AIT 642 / COSC 603

Software Maintenance & Testing

Spring 2016

**Project #2 – Refactoring and Design Smells**

*“Bug fixing often uncovers opportunities for refactoring. The very fact that you're working with code that contains a bug indicates that there is a chance that it could be clearer or better structured.”*

- Paul Butcher

**Directions:** As we discussed in class, refactoring is a central part of software maintenance to keep code from decaying to such a point that it is not maintainable – especially if agile engineering is used as the software engineering process. Software refactoring takes small steps to improve code structure and maintainability by removing bad *design smells*. A key point when doing refactoring is that the software should pass its unit tests before **and**after refactoring. That is, refactoring is only altering the *structure* of the code, not the meaning.

This project has several tasks that will continue to get you to setup a software development environment comparable to what is used in industry when developing large software systems necessitating high software quality and perform a few tasks to increase the software maintainability/quality. For this project, you should work independently. In many of the tasks, I will ask you to write about what you find or answer some brief questions (denoted in italicized bold). You should maintain a document that collects your experiences/answers throughout the project, clearly labeling each task, to make it easier for you to submit the report at the end. **Please read through all instructions prior to starting**.

**Goals:** The intention of this project is to familiarize yourself with several software development tools (Eclipse, Git, GitHub, GitHub Desktop, Javadoc, Eclipse Refactoring, JUnit, and JDeodorant) and gain experience in software quality assurance through using tools to enable refactoring, documentation, configuration management, etc. In doing so, you should develop a better understanding software quality assurance tasks that can be undertaken throughout the software development lifecycle – quality assurance isn’t simply left to testing!

**Points:** 15

**Deadline:** Friday, March 4, 2016 11:59pm checked in to your GitHub account

**Language Requirements:** Java, the Eclipse IDE and GitHub must be used for this project.

**Task 1 – GitHub Repository.**  For this project, you should utilize the same GitHub accound but create a new repository, named *cosc603*-*lastname-project2* or *ait642*-*lastname-project2* (where *lastname* is your actual last name) and add me (cosc603spring2016) as a collaborator.

**Task 2 – Refactoring Preparation.**  As was briefly mentioned in class, refactoring is a central part of software maintenance to keep code from decaying to such a point that it is not maintainable (i.e., poor quality) – especially if agile engineering is used as the software engineering process. Software refactoring takes small steps to improve code structure and maintainability by removing bad [*design smells*](http://en.wikipedia.org/wiki/Code_smell). A key point when doing refactoring is that the software should pass its unit tests before **and**after refactoring (i.e., regression testing). That is, refactoring is only altering the *structure* of the code, not the meaning.

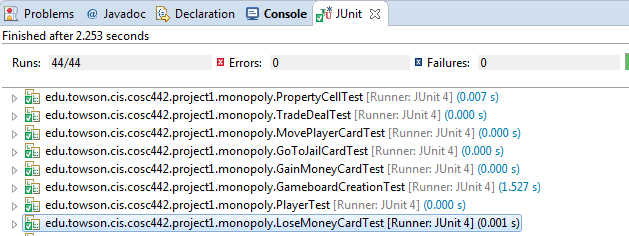
Eclipse provides many basic, automated software refactoring features. To become familiar with Eclipse’s refactoring features, you should read “Refactoring for Everyone” by David Gallardo located at <http://www.ibm.com/developerworks/library/os-ecref/>. ***Do not skip this step!***

**Task 3 – Project Setup.**  The project we will work with is that of a Java-based *Monopoly* program, originally developed at North Carolina State University. Download the Monoply.zip Eclipse project file from Blackboard (Assignments & Projects / Project 2). Then, right click in the Package Explorer window, select Import, select Existing Projects into Workspace within Eclipse and finally point to the Monopoly.zip file to import it. This should load the Monopoly source files (including JUnit source files) into your project.

You should now have an executable, GUI Monopoly program that you can run as a Java Application. Try it out (if it asks to pick a file to run, you should select Main – edu.towson.cis.cosc603.monopoly.gui).

**Task 4 – Running the Unit Tests with JUnit.** Before refactoring the Monopoly program, we need to ensure that it passes its unit test cases provided with the program. If you explore the classes within the edu.towson.cis.cosc603.monopoly package, you will notice that many classes are paired with a corresponding test class (i.e., a class that has the same name appended with Test at the end). You should take a look as several of these test source files and their corresponding Java source files to start to become familiar with them (you will be writing some yourself soon!).

