDocumentFilter(new
DocumentFilter() {
    @Override
    public void remove(FilterBypass fb, int offset, int length) throws
            BadLocationException {
        if (testController != null) {
            if (length > 0) {
                testController.addRemoveEvent(offset, length);
            }
        }
        super.remove(fb, offset, length);
    }
    @Override
    public void insertString(FilterBypass fb, int offset, String string,
                             AttributeSet attr) throws BadLocationException {
        if (testController != null) {
            if (string.length() > 0) {
                testController.addInsertEvent(offset, string);
            }
        }
        super.insertString(fb, offset, string, attr);
    }
    @Override
    public void replace(FilterBypass fb, int offset, int length, String text,
                        AttributeSet attrs) throws BadLocationException {
        if (testController != null) {
            if (length > 0) {
                testController.addRemoveEvent(offset, length);
            }
            if (text.length() > 0) {
                testController.addInsertEvent(offset, text);
            }
        }
        super.replace(fb, offset, length, text, attrs);
    }
});
```

We assume that some events may be bogus in nature, and because of that the affected text length is checked, and the program does nothing, if that is zero. A replace event is sent down to Controller as a couple of events: first remove, then insert to the same place.

The above code **works on the EDT**, and therefore, by Swing UI development convention, must return quickly to ensure the UI responsiveness from the user's standpoint. We should keep this in mind when developing the called methods for the Controller.

And finally (really finally here!) we use a `SwingWorker` idiom to fork a separate execution thread which wakes up once in half a second to do a heavier work in the Controller's `updateStats()`. This is the appropriate way to have a second thread in a Swing application.

```
// The text analyzer start
new SwingWorker() {
    // On the worker thread
    @Override
    protected Object doInBackground() throws Exception {
        // Spin until the exit command received
        boolean isComplete = false;
        while (!isComplete) {
            Thread.sleep(500); // 0.5 seconds
            isComplete = testController != null && testController.updateStats();
        }
        return null;
    }
    // On the EDT when the primary process finishes
    @Override
    protected void done() {
        setMode(ApplicationMode.TURNAROUND);
    }
}.execute();
```

Note that the overridden method `done()` is called on the EDT, as the comment implies.

`TypingTestView` also includes a number of value setter methods, out of which only **`setSampleAreaHighlight()`** is not so trivial. It receives position and length of text span to highlight, refreshes highlighting and ensure that the highlighted area is fully visible to the user, calling `setViewPosition()` of `JViewport` object of the upper text pane (see the full code).

5. Text Repository and Controller. Counting the time and progress.

There is no much to say about **`TextSampleRepository`**. It is hard-tailored to extract content from resource files put into the *resources* folder of the project or straight from *.jar file which holds the code. There may be *Integer.MAX_VALUE* short one files with names patternized like *TextFragment0*, *TextFragment1*, etc. The loading breaks when a file with a next number suffix is not found, so no enumeration holes are allowed.

Each file must contain the first line beginning with “header:” prefix and the rest of the line is treated like the text's header to display in the selection list.

All the lines following the fist one are treated like text content. Strings are read from the resources using UTF-8 character set, and each `BufferedReader` responsible for that is gracefully closed in the *finally* clause of the try-catch-finally statement (see `initContents()` method).

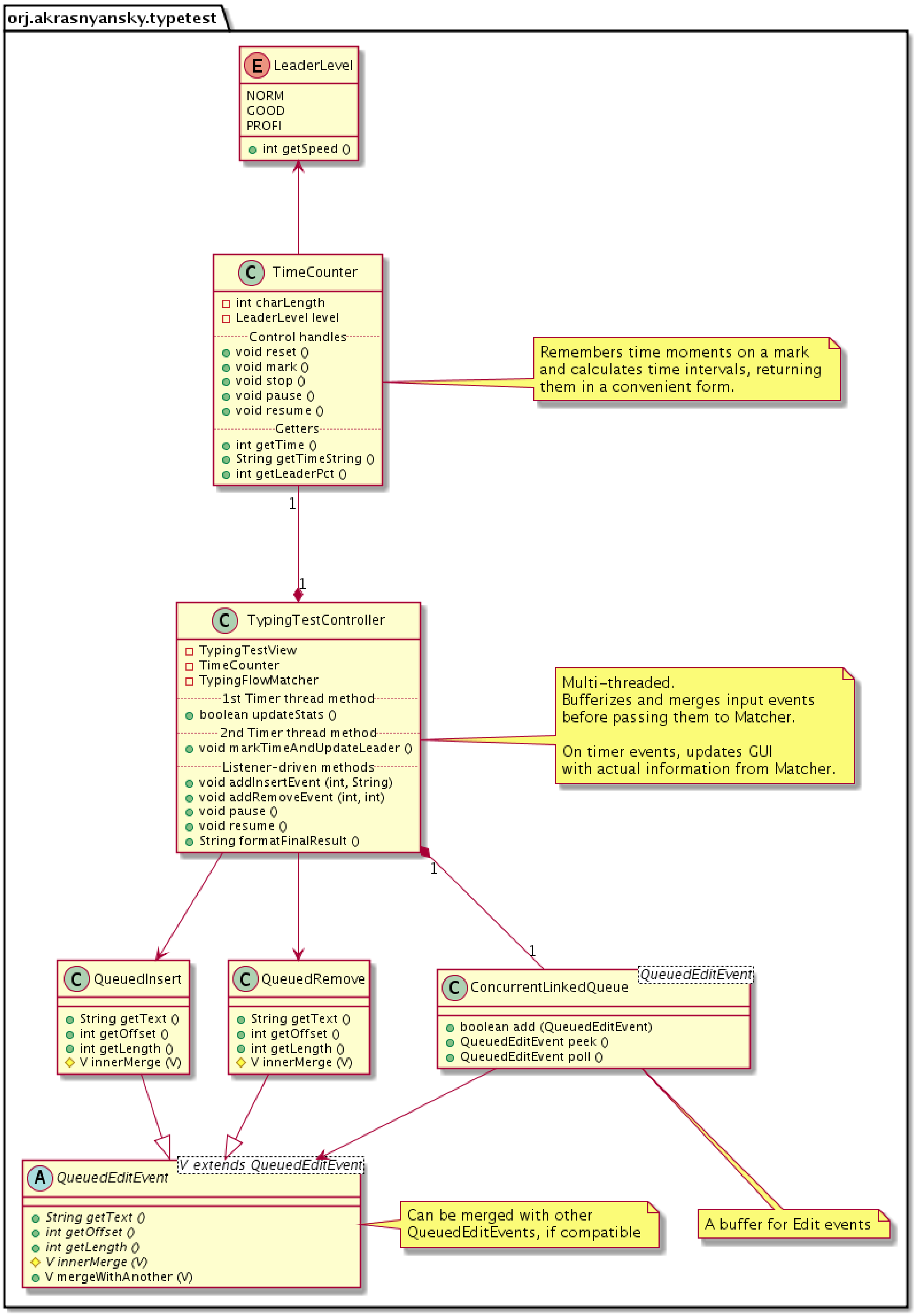
The results are returned as a map with the key set of integers, corresponding to the numerical suffixes in the *TextFragmentN* files.

`TypingTestController` holds a special sub-hierarchy of classes it needs to do the work, see Pic 21 at the next page.

When `addInsertEvent()` or `addRemoveEvent()` methods are called, they create corresponding `QueuedInsert` or `QueuedRemove` objects, one per the corresponding call, and put them into the tail of `ConcurrentLinkedQueue`.

When a timer-driven worker calls `updateStats()` method on the Controller, two things happen: firstly, `flushQueue()` method called, which polls the head of the queue and tries to merge compatible edit events:

