Obed Dominguez

AIT 642 / COSC 603

Software Testing and Maintenance

Spring 2016

Project 2 Task Write Ups

**Task 6 – Refactoring: Renaming a Class Field**

***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 don’t think that this omission is an oversight from Eclipse’s refactoring operation. As the document “Refactoring for Everyone” mentions, Eclipse “understands the code semantically” (ibm.com) identifying references to the method, variable, or class names. I believe that this operation is different than a find all and replace function, because as mentioned earlier, eclipse understands the code. Also, the “Refactoring for Everyone” paper recommends that when using the replace functionality it needs to be done with care, because different classes may have methods and variables that are named the same, and that it is also a tedious process to go through all the files to make sure all instances are correctly identified and changed.

**Task 7 – Refactoring: Changing a Class Hierarchy**

***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.***

Following the instructions for this task and reading the refactoring paper helped me have a good experience doing this task. I was able to see and understand what happened with the code and could see how it can potentially create problems if not done correctly. This step can potentially make code more maintainable if you want to create subclasses by making sure the necessary fields are taken and the appropriate setters and getters are put in place in order to make the change. If you take the correct things you need for the changes when refactoring the code, it will work fine and be easier to maintain.

**Task 8 – Refactoring: Extracting an Interface.**

***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 task took me some time to really understand how the interface would work once the changes were made. If I was adding a new class that followed the same function or behavior as the Cell.java, why create it. Once I realized that the class could act on its own, borrowing the methods from another class, it made sense. Extracting an interface could help make code more maintainable because you can create an interface that will work better using other methods and make similar functions. For this task I took the following methods from Cell.java: getName, gettheOwner, getPrice, playAction, setTheOwner, toString, isAvailable, and setAvailable methods. The new file that was created was the IOWnable.

**Task 9 – Refactoring: Extracting a Method from Code**

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

This task was easy to follow, but it took time to make sure I understood the changes that were going to happen. I chose to include the string array declaration for this refactoring because it made more sense to include it in the new method. I chose the string because it made the get rent method shorter, and the new method had the string array declaration and the for loop.

**Task 10 – Refactoring: Creating a Local Variable from Repeated Code**

***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.***

This was another task that following the steps was easy to do, but it took time to understand what would happen after changes were made. Creating a local variable could help make code more maintainable because expressions that are repeated can benefit from a local variable. Most times it can be ok to do this function, but you have to be careful if the method returns different values when it’s called.

**Task 11 – Refactoring: Changing a Method’s Signature**

***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.***

This task was difficult to perform mainly because of having to manually make changes to every error that was created after changing the method signature. For this particular task there were not that many errors to fix, so it was fine to change them each individually. If there were more errors, this could be annoying to do. I’m sure there is a way to do it easier, but I just don’t know and couldn’t figure out. Changing the method signature could help make code more maintainable by changing what the method will return. Things changed in other classes because for every instance when the Cell.java called another class, the return type was affected. The definitions changed to reflect the changes that happened in each class.

**Task 12 – Automatically Generating Documentation**

No write up requested for this section.

**Task 13 – Detecting Design Smells**

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

For this task I made a couple of changes to the Monopoly code after running the JDeodorant plugin tool. Some of the changes I made were part of the “Feature Envy” result that suggested to move some methods. I followed most of the refactoring suggestions but ignored some suggestions if it created errors in the code. Every time I made a change, I ran the JUnit test to make sure everything was still working, and any time there was an error, I would undo my changes. I also made changes in the “Type Checking,” “Long Method,” and “God Class” following the same method mentioned that if errors occurred, I would undo my changes and make changes if the JUnit test passed.

**Task 14 – Design/Code Smells and Refactoring – On your Own**

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

For this task I did not make any changes as a result of using JDeodorant because the tool did not yield any refactoring suggestions. I believe that JDeodorant did not yield any factoring suggestion because of the way my code was built. However, I did make some changes to my code and tried to do some refactoring as well. Most of the changes that I made were fixing warning errors in my code. Some of those changes were creating local variables and add setters and getters. I tried to extract some methods and make them private in an attempt to clean my code. Additionally, I tried to create a Junit Test but was not sure how to do that. I believe that had I had a good set of Junit tests, I could have found more errors.

**Task 15 – Summing it All Up**

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

For this project I got to have a more in depth understanding of how refactoring works. Each task exposed me to the functionalities the refactoring tool offers. Learning the functions refactoring offers made me think about how to approach my coding style. Based on my first project, I could see that my code had bad code smell, but with refactoring I can begin to fix that.

This project directly relates to the topics covered in lecture because it gave me hands on experience with refactoring. It’s one thing to listen to a lecture and try make sense of what is being said, and it’s another thing to put what was said into practice. The project covered how refactoring works with JDeodorant, and how JUnit tests help keep the code working correctly while you refactor.

* 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

The support that Eclipse offers while refactoring is good to certain point. It has a lot of strengths that can help you make changes to code and make it more maintainable. My impression of the limitation refactoring has was apparent to me when changing the method signature, because you have to make additional changes if errors occur.

When I got to use JDeodorant I felt that all of the previous refactoring tasks could have been done using JDeodorant. I assumed that the tasks were to manually expose us to the process of refactoring. Then, my impression of JDeodorant was that it’s a tool that can help you refactor a lot faster than just choosing certain code to refactor.

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

Unit tests are important when doing refactoring because it helps you make sure that the changes you are making are not affecting the overall function of the program. With the use of JDeodorant you can quickly find things to refactor and make those changes. Then you can run the JUnit test to make sure that the changes you made did not affect anything. If the refactoring function did affect the code, you can find out when you run the JUnit test.