R. Gaya, M. Roggenbach, CS-135 – Lab Class 8

- **To be solved in groups of two.**
- **To be submitted to blackboard by Monday, 8.4., 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 and the JavaFX library installed! – 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 download a "Reaction Game" from the internet. The idea is that you need to click the red square within a certain time frame – to test your reaction time. However, the implementation of this game is not finished. You will finish the implementation using Extreme Programming.

After getting started (Part 1), you have two stories to implement (Parts 2 & 3). Remember, that a story is a "unit of customer visible functionality". In Part 2, the story will be to highlight a red square inside the GUI; in Part 3 the story will be to detect which square the user has clicked.

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: Warm-up

- Make a new Java Project in Eclipse, e.g., "Reaction Game".
- Download the files `ReactionGameStart.zip` and `External JARs.zip` from

  `http://www.cs.swan.ac.uk/~csmarkus/Tools/`.

  Unpack these files by double clicking on them – this results in the creation of two directories.
- Create an new `JavaFX Project`, name it e.g. "Reaction Game". Ensure that the JRE is set to JavaSE-1.8.
- Delete the `application` package within this new project.
- Import `GameApplication.java` and `GameTest.java` into your project.
- Add JUnit to your project.
- Add libraries to your project by
  - right click on the projec

- choose Properties
- choose Java Build Path
- click on Librarires
- select Add External JARs
- locate all the four .jar files from the folder `External JARs`, click open
- click apply
- click ok

- Run the program (the file `GameApplication.java` contains the main method. A game panel should appear. You can't really play the game yet, as critical parts are not implemented. But try this: click into the Game Panel, more than 8 seconds after the panel appeared on your screen, the panel should read "Time up!".
- Run the JUnit tests. Two of three test should fail, one might pass or fail, depending on the random initialization.

Submission A screen-shot of the Game Panel showing "Time up!" and a screen-shot showing the results of the three tests.

Part 2: Turning squares red

You are now going to implement a missing part of the game – the method that actually turns a square red.

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

Story: implement a method `displaySquare(int x, int y)` in `GameApplication.java` that sets the square at position (x, y) red on the panel.

There are two things that could go wrong in the implementation of this method. The first is that the method might make too many squares red, or not enough squares red. The second, is that the wrong square might be coloured red. Thus, we need two tests suites: one to test exactly one square has been turned red, and one to check that the correct square has been turned red.

1. Develop two JUnit test suites in two separate files. Note that this file needs to include the method `start`. `start` is a method required by JavaFX to be present when a panel is created – as we want to do for each of the test cases.

   - Create a new JUnit file for the first test suite, called, e.g., CountSquares. The purpose of this test suite is to check that after a call of the `displaySquare` method exactly one square on the panel is red.

     To this end, write 16 test cases (one for each square of the 4×4 panel) for the property "exactly one square is red". In order to test this, there is an method `getNumberReds`. This method observes, how many square are coloured red on the panel. This method has already been implemented in `GameTest` as part of the testing infrastructure.

     A sample test case for checking that there is exactly on red square on the panel is `testTemplateOne` in `GameTest.java`:

```
00 @Test
01 public void testTemplateOne() throws InterruptedException {
02     Thread.sleep(500);
03     int x = 1;
04     int y = 1;
05     game.displaySquare(x, y);
06     Thread.sleep(500);
07     assertEquals(1, getNumberReds());
08 }
```

Line 5 creates a game panel on the screen with 4×4 grey squares including, hopefully, exactly one red square. This square is chosen via the method under test `displaySquare`, that you shall implement later. Line 7 checks that there is exactly one red square on the panel.

How do we know we coded our tests correctly? How do we check the integration of JUnit code and the SUT? We can run our tests, but they will (should) all fail as the method `displaySquare` has not been implemented yet. However, if we change the method `getNumberReds` to **always** return 1, then all our tests should pass. Do this now: change the method `getNumberReds` to always return 1 (comment out the current code) and check all your tests pass. Once this is the case, change the code back to the original. How does this help us? Well if, once we have implemented the method, we find that if a test fails, we can be sure it is because the implementation is wrong and not the encoding of the tests (i.e., the test interface).

- Create a new JUnit file for the second test suite, called, e.g., DisplayRedSquare. This test suite shall demonstrate that the correct square has been set to red. To this end, apply weak normal equivalence-class testing with classes:
  - $X0 = \{0\},\ X1 = \{1\},\ X2 = \{2\},\ X3 = \{3\}$
  - $Y0 = \{0\},\ Y1 = \{1\},\ Y2 = \{2\},\ Y3 = \{3\}$

  Hint: Remember that weak normal equivalence-class testing uses the single fault assumption.

  A sample test case for this is `testTemplateTwo` in `GameTest.java`:

```
0 @Test
1 public void testTemplateTwo() throws InterruptedException {
2     Thread.sleep(500);
3     int x = 1;
4     int y = 1;
5     game.displaySquare(x, y);
6     Thread.sleep(500);
7     assertEquals(x, game.getDisplayedSquareX());
8     assertEquals(y, game.getDisplayedSquareY());
9 }
```

Line 5 creates a game panel on the screen with 4×4 grey squares including, hopefully, a red square hopefully at position 1,1. Lines 7 and 8 check that the red square has the X-coordinate 1 and the Y-coordinate 1, resp. To this end, the methods `getDisplayedSquareX()` and `getDisplayedSquareY()` of `GameApplication.java` return the coordinates of which square has been set to red on the panel. They already have been implemented in `GameApplication.java`.