```
// On the worker thread
private void flushQueue() {
    QueuedEditEvent topEvent = editQueue.poll();
    if (topEvent != null) {
        // Try to merge with the next event
        QueuedEditEvent comboEvent;
        do {
            comboEvent = topEvent;
            topEvent = topEvent.mergeWithAnother(editQueue.peek());
            if (topEvent != null) {
                editQueue.poll(); // to delete the merged event from the queue
            }
        } while (topEvent != null);
        // Now action the event
        if (comboEvent instanceof QueuedInsert) {
            matcher.insert(comboEvent.getText(), comboEvent.getOffset());
        } else { // this is QueuedRemove
            matcher.remove(comboEvent.getOffset(), comboEvent.getLength());
        }
    }
}
```

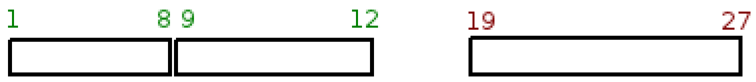


Pic 21. Hierarchy of classes subordinate to `TypingTestController`.

Each edit event has numbers of offset and length of the affected text span. In case of inserts, the length is calculated from the inserted string itself.

`MergeWithAnother()` will call to concrete implementation of `innerMerge()` method in the descendants to apply the following rules of merging:

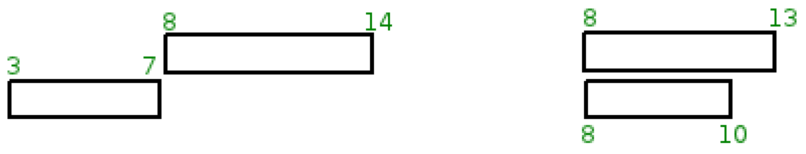
- Inserts merge to inserts, removals merge to removals.
- Inserts merge when they touch the boundaries of each other. For example, inserts 1 (spanning from position 1 to 8) and 2 (from 9 to 12) can be merged, and the next one, spanning from 19 to 27 is behind a "hole" between 13 and 18, and therefore it can't be merged.



Pic 22: Merging and non-merging of inserts.

  Merged inserts have their textual content concatenated.

- Removals merge when their boundaries touch in the order of text mutation, the picture shows two possible cases when subsequent removals can be merged. In both cases, the length of text necessary to remove is combined as a sum of both removal's lengthes.



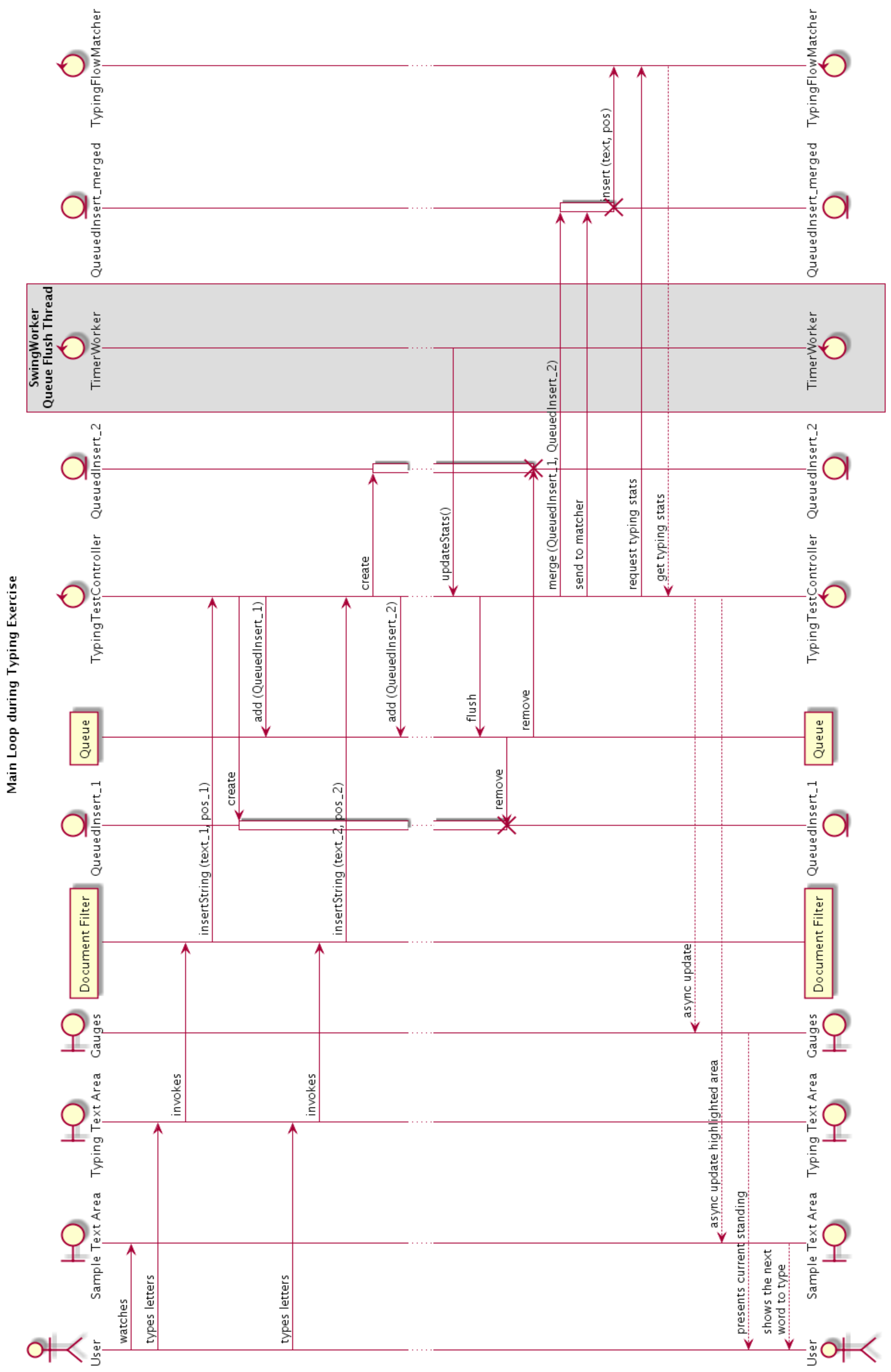
Pic 23: Merging of removals, two cases.

The ratio for buffering and combining the edit events is that because the user may type very quickly, putting many small inserts to the queue, which we poll only once in a ½ of a second (because the parsing/matching operation is heavy). Also it's not so likely that users' inputs will affect the text in scattered places, on the contrary, inserts and removals tend to chain naturally during typing or multiple pressing on the backspace key.

So, as you can see, the controller's `flushQueue()` method will call corresponding `insert()` and `remove()` method on the Matcher and upon return, `updateStats()` will synchronously call the Matcher's getters to obtain some numerical values from the text and update this information at the View.

The important line in `updateStats()` method is the last one, it sets the exit criterion for the end of the exercise: the return value becomes true when either the user or the virtual leader gets to the end of text, and that's measured as the "percentage" value reaching 100.

```
public boolean updateStats() {
	flushQueue();
	int pct;
	pct = matcher.getPercentTyped();
	// ... omitted ...
	return pct  >= 100 || timeCounter.getLeaderPct() >= 100;
}
```



Pic 24. Sequence diagram of the edits bufferization in `TypingTestController`.

Additionally, `TypingTestController` class does two more things.

Firstly, it creates and holds an instance of TimeCounter. **markTimeAndUpdateLeader()** method is driven by the following code pieces in TypingTextView:

In initApp():

```
// The timer task start
new Timer(100, new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        instance.fireTimer();
    }
}).start();
```

the fireTimer() method:

