R. Gaya, M. Roggenbach, CS-135 – Lab Class 9 – 9.4.

- **To be solved in groups of two.**
- **To be submitted to blackboard by Monday, 15.4., 11am.**
- **Submission: Every student individually, one pdf file.**
- **Please mark clearly with whom you are working together: there should be 2 student numbers on top of your submissions.**
- **No other formats than .pdf will be accepted!!!**

A word of warning for students using their own computer: this lab works only with Java 8! – it works on the computers in the lab, but might not work on your computers.

This lab class is an exercise in Extreme Programming. You will continue to develop the "Reaction Game" from Lab Class 8 using Extreme Programming.

There are two features you will implement. First is to display whether the user has clicked the correct square. The second is a scoring mechanism.

Following the paradigm of Extreme Programming, the implementation of a story is done in three steps:

- write the tests first, then
- write the implementation, and finally,
- test the implementation (and debug it if necessary).

Part 1: Initial setup

- Make a new JavaFX Project in Eclipse, e.g., "Reaction Game II".
- Download from Blackboard the Lab 9 versions of `GameApplication.java` and `GameTest.java`.
- Import these files into your Eclipse Project.
- Add JUnit (instructions in the collected hints) to your project.
- Add the jar-files from Lab 8, which are as well on Blackboard (see Lab 8).
- Run the JUnit tests. All three tests should pass as the functionality has now been implemented.

Submission A screen-shot of the 4×4 Game Panel and a screen-shot showing that all three test cases pass.

Part 2: Display Click Status

You are now going to implement a missing part of the game – the method that informs the player if they clicked into the correct square.

Note: the following story is for the whole of this exercise - don't rush off and start the implementation yet. It's the basis of the extreme programming approach of first writing tests, then writing code, and finally automatically testing the code.

Story: Implement the functionality to display whether the player clicked the correct square or not. To do this, implement the method `displayClickStatus` (in `GameApplication.java`) that changes the text of a `Label`.

1. Develop a new JUnit test suite in a separate file, e.g., `DisplayClickStatusTests`. The purpose of this test suite is to check that the displayed text is correct. That is, that the displayed text indicates that the player clicked the correct square only if they did so; and that the displayed text indicates that the player clicked an incorrect square only if they did so. To do this, a test should do the following:

   (a) Fix an *x*-*y* coordinate.

   (b) Display the square at that coordinate.

   (c) Make the robot click a button.

   (d) Get the displayed text as a string.

   (e) Let JUnit check if the displayed text is what you expected it to be (depending on whether or not you told the robot to click the correct button).

   Use equivalence class testing on the outcome, i.e., "correct" or "incorrect", in order to design your test suite.

   Hint: In order to click into a square, you can use the `clickOn` method from Lab Class 8.

   Hint: In order to observe what is written on the panel, you need to include the method `observeDisplay` from `GameTest.java` into the class of your test suite.

   The methods `getLastClickedSquareX` and `getLastClickedSquareY` have already be implemented in Lab Class 8, these allow you to find out what square was last clicked.

   The methods `getDisplayedSquareX` and `getDisplayedSquareY` are provided, these allow you to find out what square is currently being displayed

2. Develop the code for the method `displayClickStatus` to report the result of one click. The file `GameApplication.java` already includes an empty method for this.

   Hint: In order to set the text of the Label, use the code

   ```
   resultBox.setText(s);
   ```

   where `s` is the string you want to display.

3. Test the code and verify that it works as it should.

Submission The code of your tests and a screen-shot showing that all your test cases pass.

Part 3: Scoring

You are now going to implement a new feature: a score counter.

Note: the following story is for the whole of this exercise - don't rush off and start the implementation yet. It's the basis of the extreme programming approach of first writing tests, then writing code, and finally automatically testing the code.

Story: implement a score counter that counts the number of correct squares the user has clicked within the time limit. To do this, implement the method `displayResult` (in `GameApplication.java` that changes the text of the Label to be the score.

1. Create a new JUnit file for the test suite, called e.g. CheckScore. This test suite shall demonstrate that the method `displayResult` displays the correct score value in the Label.

   To this end, your test suite should cover three scenarios: the user has not clicked at all, the user has clicked at least once correctly, and, finally, several correct and several incorrect clicks.

   The test idea is as follows:

   (a) Fix an x-y coordinate.

   (b) Display the square at the coordinate.

   (c) Make the robot click a button.

   (d) Repeat as needed:

      i. The previous click will have made a new random square red. Get the coordinates of this square by using the methods `getDisplayedSquareX` and `getDisplayedSquareY`

      ii. You can now either click this square; or intentionally avoid clicking it by clicking another *different* square.

   (e) Call the method `Thread.sleep(8500)`, which waits for 8500 milliseconds, which is longer the time of the game.

   (f) Make a final click (this causes the Label to be updated).

   (g) Let JUnit check that the displayed result is as expected. Remember the displayed text is a string (and not an integer).