We can now run the unit tests against the original Monopoly source code to ensure it passes our tests prior to refactoring it. To do so, right click on the src directory, select *Run As* and then select *JUnit Test*. This will search the src directory for all test cases (Java files ending in *Test*) and execute them against the source code. After it is done executing, this should show the results in a JUnit tab as below.



This summarizes the test results and should indicate no errors/failures. Note that you can run a single Test file by right clicking on a single test, selecting *Run As* 🡪 *JUnit Test* to see the results of a single test case. Also note, one of the test cases will print “1 5” to the console – this is normal. As you will see, the test cases pass (noted by the green bar); a common saying you will hear in Agile development/testing is “keep your code clean, keep the bar green” to indicate that as you add new features, refactor or do any type of maintenance activity, you need to rerun the unit tests so that they pass.

Once you have completed this step, you should commit your project to your repository (with appropriate commit comments!).

**Task 5 – Modifying a JUnit Test Case.**  To better understand how JUnit works, we will modify one of the test cases and rerun the unit testing to see a failure. Within the CardsTest.java test source file, locate the testCardType() method and change the last *assertEquals* statement from TYPE\_CHANCE to TYPE\_CC. Rerun the JUnit tests to understand how JUnit would report a test failure. This should partly emphasize the idea that, when you get a test case failure, the first step is to decide whether it is the test case code (i.e., a false positive) or the actual source code that is in error. Before proceeding, undo this change and ensure that the JUnit tests again run without error.

Once you have completed this step, you should commit your project to your repository (with appropriate commit comments!).

**Task 6 – Refactoring: Renaming a Class Field.** The next several tasks will run through many of the common solutions to alleviating bad design smells.

Open the Cell.java source file. You will notice that this class is an abstract superclass that has many subclasses. If you do not know (or remember) what a Java abstract class is, see <http://java.sun.com/docs/books/tutorial/java/IandI/abstract.html>. You can view the subclasses of cell by right clicking on the Java file in the Package Explorer window and selecting *Open Type Hierarchy*. Alternatively, you can see the type hierarchy by hitting the F4 key when the Cell.java class is opened. Open the Outline window so that you can also see that the Cell object has a field (i.e., variable) named *owner* that is of type *Player*.

To use Eclipse’s Rename refactor operation, click on the *owner* field in the Outline view, open the Refactor menu and select Rename so that you can change the name of this field to *theOwner*. When doing so, you should check all the option within the menu to change all references, comments, and getter and setter methods. You will also have to go through the Preview to see what will change before clicking on *Finish*.

Do you really trust Eclipse? If you don’t, you can use Search 🡪 Java options in the main menu to look for any field references of *owner* and/or *theOwner* to make sure everything was appropriately updated. Additionally, look in the Cell.java source file at the setTheOwner() method – it was previously setOwner(). You should notice that the parameter is still named *owner*.

You should rerun your JUnit tests to ensure that all previous unit tests are valid after the refactoring and commit this version of the code to your repository, appropriately commented.

***For this task, briefly describe if this omission is an oversight on the part of Eclipse’s refactoring operation and why or why not. Additionally, briefly describe how (or if) this operation is any different than a simple find all and replace.***

I do believe the omission of the owner parameter might have been an oversight because it is important for refactoring for any of the design smells and there is some scenarios that will cause errors. The omission can cause confusions if you attempting to change the owner to setBuyer and also trying to create a new owner field. Both of the resulting getters and setters would have owner as parameter. You would have to physically change it. This operation is different from the simple find all and replace because it does whatever change across the entire project rather than just the class.

**Task 7 – Refactoring: Changing a Class Hierarchy.** Again, within the Cell.java source file, you should find a field named *available*. After selecting this field in the Outline view, choose the Refactor operation *PushDown* to move this field from the Cell superclass to all of its subclasses. Before doing so, be sure to carefully look at the Cell.java code to see what accompanying methods that are related to the *available* field that also need to be pushed down. Once you have decided what else may need to be pushed down, proceed with the *PushDown* refactoring operation for the field *available* and its (if any) associated member functions. When doing so, you may want to use *Preview* to see what would be changed before clicking *Finish*. After carrying out this refactoring, take a look at some of Cell’s subclasses (from the Hierarchy view or simply hitting the F4 key) and convince yourself that the *available* field has indeed been pushed down.

Was this a good idea? The red error icons next to GameMaster.java and Player.java and the problems listed in the Problems tab should tell you that we did something stupid. For example, within the Player.java class, it tries to call Cell’s setAvailable() and isAvailable() methods, but, after refactoring, Cell no longer has these methods!