```
void fireTimer() {
    if (mode == ApplicationMode.TESTING) {
        if (testController != null) {
            testController.markTimeAndUpdateLeader();
        }
    }
}
```

Each 100 milliseconds `markTimeAndUpdateLeader()` calls `TimeCounter` methods to get reading of a time passed since the controller's creation and calculates the number of characters the leader should have typed by this moment, based on the known length of text and the leader's typing speed, obtained from `LeaderLevel` enum (it has `getSpeed()` method for that). The leader's progress bar at the main view is adjusted to this number, expressed in percents of the whole text.

Secondly, the controller has `formatFinalResults()` method, which is simply a concatenation routine for textual constants and numerical readings from `TimeCounter` and `TypingFlowMatcher`. It is called by the view to get the lines showing the user her or his standing for the Turnaround mode.

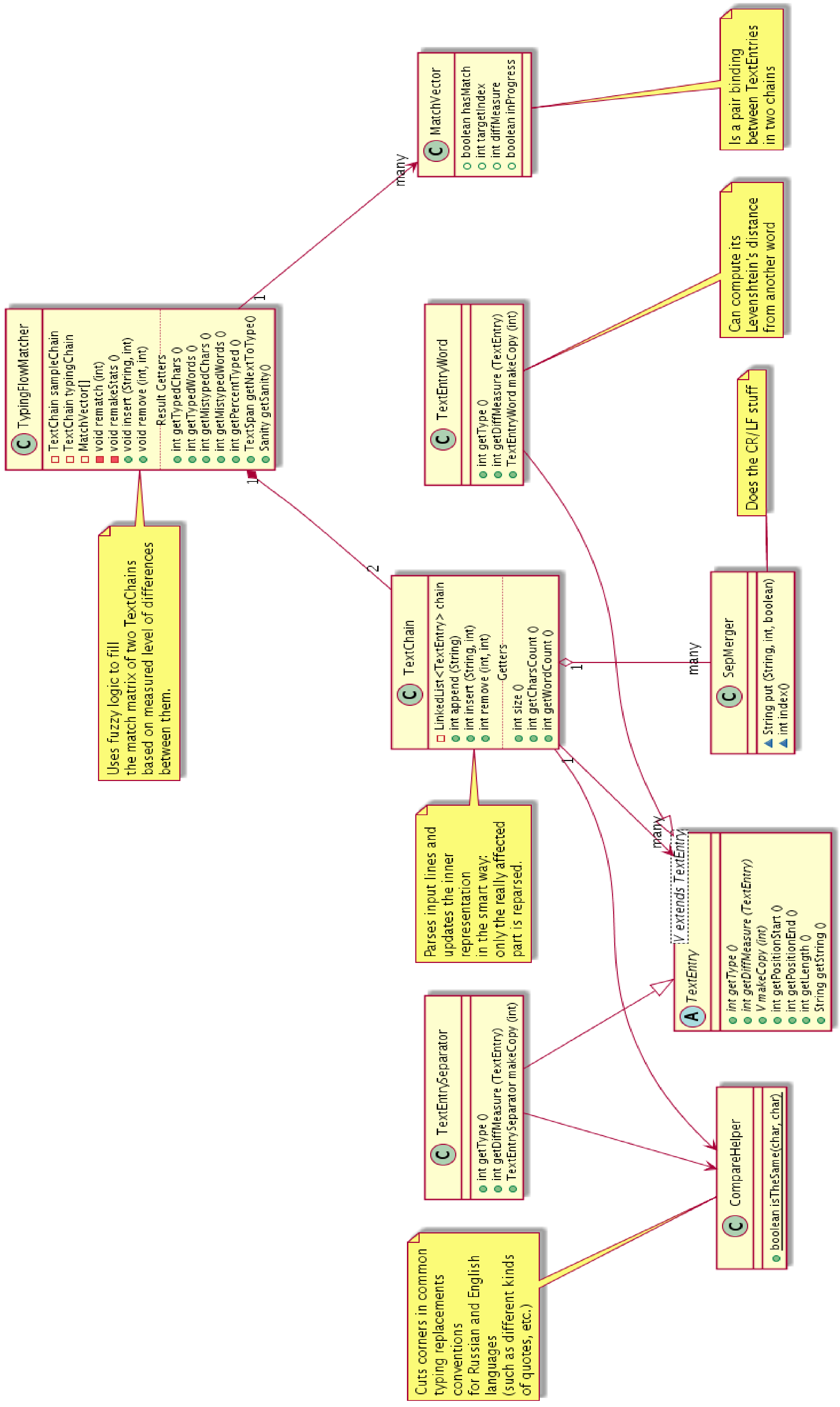
6. Recognizer package and TypingFlowMatcher.

The function of *recongizer* package and classes included into it is maintaining an abstraction of text fragments understood as words and separators dividing them, and drawing a measure of match between the sample text and the text the user have typed.

When full match between two texts is reached, it means that the user have typed everything completely correct.

This analysis has to be done in parallel with user typing and cause no observable lag in live displaying of the output numbers, so it was kept in mind during the solution design. Some optimization is already provided by the Controller which bufferizes text alterations as described in the previous chapter, but that's not yet all.

Let's regard the structure of classes that do the work here:



Pic 25. Herarchy of classes subordinate to `TypingFlowMatcher`.

The gateway class for everything is **`TypingFlowMatcher`**. On creation it takes two new `TextChain` objects, the sample chain initialized with the sample text, and an empty chain to keep typing objects:

```
matcher = new TypingFlowMatcher(new TextChain(sampleText), new TextChain());
```

The data structures it maintains are two `TextChains`, a `LinkedList` for indexed `TextEntries`, and two arrays, called **`vectors`** (keeping the cross-match information between the chains) and `wordMap` which helps to remembers which are of the entries in the source chain are words, and which are not.

```
private final TextChain sampleChain; // Holds the sample text
private final TextChain typingChain; // Holds the user input
private final LinkedList<IndexedEntry> errorChain; // Holds unmatched entries from
the user input, which we cannot match to anything
/**
  * Is a pair binding between TextEntries in two chains. Mutable.
  */
private class MatchVector {
      public boolean hasMatch = false;   // true if there is a match
      public int targetIndex = -1;       // Element index in the list of the second chain's
element
      public int diffMeasure = -1;       // 0 if there's a full match, the greater, the bigger
is the difference
      public boolean inProgress = false; // true if we conclude it's a word which the user
is normally typing, but
                                                      // just haven't got to the end of that
word yet (last in the current text)
}
private final MatchVector[] vectors; // Playground for matching, N of elements = N of
items in the sampleChain
private final boolean[] wordMap;     // true if a word, false if a separator, N of elements
= N of items in the sampleChain
```

NOTE on term usage:

- *Position* is a counter of characters in a text string staring from its beginning, zero-based.
- *Index* is a counter of items in a collection holding text entries (`TextChain` or array).

These two terms should never be mixed in context of this application for clarity.

Before we review what `TypingFlowMatcher` does, let's drill down into details of the manipulated data entities, because some logic is scattered there too.

Abstract class **`TextEntry`** is realized as either **`TextEntryWord`** or **`TextEntrySeparator`**. Each realization is an immutable holder for an information about the underlying character sequence and a number denoting its position in the source text.

`TextEntrySeparator` can only hold one character, so if there are few separators between words, each gets its own entry.

The biggest difference are in their realization of `getDiffMeasure()` method. This method should return 0 in case when two text entries fully match and some greater number otherwise, the greater, the bigger the difference.

`TextEntrySeparator` works as:

```
@Override
public int getDiffMeasure(TextEntry o_entry) {
	if (o_entry == null) {
		return 1;
	}
	if (this == o_entry) {
		return 0;
	}
	if (!(o_entry instanceof TextEntrySeparator)) {
		return 1;
	}
	if (realSep != ((TextEntrySeparator) o_entry).realSep) {
		if (CompareHelper.isTheSame(realSep, ((TextEntrySeparator)
o_entry).realSep)) {
			return 0;
		} else {
			return 1;
		}
	}
	// Otherwise, it's the same
	return 0;
}
```

It relies on `CompareHelper`'s static method `isTheSame()`, which helps to recognize when the supplied character is replaceable by some different one. For example, if the source text has an opening guillemot anywhere: «, it is regarded the same as anything of: (",„,“,‘,‹).
At the same time, it should not be confused with any kind of closing quote, so `CompareHelper` has few different sets for all kinds of commonly used interchangeable separators.

`TextEntryWord` relies on library function `getLevenshteinDistance()` from Apache *StringUtils* package – that is the number of changes required to turn one string into another:

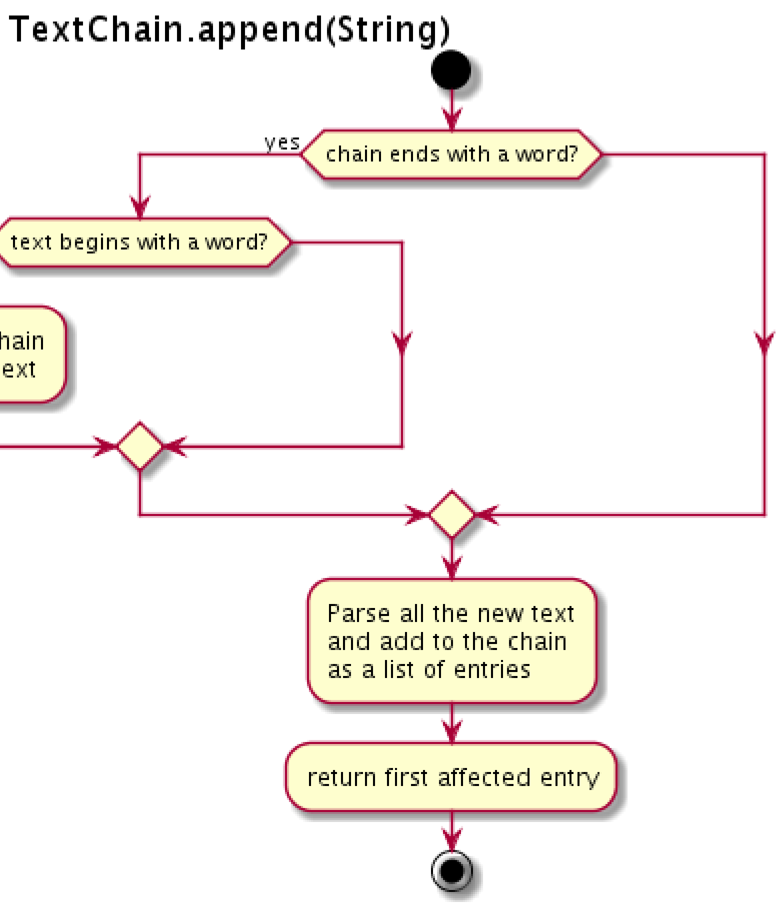
```
	@Override
	public int getDiffMeasure(TextEntry o_entry) {
		if (o_entry == null) {
			return content.length();
		}
```