   After the time is up, you need to click once more into the panel in order to trigger a change of the display. This click does not count to the score as it is out of time (this logic is already implemented).

   You can once again use the method `observeDisplay` to access the displayed text and the method `clickOn` to click a button.

   Hint: For a test strategy, i.e., an effective way of 'proving' that things work correctly, think about what can happen: if the user clicks into a correct square, some counter shall be increased; if it was not the correct square, there is no increase. Also, the counter should be initialised correctly.

2. Implement the method `displayResult` and change the code of other methods. You can use the global variable `result` (declared in `GameApplication` as an integer) in order to count the score. Reflect on the question, which method actually decides if a click was right or wrong. This method might be a good place to handle the `result counter. The method displayResult is called automatically once the time limit is reached. This method should display the current score in the Label.`

3. `Test the code and verify that is works as it should.`

Submission The code of your tests and a screen-shot showing that all your test cases pass.

Fun Submission A picture of you playing the game.

Computer Instructions

1 Getting a terminal

Under the “Specialist Apps”, open the folder “College of Science”. Within this folder, open the folder “Computer Science”. There, you find various version of the “Java Development Kit”, which provides a terminal configured for Java. For our purposes, any of them will do.

2 Changing to the right directory

Typing `p:` changes the drive to the one with your home directory.

The **Change Directory - Select a Folder (and drive) command:**

```
Syntax
      CD [/D] [drive:][path]
      CD [..]

Key
   /D : change the current DRIVE in addition to changing folder.
Examples

   To change to the Desktop.
   p:\> cd Desktop

   To change to the parent directory.
   p:\Desktop> cd ..
```

3 The Java commands

These commands can be typed in the command line of the terminal:

1. `javac <filename.java>` compiles a program.
2. `java <filename>` executes the .class file.
3. `javadoc -d <directory-name> -version -author <filenname.java>` produces the documentation.

4 Recommended editor

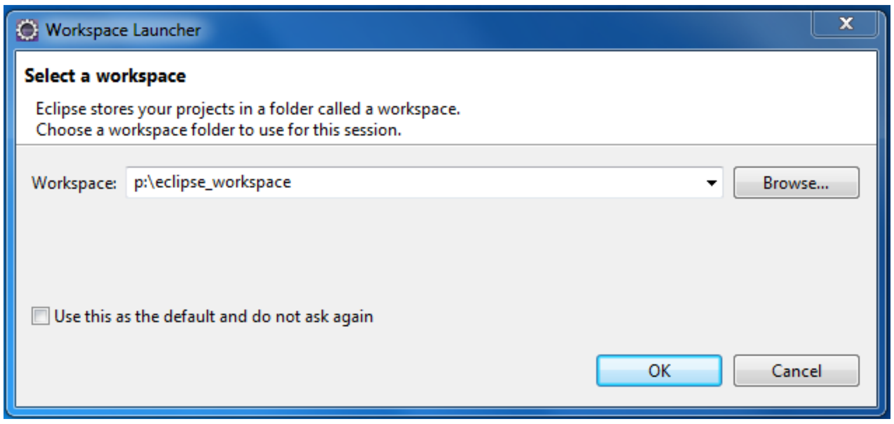
Under the “Specialist Apps”, open the folder “College of Science”. Within this folder, open the folder “Computer Science”. There, you find the program “Notepad++”, which provides a reasonable environment to edit Java programs.

5 Making a screen-shot

Click on ‘Start’, type ‘Snipping Tool’ in the search field, press ‘enter’. Use the tool.

6 Eclipse

Under the “Specialist Apps”, open the folder “College of Science”. Within this folder, open the folder “Computer Science”. There, you find the program “Eclipse”. When you start Eclipse you might be asked for the workspace path. This path should be set as follows:


Workspace Launcher

Select a workspace

Eclipse stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: p:\eclipse_workspace

6.1 Making a new project

1. Click `File` → `New` → `Project` → `Java Project`.
2. Typing a good project name i.e. `Sphinx`.
3. Click `Finish`.

6.2 Importing a file into a project

1. Expand your project, say `Sphinx` in the left hand panel (Package Explorer),
2. Right click the `src` folder, click import.
3. Select `File System` under `General`, click `Next`.
4. Locate the directory containing the `Sphinx.java` file, click `OK`.
5. Check the file, e.g. `Sphinx.java`, in the right hand list, Click `Finish`.

6.3 Running a program

You run a program, e.g., `Sphinx.java`, by clicking the play icon. This may bring up a wizard where you need to select to run a `Java Application`. You may need to show the `Console` view by clicking `Window` → `Show View` → `Console`.

6.4 Activating JUnit4 for a project

1. Right-click on your project and select `Properties`.
2. Click on `Java Build Path`.
3. Select `Libraries`
4. Select `Add Library`.
5. Select `Junit`.
6. Click on next, select the Junit Version `JUnit 4`.
7. Click `Finish.`
8. Click `OK.`