2. Develop the code for the method `public void displaySquare(int x, int y)` to set a red square on the panel. The file `GameApplication.java` already includes an empty stub method for this:

   ```
   public void displaySquare(int x, int y) { // to be implemented }
   ```

   - There is an ArrayList of buttons. However, the array list is a one-dimensional list, while the screen layout is two dimensional. Each button on the screen is held within the ArrayList with the following mapping:

     Screen Arrangement:

(0,0)	(1,0)	(2,0)	(3,0)
(0,1)	(1,1)	(2,1)	(3,1)
(0,2)	(1,2)	(2,2)	(3,2)
(0,3)	(1,3)	(2,3)	(3,3)

     Representation as a data structure (ArrayList):

(0,0)	(1,0)	(2,0)	(3,0)	(0,1)	(1,1)	(2,1)	(3,1)	(0,2)	(1,2)	(2,2)	(3,2)	(0,3)	(1,3)	(2,3)	(3,3)
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

   - In order to get an element from an ArrayList, there is a method `get` (on ArrayLists).
   - In order to find the index in the array of a button you wish to change the colour of (given grid coordinates (x, y)), you might want to use the formula

     $$(y \times \texttt{GRID\_SIZE}) + x \ .$$

     `GRID_SIZE` is the constant that determines the size of the panel. For example, given a button at grid coordinates (3, 1), we find that it is stored in the ArrayList in index 7.
   - To set the colour of a button, apply the method `setStyle(BUTTON_STYLE_RED)` to the object `button`.

3. Run your test suites and verify that the method `displaySquare` works as it is supposed to.

Submission Screen-shots showing that all your test cases for `displaySquare` are passing.

Part 3: Detecting clicked squares

You are now going to implement another missing part of the game – the methods that allow us to know which square was clicked last. These methods will not be used by the game just yet – next weeks lab will use them.

Story: implement methods `getLastClickedX` and `getLastClickedY` in `GameApplication.java` that recognise the X and Y coordinates of the square which the user clicked last.

1. Create a new JUnit file for the test suite, called e.g. CheckClicks. This test suite shall demonstrate that the methods `getLastClickedX` and `getLastClickedY` return correct values. To this end, apply weak normal equivalence-class testing with classes
   - $X0 = \{0\},\ X1 = \{1\},\ X2 = \{2\},\ X3 = \{3\}$
   - $Y0 = \{0\},\ Y1 = \{1\},\ Y2 = \{2\},\ Y3 = \{3\}$

Hint: Remember weak normal equivalence-class testing uses the single fault assumption.

A sample test case for `getLastClickedX` and `getLastClickedY` is `testTemplateThree` in `GameTest.java`:

```
0  @Test
1  public void testTemplateThree() throws InterruptedException {
2      Thread.sleep(500);
3      int x = 1;
4      int y = 1;
5      game.displaySquare(x, y);
6      clickOn("#button5");
7      Thread.sleep(500);
8      assertEquals(x, game.getLastClickedSquareX());
9      assertEquals(y, game.getLastClickedSquareY());
10 }
```

Line 5 creates a Game panel with one red square (yes, we already tested that this works!). In line 6 the call `clickOn("#button5")` makes a software robot press the 6-th button in the ArrayList `buttons`. Lines 8 and 9 check if the detected X and Y coordinates are as expected.

2. Implement the methods `getLastClickedX` and `getLastClickedY` in the class `GameFrame`:

```
public int getLastClickedSquareY() {
    // to be implemented
    return -1;
}

public int getLastClickedSquareX() {
     // to be implemented
     return -1;
}
```

Note: The -1 return values are dummy return values and should not be used in your real implementations.

Hint To calculate the Y value, you might want to use the formula

$$\texttt{clickedSquare} \,/\, \texttt{GRID\_SIZE}$$

and to calculate the X value the formula

$$\texttt{clickedSquare} \,\%\, \texttt{GRID\_SIZE}$$

might come handy.

Here, `ClickedSquare` is a global integer variable in `GameFrame`. It stores the index of the button in the ArrayList that was clicked last.

3. Run your test suite and verify that your implementation works as it is supposed to.

Submission Screen-shots showing that all your test cases for `getLastClickedX` and `getLastClickedY` are passing.

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

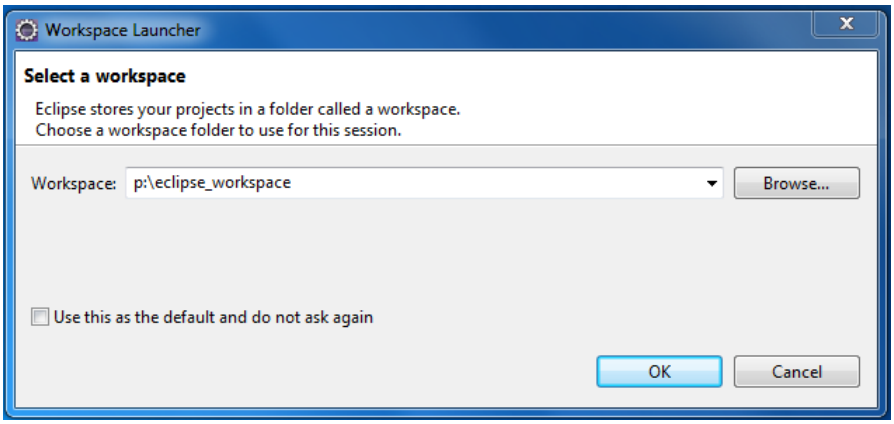
Under the "Specialist Apps", open the folder "College of Science". Within this folder, open the folder "Computer Science". There, you find the program "Notepad++", which provides a reasonable environment to edit Java programs.

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:



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.`