CS 301 Project Assignment 5 Monday, October 30, 2017

To deliver: Release 5.0 in your SVN repository
To download: -

Hard & final submission deadline: November 6, at 11.59 PM

Learning Objective:
- Refactoring in Eclipse
- Encapsulation of Graphics Code ("Encapsulate what changes")

"All problems in computer science can be solved by another level of indirection" David Wheeler

Motivation
We discuss in class refactoring as a necessary step in test driven design to achieve a good quality of code. The key observation is that a series of small changes which consists of individual steps that are of low complexity and possibly well supported by an IDE like Eclipse can be performed to obtain a cleaner and clearer code base without introducing errors in this transformation. Refactoring is a transformation of code that does not introduce new functionality or does not change observable behavior of a program.

The goal of this assignment is to get more experience in refactoring. To make this exercise worthwhile, we will use it to localize and encapsulate access to graphics in the maze program. On an Android platform, java.awt and java.swing packages do not exist. So we will need to replace those by some corresponding Android-supported packages.

In order to port our existing maze project to Android, we will:
- redesign and recode the user interface (next Project assignment 6), which will make MazeApplication.java obsolete for our new Android app. It will also move all graphical UI functionality away from Maze.java and its viewers into a new UI with screens and a switching between screens via message passing (Android intents).
- recode the graphics part when moving falstad/*.java classes to the AMaze Android project (Project assignment 7)

Note that the topic of this project assignment is a preparation for the last item above.

Let's check a list of classes that have the AWT problem:
1. MazeApplication.java and SimpleKeyListener.java: Both classes will not be used in an Android app. So we need not change anything here (unless changes elsewhere imply changes for this class).
2. **MazeController.java** imports java.awt.* to have the MazePanel as an attribute. This class needs to be considered.
3. **Viewer interface** imports awt because methods of its implementing classes operate with a graphics objects. This needs to be considered.
4. **DefaultViewer** imports jawa.awt.Graphics just because methods of its subclasses operate with a graphics objects. This class needs to be considered together with its subclasses FirstPersonDrawer and MapDrawer.
5. **FirstPersonDrawer.java** imports java.awt.Color, java.awt.Graphics, java.awt.Point; This class needs to be considered!

6. **MapDrawer.java** imports java.awt.Color, java.awt.Graphics; This class needs to be considered! HINT: One can encapsulate Colors in MazePanel and have a getColor method return an Object to obscure that a Color is returned.
7. MazeView.java imports awt classes. It will become obsolete in Android such that changes should only be a consequence of changing the Viewer interface.
8. **RangeSet.java**:import java.awt.Point; This class needs to be considered! HINT: Point is actually misused here and replacing it by something else is the easiest fix.
9. **Seg.java**:import java.awt.Color; This class needs to be considered!
10. **MazeFileReader.java**: import java.awt.Color. The color setting for elements are stored in a file and read back in with the file reader. The Seg.setColor(Color) method causes this dependency.
11. **MazePanel.java**: import java.awt.Panel: This class needs to be considered.

To facilitate the graphics part, we isolate the graphics as much as we can into a single class and move all functionality that requires objects from AWT into this class. For instance the buffer image, fonts, graphics object and so forth will be attributes of this class such that other existing classes call methods for graphics by calling such methods on this wrapper object. Do we need a new class for this? May be not as the MazePanel class already wraps the AWT Panel. Let's revise the MazePanel as a graphics wrapper class.

The wrapper provides its functionality by delegation, it is basically the “middle-man” who gets all of its work done with the help of some assistant like AWT and Swing behind it. To implement this, one simply needs to know which methods to call inside the wrapper class. In the ordinary Java environment, this will simply take place by using existing functionality of AWT entities. So we basically move existing functionality around, we need to relocate methods and adjust parameter types as necessary.

If you manage to get this right, you will be able to move existing classes towards Android such that only the MazePanel class will require a reimplementation in the new Android environment to work as the backend behind a newly designed UI.

The benefit of the wrapper is that you can do all necessary adjustment within a single encapsulated class such that you can test the functionality locally and then use a working version of the wrapper on Android to make your maze code work in that environment.

Of course existing test classes are still supposed to work (possibly after marginal adjustments).

Requirements

Add new methods to the MazePanel class to support necessary graphics methods (like drawing a line or a polygon). Exercise Eclipse methods for refactoring to achieve this goal.
Eliminate import statements for awt and swing packages from certain classes (see check list above) and move graphics methods to the MazePanel class.

To do list:

- Check supported refactoring functionality in Eclipse (Eclipse->Refactor pull down menu). Work on a copy of some classes first to see how functions work, e.g. how to rename variables and methods, how to extract methods, how to push methods up and down the

inheritance hierarchy, how to extract an interface and so on. Use those files to figure out how things work and then delete those damaged copies.
- Transform your code base in a sequence of steps to move functionality for graphics into the MazePanel class and have that class provide corresponding functionality on its own such that **classes listed as 2-10 (in bold)** above become independent from AWT objects (thanks to the dependence on the MazePanel). Also read code carefully and fix inappropriate usage of AWT objects: there is RangeSet.intersect(Point p) which uses the Point class but not for graphics purposes. For this case, check the code and replace Point by something else that also does the job but does not misuse a graphics class for that.
- Clean up import statements in your classes.
- Update your Java maze project in your svn repository.
- For **Bonus Points:**
    - clean up your new classes BasicRobot, Wizard, Wallfollower, Pledge
    - use helper methods to avoid repetitive code
    - use encapsulation: fields are private, access to fields is controlled with corresponding get/set methods
    - functionality that exclusively works on data from a different class is relocated to that class
- Produce a tag release 5.0

Grading:

You will receive points
1. if your SVN repository contains a release 5.0 and at least 10 revisions after project 4 with meaningful comments.
2. if Viewer, FirstPersonDrawer, MapDrawer, RangeSet and Seg classes do not directly rely on AWT any more.
3. if your MazePanel class encapsulates access to AWT objects like an Image, Fonts, Colors and so forth.
4. if your maze application works with the new MazePanel class such that one can play the Maze game as before
5. if your junit test cases operate with no errors or failures.
6. BONUS points: if your BasicRobot, Wizard, Wallfollower, Pledge classes are encapsulated, i.e. fields are private, access to fields is controlled with corresponding get/set methods; functionality that exclusively works on data from a different class is relocated to that class; helper methods are used to avoid repetitive code etc.