Assignment #7

Practice with Refactoring in Eclipse

CS 4354 Summer II 2016
Instructor: Jill Seaman

Due: in class **Tuesday, 8/9/2016** (upload electronic copy by 11:50am).

---

1. **SETUP**: We'll use a Monopoly game written in Java (complete with JUnit testcases) developed at North Carolina State University. I have modified the code slightly for the purposes of this assignment.

   Get a copy of this project's source files (Monopoly4.zip) from the class website. Create a new Java project, then import these files as follows:
   - Create a new Java project.
   - Make sure that project has a folder named src in it. If not, select the project name, then right-click and choose New -> Source Folder. In the pop-up window, name it src.
   - Right-click on the folder src and choose Import, General, **Archive File**
   - Browse and choose the Monopoly4.zip file, and then hit Finish.

   You should see the java files in two packages under src (once you open src). There will be errors in the files until you add JUnit.

   Set up and run JUnit tests:
   - Right-click on the Project, choose Build Path > Add Library > JUnit. Then select either JUnit 4 or JUnit 3 (the code is written in JUnit 3 style).
   - Right-click on the Project, choose Run As > JUnit Test (you might need to Build Project first, in older Eclipses) Make sure all the tests run and pass (44/44).

   Let's play Monopoly!
   Try clicking on the GUI package (edu.ncsu.monopoly.gui), then right-clicking and choosing Run As Java Application….

   Now let's Refactor:

2. **Rename class field:**
   - Locate and open class Cell (which represents a square on the game board) in the monopoly (edu.ncsu.monopoly) package.
   - Highlight the name of the class and select Navigate > Open Type Hierarchy to see its subclasses in the Type Hierarchy pane. You can also see that Cell has a

field named **player** of type Player. This name is not very descriptive of the role of the Player with respect to the Cell.

- This Player is really the owner of that Cell, so click on "player" and use the Refactor > Rename option to change the name to "owner" (click on the triangle to Open Rename Dialog). Select the boxes to update references, textual occurrences, and to rename the setters and getters. Use the Preview option to see what will change before you select OK to finish.
- Save and Build (if necessary) and Run the JUnit tests again to make sure nothing is broken.

3. Extract Local variable:

- Go to GameBoard.addCell(PropertyCell). See that the expression `cell.getColorGroup()` is used twice? Highlight to select one of those usages and then use Extract Local Variable from the refactoring menu. Note that Eclipse suggests names for the local variable.
- Note what options are offered, and use Preview to make sure you understand what will change before you carry out the refactoring.
- Run JUnit tests to help confirm nothing is broken.
- Is it always OK to do this to a function call like this? Could it change the behavior of the program?

4. Extract Method:

- Go to GameMaster.btnGetOutOfJailClicked(). Select all of the statements inside the **if** block (lines 73-79), and use Extract Method. Call the new (private) method `setAllButtonsDisabled`. Make sure it will replace the additional occurrence of these statements in the previous method (btnEndTurnClicked) with a call to the new method as well (use the preview to verify before making changes).
- Go to PropertyCell.getRent(). Notice the for loop. Extract a method (call it `rentForMonopolies`) containing the for loop (lines 29-33) and optionally the declaration of the array named `monopolies` that occurs before it (lines 28-33). Choose the one that gives you the FEWEST parameters.
- Run JUnit tests to help confirm nothing is broken.

5. **Extract Subclass:**

If you investigate the Cell hierarchy (using the Type Hierarchy tab, double-click on the classes listed below Cell to open them in the editor) you may notice that the fields `owner` and `available` (and their getters and setters) are used only in three subclasses: `PropertyCell`, `RailRoadCell` and `UtilityCell` (but there are a total of 8 subclasses). We will use a Refactoring called "Extract Subclass" to make a special subclass of Cell that will be the superclass of these three classes. This must be done in steps:

(1) Click on one of the three subclasses that uses the owner and available and use the Refactor > **Extract Superclass** option. Call it `OwnedCell`. Be sure to Add… all three subclasses. This will create the new class and add it to the Cell type hierarchy. Use preview to see the changes. Finish and check for compiler errors.

(2) Use Push Down Field to push the **`available`** field (from the Cell class) and its getters and setters to the OwnedCell class. After you select Preview, un-check the subclasses you do not want to receive the pushed down field (hint: `available` is used only in the OwnedCell subclasses) (but keep Cell checked!). If you don't uncheck the subclasses, after the refactoring is done, you'll have to edit each of the non-OwnedCell classes to remove the field and getters/setters! Or use Edit > Undo to rollback the refactoring and do it again correctly.

(3) Now when the project builds, you will have some compiler errors. Fix them:
GoCell: delete the statement. This will be unnecessary shortly.
GameMaster: Hover over the call to isAvailable() (underlined in red). Click on "create method isAvailable() in type Cell". It will take you to a newly added default definition of isAvailable() in Cell. It should return false. This is exactly what you want. This method is overridden in the OwnedCell class to return the value of the attribute field in those subclasses. Save the Cell.java class.
Player: This still has an error because setAvailable is called on a Cell. Hover over the call to setAvailable() (underlined in red). Click on "Add cast to c". It will add an explicit cast on c to OwnedCell, which again is exactly what we want. if c.isAvailable() is true, it must be an OwnedCell (it is automatically false for other kinds of Cells). Save Player.java.

(4) Save changes (build) and Run JUnit tests (fix any mistakes).

(5) Redo steps (2)-(4) to Push Down the **`owner`** field.
In **Player**, don't cast, but rather change the parameter type to OwnedCell (and save your changes). Then find where buyProperty and SellProperty are called (from GameMaster). Hover over the underlined errors, and select "cast argument property to OwnedCell". Then put these two statements inside an if that makes sure that property is an instance of OwnedCell. [Note that a more complete solution is to make TradeDeal contain an OwnedCell (in place of its propertyName) because only OwnedCells can be Traded].
In the **GUI test cases**, you do not need to cast, just use the proper variable for calling getOwner.

Look at the code for sellProperty and buyProperty in the Player class. Note the bad smell where it checks instances for one of the three OwnedCell subclasses. We might be able to use refactoring and polymorphism to move the subclass-specific code to the subclasses. (But that exercise is left for another time . . .).

NOTES:

- This assignment is to be done during class on Tuesday 8/9 in groups of 2-3 people.
- You should use Eclipse (it offers the necessary automatic refactorings).
- Make sure your code compiles and that the JUnit tests still pass before submitting it.

- I will make a list of who is working in each group during the lab exercise.

Submit:

Please combine the *.java files from the completed project into a single zip file (assign7.zip). Submit an **electronic copy**, using the Assignments tool on the TRACS website for this class, before the end of class. Submit only one file per group!