We could easily undo this action (by either hitting Ctrl+z or using the Edit 🡪 Undo menu) but, instead, you should use Eclipse’s Push Up refactoring operation. To do so, select one of the subclasses of Cell (e.g., CardCell.java) and run the PullUp refactoring option. When doing so, make sure to select also include the associated member methods that you previously pushed down. This would be annoying if we had to do this for all subclasses of Cell, but just forget this for now. But, as you should see, Eclipse recognizes this problem and presents all identically named fields and methods in other subclasses of Cell to also be pulled up. Be sure to select all of them (unselect the Cell box, and then select it again). This should put everything back in order – you should prove to yourself this is the case by rerunning the JUnit test cases. This should also demonstrate how time-saving Eclipse’s Refactoring operations could be on a large project by automating some tasks that would be tedious, time-consuming and error prone if done manually.

You should rerun your JUnit tests to ensure that all previous unit tests are valid after the refactoring and commit this version of the code to your repository, appropriately commented.

***For this task, briefly describe your experience with this task and for which design smells pushing down or pulling up a class’s field(s) and/or associated methods could help make the code more maintainable and why.***

I guess this task help me understand that the type hierarchy view makes identifying which class is a subclass of which class easy. It also helps with understanding the code and editing faster. The task also helps developers to take care of the duplicated code design smell because pushing and pulling fields and methods in the hierarchy to a more suitable place can help with easier understanding. If something doesn’t need to be in the superclass then it shouldn’t.

**Task 8 – Refactoring: Extracting an Interface.** Remaining in the Cell.java source file, you will see a field called *owner* because in the game of Monopoly, players can purchase properties and “own a square”. As an entrepreneurial student, you have come up with a new innovation to Monopoly that you want to market and sell. Your idea is that, in addition to owning squares (i.e., properties), players can own other things, something not in the original Monopoly game. To get your idea out to the market as soon as possible, you decide to simply modify this program and add your innovation. The best way to do so is to add a Java interface. If you don’t know (or remember) what a Java interface is, see <http://java.sun.com/docs/books/tutorial/java/concepts/interface.html>. To make this a new interface, while the Cell.java source file is open, perform the *Extract an Interface* refactoring operation providing the name *IOwnable* for the new interface name. At this point, you will also need to consider which (if any) of the methods in Cell.java should also move into this interface. Carry out this refactoring operation and make sure you understand how (and why) the code changed.

You should rerun your JUnit tests to ensure that all previous unit tests are valid after the refactoring and commit this version of the code to your repository, appropriately commented.

***For this task, briefly describe your experience with this task and for which design smells extracting an interface could help make the code more maintainable and why. In your description, be sure to include a description of which methods you extracted into the interface and what new files were created in this operation.***

This functionality is a really great and easy to create and implement interfaces by extracting like functionality into an interface making the program easier to understand because you are using the same interface in different types. A principle of OOP/OOD. The design smell that is solved is This solves the design smell “Alternative Classes with Different Interfaces”. I extracted getTheOwner and setTheOwner methods into the new interface IOwnable, these methods will allow you to get and set an owner for any object that implements the interface.

**Task 9 – Refactoring: Extracting a Method from Code.** One common problem with novice programmers is appropriately partitioning procedures into separate methods – this leads to large methods that try to accomplish too many things. To solve this bad design smell, we can utilize the useful refactoring operation of taking a chunk of code and turning it into a method. Doing so, we can reuse it elsewhere (if needed). This can also remove the bad design smell of duplicated code.

In the PropertyCell.java source file, you will find a method named getRent(). Within the getRent() method, there is a for loop that aids in determining the rent. Highlight the for loop (and nothing else!) and then select the *Extract Method* refactor operation. You should name the new method *calculateMonopoliesRent***.** Carefully examine the menu so that you fully understand what is going to happen, especially the *Method Signature Preview* portion, but don’t carry out the extraction yet! Rather, cancel this and go back and highlight the for loop as well as the preceding String array declaration. Now, perform the *Extract Method* refactor operation on that code. You should notice that the signature of that new method signature for this proposed new method is different. Decide which you think is best and actually carry out this refactoring. Examine the newly generated code in this file to make sure you understand what happened.

You should rerun your JUnit tests to ensure that all previous unit tests are valid after the refactoring and commit this version of the code to your repository, appropriately commented.