```
        if (this == o_entry) {
            return 0;
        }
        if (!(o_entry instanceof TextEntryWord)) {
            return content.length();
        }
        return StringUtils.getLevenshteinDistance(content, o_entry.content);
    }
```

Going a level up, TextEntries are kept within **`TextChain`** objects, whose main purpose is to parse the input lines and be smart at parsing the same line again: only the really affected parts should be processed.

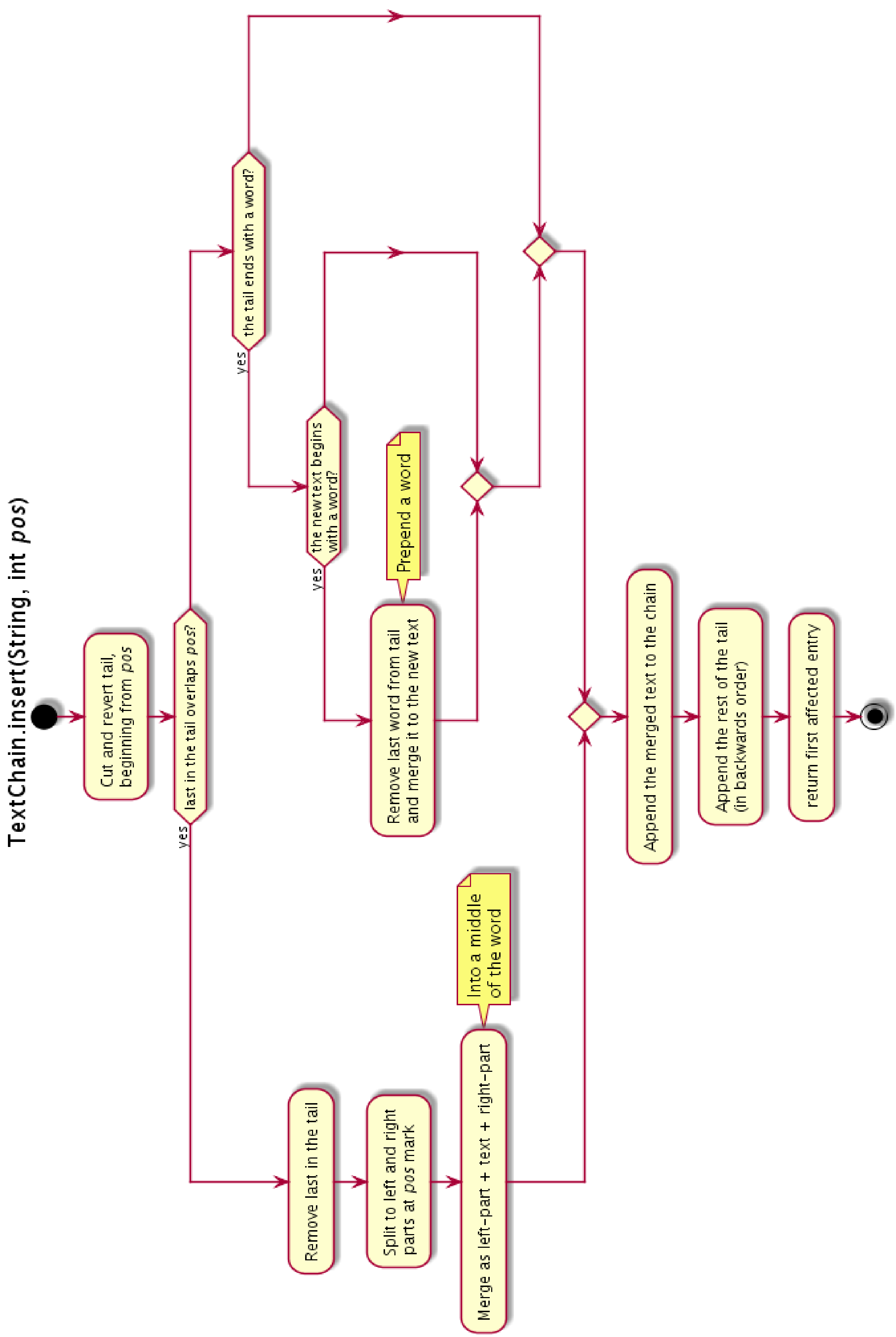
This is the most complex part of the program, and it could not realistically be debugged without the technique of Test-Driven Development, when I first made enough Junit tests and then found all the corner cases.

Alterations to a `TextChain` happen by calling one of the three methods: `append()`, `insert()`, or `remove()`.

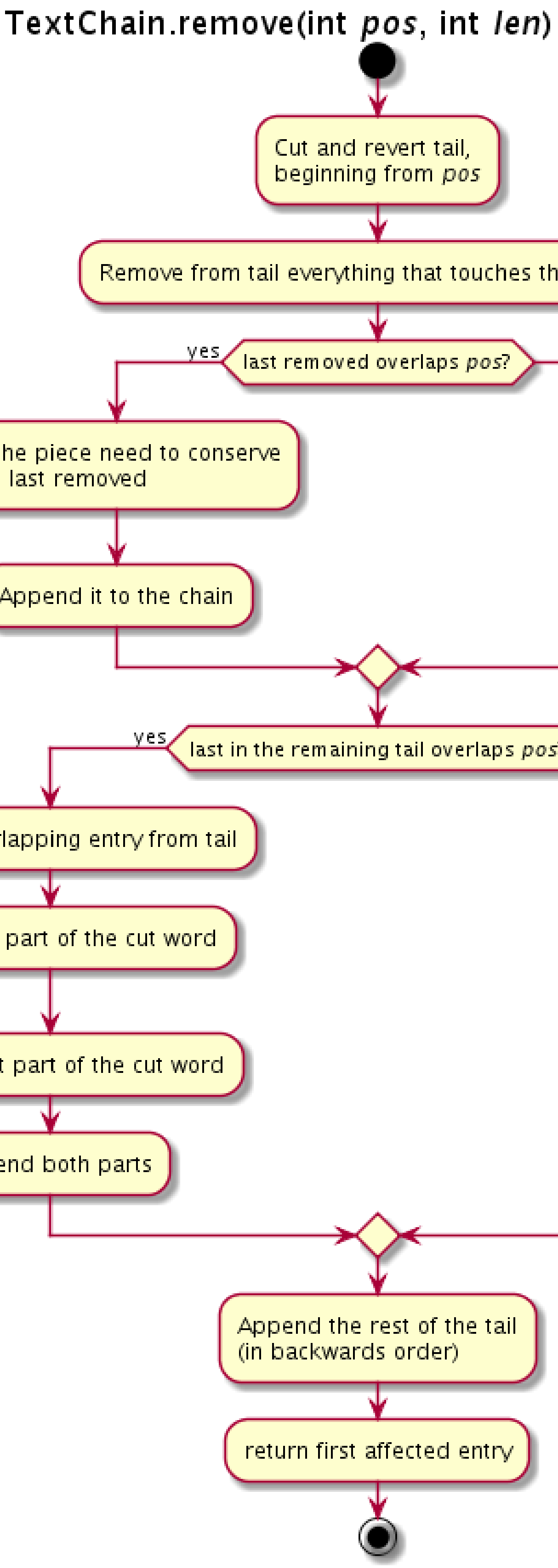

TextChain.append(String)
yes
chain ends with a word?
yes
text begins with a word?
Remove last word from chain and merge it to the new text
Parse all the new text and add to the chain as a list of entries
return first affected entry

Pic 26. Activity diagram of `TextChain.append()`.

This `append()` method is also used for two other routines:

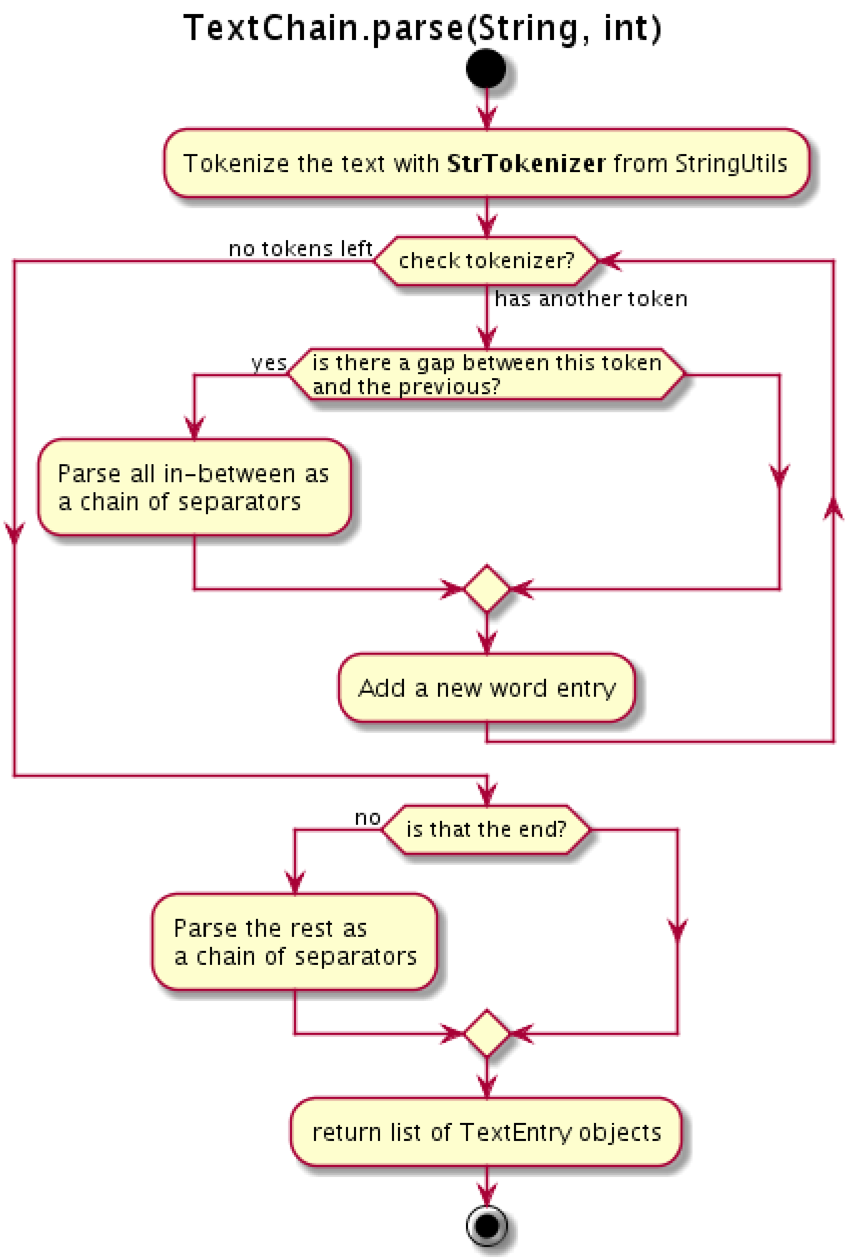

TextChain.insert(String, int pos)
Cut and revert tail, beginning from pos
last in the tail overlaps pos?
yes
Remove last in the tail
Split to left and right parts at pos mark
Merge as left-part + text + right-part
Into a middle of the word
the tail ends with a word?
yes
the newtext begins with a word?
yes
Remove last word from tail and merge it to the new text
Prepend a word
Append the merged text to the chain
Append the rest of the tail (in backwards order)
return first affected entry

Pic 27. Activity diagram of `TextChain.insert()`.


TextChain.remove(int *pos*, int *len*)