***For this task, briefly describe your experience with this task including the method signature you extracted and why you chose this one.***

I extracted the string array declaration and the for-loop from the method into another method. To make reading through the original method easier and the string array was also included because it also helped with the understanding. You don’t need the string for getting the rent, just calculating the monopolies rent.

**Task 10 – Refactoring: Creating a Local Variable from Repeated Code.** The *Extract Local Variable* refactor operation allows you to take an expression that may be repeated and create a local variable from the expression.

In the GameBoard.java, you should find a method named addCell(). You should notice that within this method the expression cell.getColorGroup() is repeated – this may not be the best way to do it. Rather, we can automatically create a local variable from this repeated code. To do so, highlight one of the uses of cell.getColorGroup() and select *Extract Local Variable* from the refactor menu and perform this refactoring. Investigate the code of this method to make sure you understand what has changed.

You should rerun your JUnit tests to ensure that all previous unit tests are valid after the refactoring and commit this version of the code to your repository, appropriately commented.

***For this task, briefly describe your experience with this task and for which design smells creating a local variable from repeated code could help make the code more maintainable and why. In addition, comment on whether it is always OK to do this to a function call and whether it could affect the correctness of a program.***

I extracted a local variable named colorGroup from cell.getColorGroup so that the variable can be used multiple times locally rather than having to write the entire expression every time. This could be used to fix indecent exposure because you are making sure that not every method know how you call thing multiple times. It is sometime OK change to make as long the variable isn't going to be changed between calls.

**Task 11 – Refactoring: Changing a Method’s Signature.** From reading the article in Task 4, you should recall that changing the signature of a method is something that needs to be carefully considered before performing this refactoring operation. Additionally, this is a refactoring operation in which Eclipse will not be able to make all the needed logical changes – some manual refactoring work will need to be done to complete the refactoring.

In the Cell.java source file, select the abstract playAction() method and perform a *Change* *Method Signature* refactoring operation. The new method signature should have a Boolean return type rather than the original void type and add a parameter, named msg, that is of type String. Before completing this refactoring, use the *Preview* to see where and how things will change. You will notice that, making this change causes a number of compiler errors that indicate the places where you would have to additionally modify the code to complete this refactoring.

You should rerun your JUnit tests to ensure that all previous unit tests are valid after the refactoring and commit this version of the code to your repository, appropriately commented.

***For this task, briefly describe your experience with this task and for which design smells changing a method’s signature could help make the code more maintainable and why. In addition, comment on why things are changing in other class than just Cell.java and how this affected the definitions of any other classes besides Cell.java.***

Changing a methods signature changes how the method is called and used by other classes which causes errors in the code forcing developers to change other methods. There is no easy way for the refactoring to add the appropriated changes in the other methods automatically. Developers have to go one by one to change the calls or values for each specific call. Changing a methods signature can help with the design smell of complex method calls if done correctively.

**Task 12 – Automatically Generating Documentation.** In addition to refactoring code to remove design smells, a common, easy way to improve the quality of code is to add good documentation. For Java programs, the most common form of documentation is to use [Javadoc](http://en.wikipedia.org/wiki/Javadoc). If you *still* aren’t familiar with Javadocs, you should read <http://www.oracle.com/technetwork/articles/java/index-jsp-135444.html> to become familiar with the use of Javadoc in Java programs, how to write Javadocs at <http://www.oracle.com/technetwork/articles/java/index-137868.html>, as well as <http://www.eclipse-blog.org/eclipse-ide/generating-javadoc-in-eclipse-ide.html> to see how to configure and use Javadoc within Eclipse.

With existing code, such as this Monopoly project, writing Javadocs from scratch is not always a viable option - at least not a viable, *quick* option. Instead, we will use the JAutodoc Eclipse plugin to automatically generate pretty good Javadoc documentation. To install the JAutodoc Eclipse plugin, follow the same procedure as you did in Task 1 and search for the *JAutodoc* plugin.

Once installed, you should generate Javadoc comments using JAutodoc by selecting all Java source files in the edu.towson.cis.cosc603.project1.monopoly package, right clicking on the highlighted files, selecting JAutodoc and then Add Javadoc. This should generate the Javadoc annotations in the Java source files. Finally, you should generate the HTML Javadoc documentation by selecting the Project menu in Eclipse and then selecting Generate Javadoc. This will generate the full HTML Javadoc documentation in a doc directory. You can view the documentation from the index.html file in a web browser.

Once the documentation is generated, you should rerun your JUnit tests to ensure that all previous unit tests are valid after the documentation and commit this version of the code to your repository, appropriately commented.

**Task 13 – Detecting Design Smells.** The Eclipse refactoring functionality is very good at automating simple refactoring tasks, but it does little to detect [design smells](https://en.wikipedia.org/wiki/Code_smell). However, there are some tools that help in identifying code/design smells and suggesting potential refactoring solutions. The one we will use in this project is, appropriately named, [JDeodorant](http://users.encs.concordia.ca/~nikolaos/jdeodorant/index.php?option=com_content&view=article&id=45&Itemid=63). To install JDeodorant, search for it on the Eclipse Marketplace (as you did previously for JAutodoc) and install it. To read how to use this plugin, check out <http://users.encs.concordia.ca/~nikolaos/jdeodorant/index.php?option=com_content&view=article&id=45&Itemid=63>. Note that, to actually run one of the design smell analyses, you need to click on the  in the Bad Smell view.

For this task, use JDeodorant to detect any design/code smells in the Monopoly project and make any refactorings you see necessary. You should rerun your JUnit tests to ensure that all previous unit tests are valid after the refactoring and commit this version of the code to your repository, appropriately commented.

***For this task, briefly describe the refactorings you made (if any) as a result of using JDeodorant.***

Using JDeodorant, Design smells such as Duplicated code, God Class, and Feature envy were found in the code so extracting some classes and methods were the solutions that were suggested by JDeodorant while the test cases still passed.

**Task 14 – Design/Code Smells and Refactoring – On your Own.** Finally, utilizing JDeodorant and Eclipse’s refactoring features, analyze your Project 1 solution (the fire danger system) and make any refactorings you deem necessary to alleviate any potential design/code smells.

You should commit this new and (hopefully!) improved version of the code to your Project 1 repository, appropriately commented.

***For this task, briefly describe the refactorings you made (if any) as a result of using JDeodorant.***

Using JDeodorant for project 1, I found no design smells.

**Task 15 – Summing it All Up.** Upon completion, each student is to submit a short report that includes your write-ups from the previous tasks (clearly labeled) and the following:

* A description (2-3 paragraphs) of what you learned from this project and how it relates to some of the topics covered in lecture

After completing these exercises, I learned that refactoring is a good way to resolve design smell. I learned that when tasked with the maintenance or job of making the code look better, IDE’ provide great tools to help enhance the code during the refactoring process. I also learned that my silly thought that refactoring was just renaming variables and moving classes is false. It is much more than that. The ability being associated with eclipse saves a lot of time in fixing code and improving code.

We have talked about maintenance in class and this exercise shows that when tasked with maintaining existing code, refactoring allows you update the code to increase readability without changing the internal functionality. Utilizing unit tests while refactoring allows developers to build and maintain flexible reusable code in their projects while not messing anything up or creating errors.

* A description (2-3 paragraphs) of what you liked about Eclipse’s support for refactoring including its strengths and limitations as well as your impression of JDeodorant

I am not sure which other IDE’s support refactoring but Eclipse having the functionality of refactoring makes it an extremely powerful tools in addition to all its other offerings. Its strengths are the functions of moving parameters, methods, classes, and making interfaces. The refactoring tool can move code more efficiently then a developer can because we cannot see how a large project’s classes interact with each other.

Its limitations also exists. Refactoring can present new problems in code if not properly tested such as the one we saw in task 11 making you have to manually update code in certain situations in where eclipse cannot see the necessary changes. JDeodorant

* A description (1-2 paragraphs) of how/why unit tests are important when doing refactoring

Unit tests or some type of testing are extremely important in refactoring because it allows a quick and easy way to validate and ensure the changes you made did not affect or break the original functionality of the refactored code. If you don’t make sure that your program still works as before then you never know whether or not you broke the part of the code of functionality that you changed. That is why unit tests are important.

**Grading:** Again, the goal of this project is for you to gain experience in software quality assurance, software documentation, configuration management and utilizing real world software development tools – this is NOT just a programming assignment!

Each project will be graded as follows: 60% allocated for your source code development/refactoring tasks; 20% for repository commit comments/quality/frequency; 10% for your task write-ups; and, 10% for your generated Javadocs.

**Submission:** You must submit your Eclipse project through your GitHub Git repository. You should develop this code within Eclipse and ***frequently check your code*** into your GitHub project site using the GitHub Desktop client. That is, I should see several checked in versions of your code and documentation when I browse your project after the deadline.