Cut and revert tail, beginning from *pos*
Remove from tail everything that touches the interval
yes
last removed overlaps *pos*?
Cut the piece need to conserve from last removed
Append it to the chain
yes
last in the remaining tail overlaps *pos*?
Remove overlapping entry from tail
Extract left part of the cut word
Extract right part of the cut word
Append both parts
Append the rest of the tail (in backwards order)
return first affected entry

Pic 28. Activity diagram of TextChain.remove().

As you can see, the `append()` function relies on text parsing routine, which, in turn, relies on `StrTokenizer` class from Apache *StringUtils* to break the line into words. However, we need not only words, but also space between them, so it's treated especially:



TextChain.parse(String, int)
Tokenize the text with StrTokenizer from StringUtils
check tokenizer?
no tokens left
has another token
is there a gap between this token and the previous?
yes
Parse all in-between as a chain of separators
Add a new word entry
is that the end?
no
Parse the rest as a chain of separators
return list of TextEntry objects

Pic 29. Activity diagram of `TextChain.parse()`.

When parsing chains of separators, `SepMerger` class turns all CR+LF and LF into standard CR's, and this way the program is immune to platform differences of line separators.

Now, when we see how texts are turned to TextChains, let's return to the level of **`TypeFlowMatcher`** again.

Its entry points are `insert()` and `remove()` methods, which appropriately modify the typingChain with the methods we've just made ourselves familiar with:

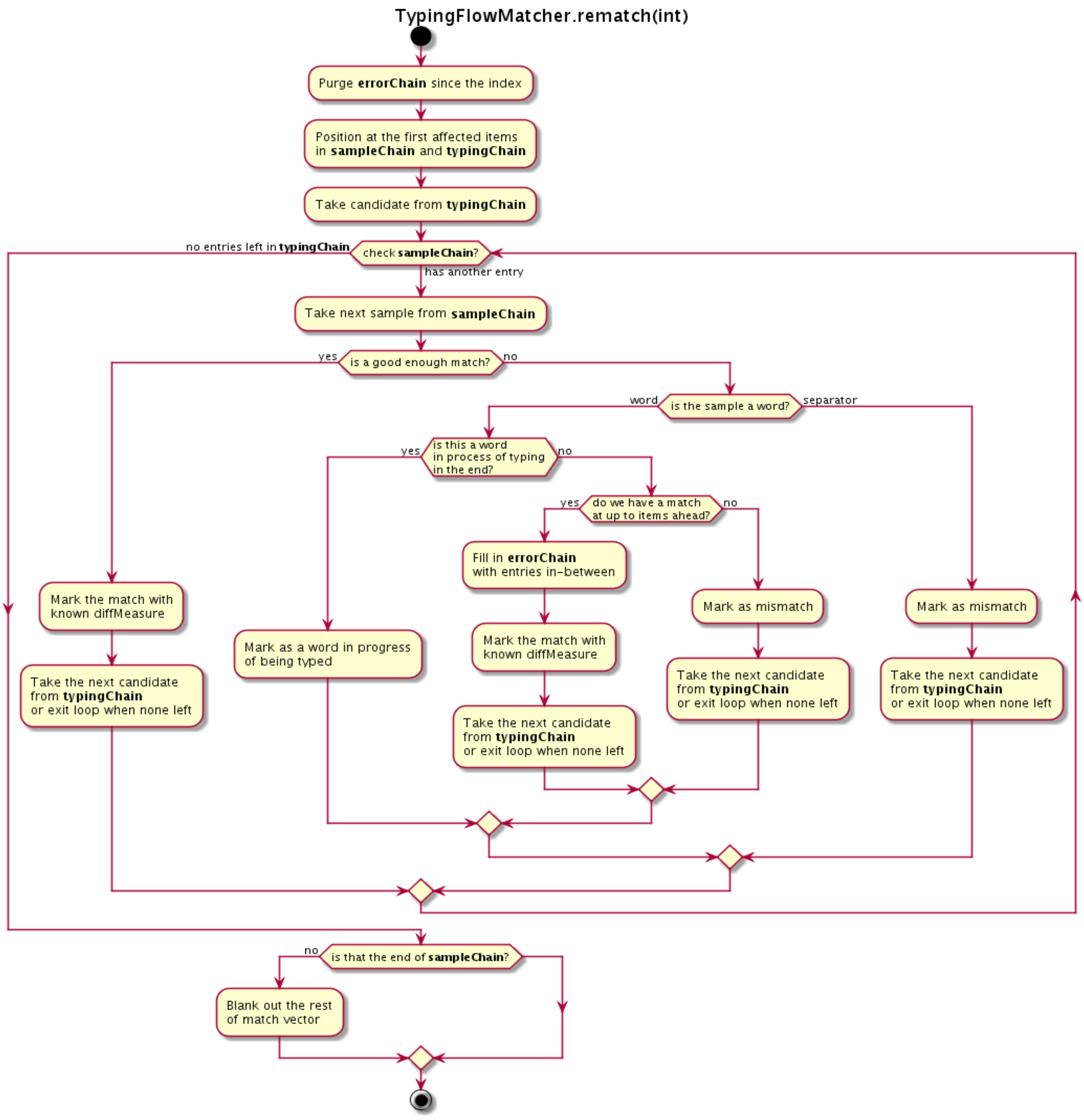
```
/**
 * Causes insertion to the typingChain, then runs matching and update to summaries.
 * @param p_text String to parse and insert to the chain
 * @param p_position Position in the text where the insert happens
 */
```

```
public void insert(String p_text, int p_position) {
	// Append or insert?
	if (typingChain.size() == 0 || typingChain.getLastEntry().getPositionEnd() ==
p_position) {
		int firstAffected = typingChain.append(p_text);
		rematch(getAffectedInSample(firstAffected));
		remakeStats();
	} else {
		int firstAffected = typingChain.insert(p_text, p_position);
		rematch(getAffectedInSample(firstAffected));
		remakeStats();
	}
}
/**
 * Causes removals from the typingChain, then runs matching and update to
summaries.
 * @param p_position Start position in the text for removal
 * @param p_length Length of the text to clip
 */
public void remove(int p_position, int p_length) {
	int firstAffected = typingChain.remove(p_position, p_length);
	rematch(getAffectedInSample(firstAffected));
	remakeStats();
}
```

Counting *firstAffected* number is an optimization measure to avoid reparsing and rematching of too much of a lengthy text. During the normal course of events, there's usually a need to search for match of the last word or two, and the white space separating them. However, to be dead sure we don't miss an important edit, rematching will always begin from the point where the first detected typo occurs:

```
/**
 * Find the rightmost index, from which to begin rematching
 * @param p_typingIndex The first touched (added or removed) in the typingChain
 * @return The index in the source chain and match vector, since which rematching is
required
 */
private int getAffectedInSample(int p_typingIndex) {
	// To be sure, always begin rematching when a discrepancy begins, if any
	int lastGood = 0;
	for (int i = 0; i < vectors.length; i++) {
		if (!vectors[i].hasMatch) break;
		if (i == vectors[i].targetIndex) {
			lastGood = i;
		}
	}
	return lastGood < p_typingIndex ? lastGood : p_typingIndex;
}
```

The most magic is held in the single method `rematch()`, whose algo is drawn at the next activity diagram.



TypingFlowMatcher.rematch(int)
Purge **errorChain** since the index
Position at the first affected items in **sampleChain** and **typingChain**
Take candidate from **typingChain**
no entries left in **typingChain**
check **sampleChain**?
has another entry
Take next sample from **sampleChain**
yes
is a good enough match?
no
word
is the sample a word?
separator
yes
is this a word in process of typing in the end?
no
yes
do we have a match at up to items ahead?
no
Fill in **errorChain** with entries in-between
Mark the match with known diffMeasure
Mark as a word in progress of being typed
Mark the match with known diffMeasure
Mark as mismatch
Mark as mismatch
Take the next candidate from **typingChain** or exit loop when none left
Take the next candidate from **typingChain** or exit loop when none left
Take the next candidate from **typingChain** or exit loop when none left
Take the next candidate from **typingChain** or exit loop when none left
no
is that the end of **sampleChain**?
Blank out the rest of match vector

Pic 30. Activity diagram of `TypingFlowMatcher.rematch()`.

Two special code pieces from this class source we should look at, are methods `isGoodEnough()`:

```
/**
  * Decides whether two text entries are looking like each other enough to draw a
match between them.
  * @param src The first (source) entry to compare
  * @param tgt The second (target) entry to compare
  * @return true if there is even a slight match
  */
private boolean isGoodEnough(TextEntry src, TextEntry tgt) {
	if (src.getType() != tgt.getType()) return false; // they must be of the same kind
	if (src.getType() == TextEntry.TYPE_SEPARATOR) {
		return src.getDiffMeasure(tgt) == 0; // Separators may only match completely or not at all
	}
	// Words are compared fuzzily, depending on their mutual Levenshtein's distance and length
	int srcLength = src.getLength();
	int diffChars = src.getDiffMeasure(tgt);
	boolean result;
	switch (srcLength) { // The longer the source word is, the more typos in it we tolerate
		case 1:
			result = diffChars == 0;
			break;
		case 2:
		case 3:
		case 4:
			result = diffChars <= 1;
			break;
		case 5:
		case 6:
		case 7:
			result = diffChars <= 2;
			break;
		default:
			result = diffChars <= 3; // Maximum 3 typos are allowed in any word!
	}
	return result;
}
```

And the loops for accumulating of summaries values in `remakeStats()`:

```
// The loop to count sum of mistyped characters, words and how many mismatches are at the chain's tail
for (int i = 0; i < vectors.length; i++) {
	if (vectors[i].diffMeasure > 0) {
		cntMistypedChars += vectors[i].diffMeasure;
		if (wordMap[i]) {
			cntMistypedWords++;
		}
	}
	if (vectors[i].hasMatch) {
		if (!vectors[i].inProgress) {
			indexNextToType = i + 1;
		}
		badMatchTailCount = 0;
	} else {
		if (vectors[i].diffMeasure > 0) {
			badMatchTailCount++; // tokens with mismatches are counted
		}
	}
}
// If complete, turn nextToType to zero item (the first word in the text)
if (indexNextToType >= vectors.length) {
	indexNextToType = 0;
}
// Now count the error chain
if (errorChain.size() > 0) {
     for (IndexedEntry indxEntry : errorChain) {
          TextEntry errorEntry = indxEntry.entry;
          if (errorEntry.getType() == TextEntry.TYPE_WORD) {
               cntMistypedWords++;
          }
          cntMistypedChars += errorEntry.getLength();
     }
}
```

As you can see, this is where *vectors[ ]* array is put to use. Variable *badMatchCount* (that's how many tailing elements are not in match) is then used to determine the "Sanity level" and provide a reassurance or to tell off the user. :-)

A good idea for getting understanding of how `TextChain` and `TypingFlowMatcher` work is to review the JUnit tests for them: **`TestTextChain`** and **`TestTypingFlowMatcher`**. These test suits include many working examples of text piece manipulation using classes described in this chapter.

It is necessary to say that for successful build of classes referring to Apache StingUtils, the following dependency must be added to *pom.xml*:

```
<dependency>
     <groupId>org.apache.commons</groupId>
     <artifactId>commons-lang3</artifactId>
     <version>3.0</version>
</dependency>
```

And for the JUnit tests to run, you'll need another one, defined in the test scope:

```
<dependency>
     <groupId>junit</groupId>
     <artifactId>junit</artifactId>
     <version>4.12</version>
     <scope>test</scope>
</dependency>
```

7. Localization (adding Russian language).

To add a value to the application, we should add a support for a national language. IntelliJ Idea makes it an easy task. The step-by-step for it follows.

1. Create a directory (for example **i18n**, a short for "internationalization") under `src/main/resources`. Add there an empty property file `TypingTest.properties`.

2. Set up a special code inspection: go to menu File – Settings – Editor – Inspections

There find and check the item ***"Internationalization issues"*** and click Ok.

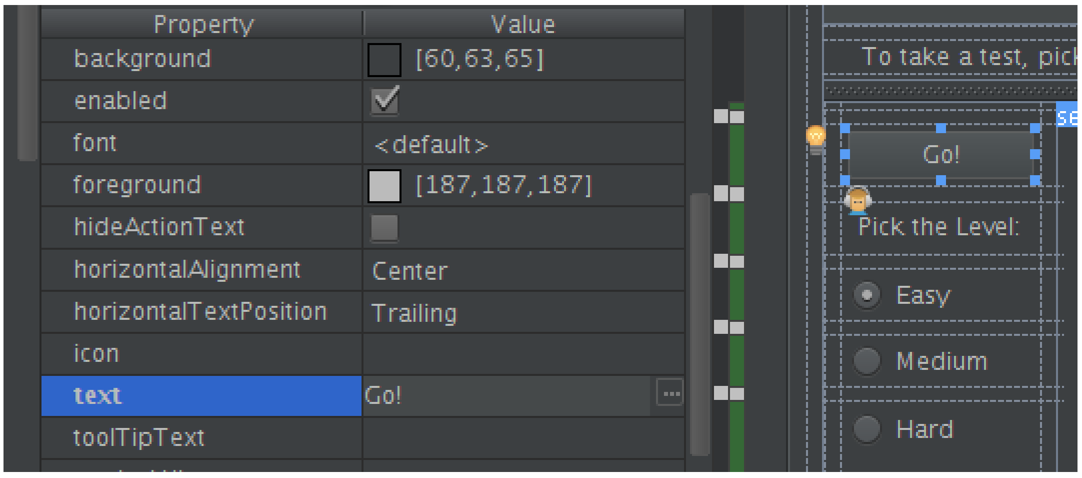
3. Add the following dependency to your project's *pom.xml*, it is necessary to operate with Idea's annotations for internationalized content in code:

```
<dependency>
	<groupId>org.jetbrains</groupId>
	<artifactId>annotations</artifactId>
	<version>13.0</version>
</dependency>
```

4. Create a helper class to serve a single point for all calls to international string resources. In my project, this is `org.akrasnyansky.typetest.i18n.I18nHelper`:

```
public class I18nHelper {
	private static final String BUNDLE_NAME = "i18n.TypingTest"; //NON-NLS
	public static String message(@PropertyKey(resourceBundle=BUNDLE_NAME)
String key, Object... params) {
		ResourceBundle bundle = ResourceBundle.getBundle(BUNDLE_NAME,
Locale.getDefault());
		String value = bundle.getString(key);
		if (params.length > 0) {
			return MessageFormat.format(value, params);
		}
		return value;
	}
}
```

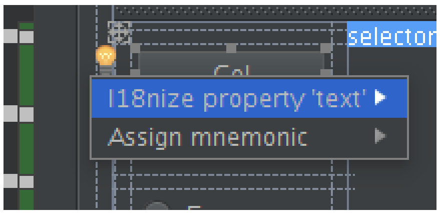
4. Then run “Analyze – Inspect Code” for the whole project. Open the *“Internationalization issues”* node and firstly find **“Hard coded string”** sub-node, and look for any **“Hard coded string literal in form”** warnings. When you click such, you are taken to the problem spot in the form editor:

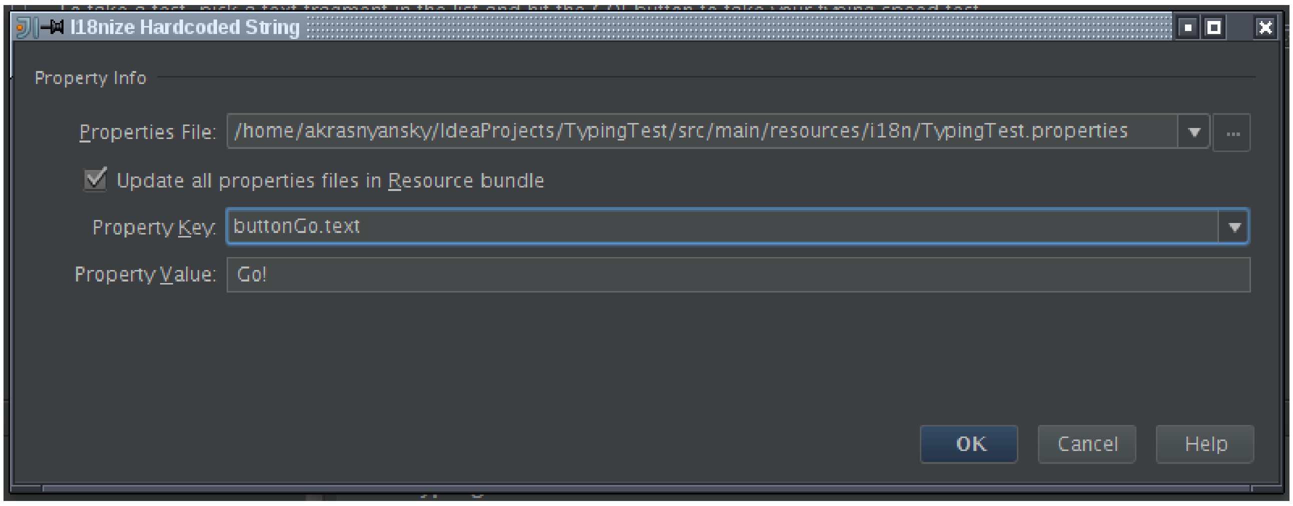


Property	Value
background	[60,63,65]
enabled	✓
font	<default>
foreground	[187,187,187]
hideActionText	
horizontalAlignment	Center
horizontalTextPosition	Trailing
icon	
text	Go!
toolTipText	

Pic 31. A hard-coded string internationalization in the form.

Click the light-bulb and pick the following item in the pop-up:





Properties File: /home/akrasnyansky/IdeaProjects/TypingTest/src/main/resources/i18n/TypingTest.properties

[✓] Update all properties files in Resource bundle

Property Key: buttonGo.text

Property Value: Go!

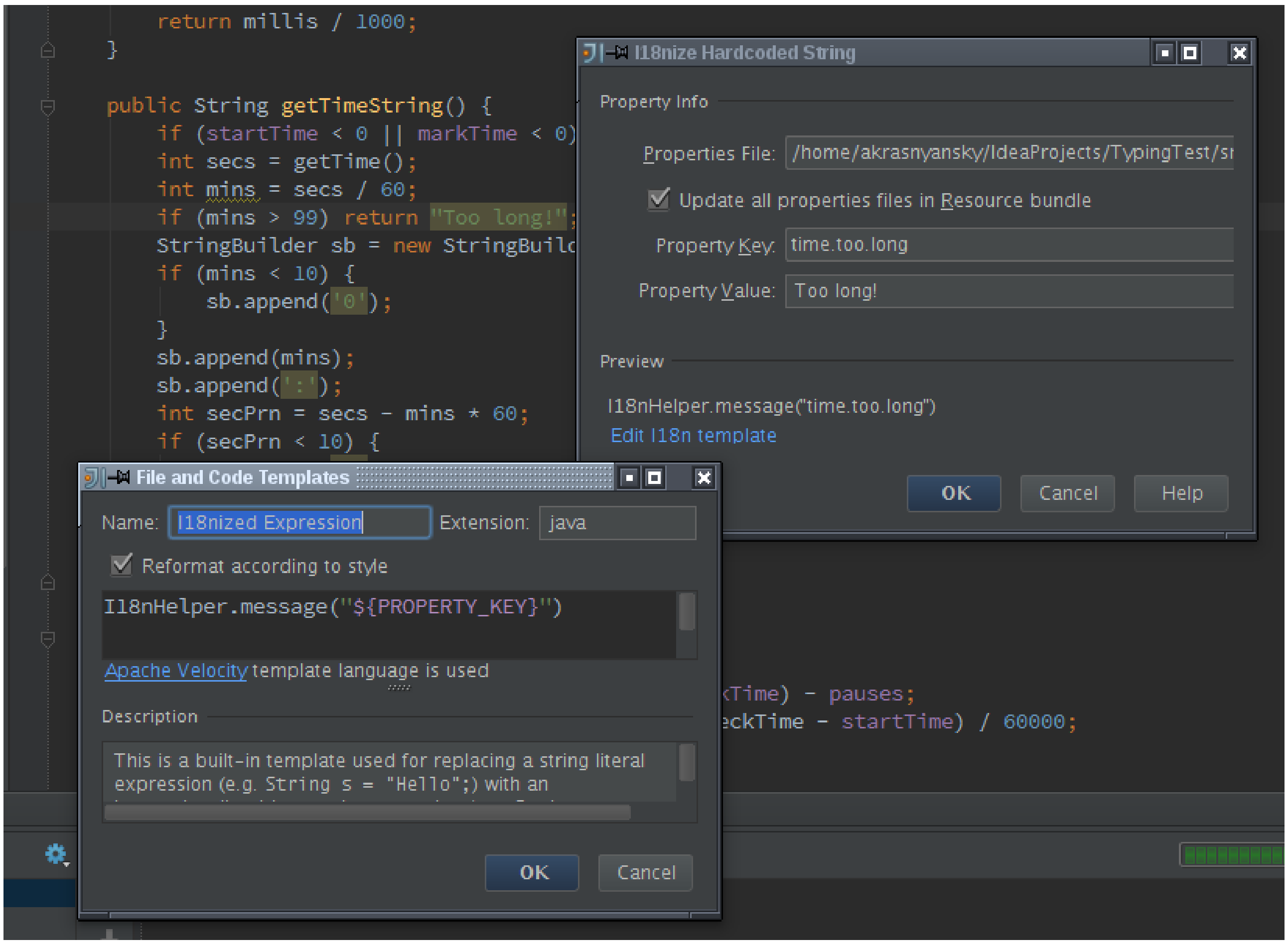
Pic 32. Entering a new property key.

Enter the new property key and value (for how it should be titled in English) and the new property will appear in the `TypingTest.properties` file:

```
buttonGo.text=Go\!
```

Do the same thing for all strings that appear in the form.

5. Now go over all hard-coded literals that appear across java code. You can set various resolution methods for the found issues: either suppress it with a @NonNls annotation or choose to internationalize. At first call of the Idea's helper, define the template as shown,



```
        return millis / 1000;
    }

    public String getTimeString() {
        if (startTime < 0 || markTime < 0)
        int secs = getTime();
        int mins = secs / 60;
        if (mins > 99) return "Too long!";
        StringBuilder sb = new StringBuilc
        if (mins < 10) {
            sb.append('0');
        }
        sb.append(mins);
        sb.append(':');
        int secPrn = secs - mins * 60;
        if (secPrn < 10) {
```

```
:Time) - pauses;
:ckTime - startTime) / 60000;
```

I18nize Hardcoded String

Property Info

Properties File: /home/akrasnyansky/IdeaProjects/TypingTest/sr

☑ Update all properties files in Resource bundle

Property Key: time.too.long

Property Value: Too long!

Preview

I18nHelper.message("time.too.long")

Edit I18n template

File and Code Templates

Name: I18nized Expression Extension: java

☑ Reformat according to style

```
I18nHelper.message("${PROPERTY_KEY}")
```

Apache Velocity template language is used

Description

This is a built-in template used for replacing a string literal expression (e.g. String s = "Hello";) with an

Pic 33. Internationalization of a hard-coded line. Defining the I18n template.

by clicking on *“Edit I18n template”* link in the first box. After you do, the helper will use the calls to the static method `I18nHelper.message()` everywhere.

6. When no hard-coded lines which should undergo internationalization is left, add a new property file **`TypingTest_ru.properties`** to the same package folder where `TypingTest.properties` is. IntelliJ Idea will automatically recognize it as a new language pack added to the same bundle. Open it and go to *Resource bundle* tab in the editor. You'll see the list of the properties, see what properties are missing and two editors for English and localized version of each property. Just fill those in with translation.

7. The last thing we need is the feature to switch between languages.

Add the following listener code to **`buttonToggleLanguage`**:

```
buttonToggleLanguage.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        if (Locale.ENGLISH.equals(Locale.getDefault())) {
            Locale.setDefault(Locale.forLanguageTag("ru"));
        } else {
            Locale.setDefault(Locale.ENGLISH);
        }
        ResourceBundle.clearCache();
        changeLocale();
    }
});
```

The `changeLocale()` call is necessary to update internationalized lines in the form components, it is a bunch of UI widget calls like:

```
SwingUtilities.invokeLater(new Runnable() {
    @Override
    public void run() {
        radioLvlHard.setText(I18nHelper.message("radioLvlHard.text"));
    }
});
```

At this point, you are done.

8. Assembly of the program into an executable jar.

We are going to use a jar packaging for the program. It offers many further options for the code distribution: it could be just uploaded to a shared space or be bundles into a Java Web start, so it could run right off some web page.

You have to define a <build> section in your pom.xml and add there the necessary build plugins. You need at least two: **`maven-jar-plugin`** and **`ideauidesigner-maven-plugin`**. In configuration of `maven-jar-plugin` define the main class as shown in the following code piece:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-jar-plugin</artifactId>
            <version>2.5</version>
            <configuration>
                <archive>
                    <manifest>

<mainClass>org.akrasnyansky.typetest.TypingTestView</mainClass>
                    </manifest>
                </archive>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>ideauidesigner-maven-plugin</artifactId>
            <version>1.0-beta-1</version>
            <executions>
                <execution>
                    <goals>
                        <goal>javac2</goal>
                    </goals>
                </execution>
            </executions>
```

```
                <configuration>
                    <fork>true</fork>
                    <debug>true</debug>
                    <failOnError>true</failOnError>
                </configuration>
            </plugin>
        </plugins>
</build>
```

Also maven won't be happy if you don't have this piece in your pom:

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
```

Go to `File – Project Structure` and make sure you don't have any libraries in dependencies that are not imported from maven. If there are any, delete them.

Invoke maven's **clean package** targets either from inside IntelliJ Idea or from command line. Check that there is no error reported by maven in the log trace, and all the JUnit tests pass fine.

Such way we have a small jar, but it depends on external libraries being on the classpath. Because the program we have made is lightweight, it makes sense to use a **“One-jar”** plugin to produce a single distribution jar which is extremely easy to run for a user.

Add the following section to your pom:

```
<!-- One-Jar is in the googlecode repository -->
<pluginRepositories>
    <pluginRepository>
        <id>onejar-maven-plugin.googlecode.com</id>
        <url>http://onejar-maven-plugin.googlecode.com/svn/mavenrepo</url>
    </pluginRepository>
</pluginRepositories>
```

And then add **onejar-maven-plugin** to <plugins> section.

```
<plugin>
    <groupId>org.dstovall</groupId>
    <artifactId>onejar-maven-plugin</artifactId>
    <version>1.4.4</version>
    <executions>
        <execution>
            <goals>
                <goal>one-jar</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

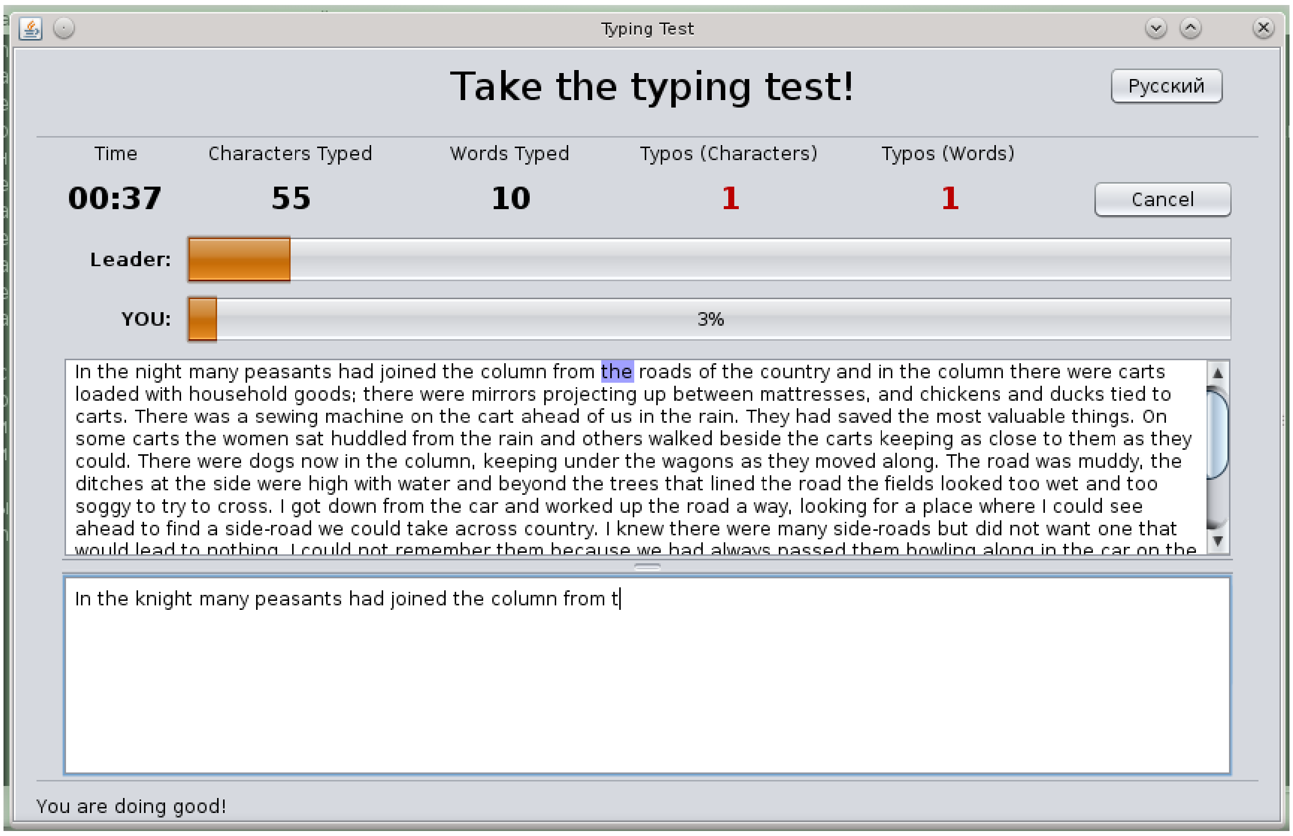
After you run

```
    mvn clean package
```

In your target directory a ready jar file appears with suffix “one-jar.jar”.

Run it with the following command line and check yourself how quickly and nicely you can type! :-)

```
    java -jar typetest-1.0-SNAPSHOT.one-jar.jar
```



In the night many peasants had joined the column from the roads of the country and in the column there were carts loaded with household goods; there were mirrors projecting up between mattresses, and chickens and ducks tied to carts. There was a sewing machine on the cart ahead of us in the rain. They had saved the most valuable things. On some carts the women sat huddled from the rain and others walked beside the carts keeping as close to them as they could. There were dogs now in the column, keeping under the wagons as they moved along. The road was muddy, the ditches at the side were high with water and beyond the trees that lined the road the fields looked too wet and too soggy to try to cross. I got down from the car and worked up the road a way, looking for a place where I could see ahead to find a side-road we could take across country. I knew there were many side-roads but did not want one that would lead to nothing. I could not remember them because we had always passed them bowling along in the car on the

In the knight many peasants had joined the column from t

You are doing good!

Pic 31. The ready application running.