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

Eclipse omitted an instance where the variable *owner* was supposed to be changed to *theOwner* is probably an oversight on Eclipse’s part. I can see if it would miss an instance where the variable is in another class, but in this case the variable is in a setter method from the same class.

Additionally, this operation is no difference than a simple find and replace. It is definitely convenient and a nice feature to have when you have a refactoring undertaking such as Monopoly that involves many instances in as many files.

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

For this task, we experienced using the Refactor->Push Down and Refactor->Pull Up functions to push a variable down from a superclass to all its subclasses. The accompanying methods to the variable in question were also pushed down to the subclasses. However, this created a problem with other classes where the methods that contained the variable in question were called and they were no longer available due to the result of the Refactor->Push Down function being executed. We had to undo this action by executing the Refactor-> Pull Up functionality. This task has to be executed with care to ensure that all methods and named fields associated with the subclasses be pulled up. JUnit test was run to ensure all tests executed successfully.

The Push Down and Pull Up functionalities of the Refactoring tools in Eclipse are helpful in making the code more maintainable and the tool certainly helps save time if you have to do it manually. However, the take-away for this task is the coder still had to do some due diligence, i.e., manually check to ensure that the superclass and the subclasses that were affected by this task are still functioning as they were intended and that this task did not create some unintended results such as what was observed in the previous paragraph.

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

This task had to do with extracting an interface by using the Cell.java source file and creating an interface named *IOwnable* to represent an interface of “owner” that also contains the getter and setter methods. This was easily accomplished by performing Refactor->Extract Interface… function from the Refactor menu. The new file is called IOwnable.java and the methods that were extracted were getTheOwner() and setTheOwner(Player theOwner). The Cell class now extends *IOwnable* interface since *owner* was its original variable that was extracted out.

By creating the interface *IOwnable* makes the code easy to maintain. For example, when a class is extending this interface, it can override the interface’s getter and setter methods, so the method signatures are not altered, just the implementation of the these methods inside the classes that are calling the interface’s methods.

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

The goal of this task is to extract a method out of a chunk of code to make it more usable somewhere else and also making the code easily maintainable. In this task, the *for* loop in the *getRent()* method from the PropertyCell.java source file used to calculate the rent was extracted out to create the *calculateMonopoliesRent(rentToCharge)* method. Looking at this method, it is easier to calculate the monopolies rent based on the *rentToCharge* input variable. Also, the signature of this method is changed from the original *getRent()* method which is public and it is now a private method which can only be accessible inside the *getRent()* method (if that is the intention for this method). This way the *getRent()* method is calling the *calculateMonopoliesRent(rentToCharge)* and the actual calculation is carried out inside this method, so there’s a level encapsulation that exists. The *getRent()* method is a public method which can be accessed anywhere from the PropertyCell class, therefore the implementation to calculate the rent should not be present in this method.

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

The goal of this task was to create a local variable out of repeated code within a method. In this case, the *addCell()* method from the GameBoard.java file uses the expression *cell.getColorGroup()* is repeated. A local variable named *colorGroup* was created and its value is set by calling the *getColorGroup()* method of the PropertyCell class. This change made the code more readable and maintainable. The *getColorGroup()* method is only called once when the *colorGroup* variable is set. After that, *colorGroup* can be referenced elsewhere in the code instead of calling the *getColorGroup()* method repeatedly. It is fine to carry out this refactoring task as long as the JUnit Test still ran correctly, which it did. To answer the question if it is okay to do this all the time, it depends…as stated above.

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

The goal of this task was to change a method’s signature to make it more visible by changing the return type and adding an extra parameter. The *playAction()* method of the Cell.java source file was changed from the return type *void* to Boolean and adding a String variable named *msg*. The Preview option was used to see what other changes will take place because of this change. The affected files that used the original *playAction()* method were also updated to prevent compilation errors. This is by far the most difficult and tricky of the Refactoring tools to use since it seems that more work is done to change just one method’s signature! However, it is still a powerful and time-saving tool to use with careful planning and anticipation of the outcome. JUnit Test was run successfully after all changes were made to the affected files.

**Task 13 – Detecting Design Smells**

JDeodorant was easy to use. The real problem was sometimes the errors were introduced after the refactoring. It is quite time-consuming to do one class or method at a time. Perhaps I’m doing it all wrong, I’m not sure! The first time I did it, I selected all java files in both packages. The gui files created a lot of problems that I ended up spending way too much time fixing the errors after all the refactorings. I decided to do away with the gui files since most of them have to do with clicking buttons and actions performed after some buttons are clicked. It would be too messy to refactor those. I did a number of java files under the monopoly package and was able to find suggestions for refactoring in all areas of Bad Smells using JDeodorant. I did refactor all files suggested for God Class, Long Method, Type Checking, and Feature Envy. Unfortunately I ran into a problem with Eclipse and GitHub and lost the whole Monopoly project that contained the refactored files. I will have to redo this task, if it’s ok with you.

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

The goal for this task is to use JDeodorant to detect design and code smells of the Fire Danger program. The only Bad Smells that JDeodorant detected was Long Method smell. The *calculateFineFuelMoisture()* method was thought to be too long and it was suggested to be broken/extracted into two smaller methods to calculate the coefficients A and B and later used in the formula to calculate the difference between the wet and dry bulb temperatures. I applied the refactoring suggested and reran the program and the results showed to be the same between the original code and the refactored code.

**Task 15 – Summing It All Up**

I learned a lot from doing the Monopoly project. I personally never used JDeodorant before, so it is a learning curve for me. There must be an easier way to do code smells. I did not have any issues using the tool, however, the tedious task of refactoring one class or method at a time was very time-consuming and then to find out, there were errors introduced by performing some of these refactoring tasks. I needed to undo them to get the code back to the no-error version. Like I mentioned earlier, perhaps I’m doing it all wrong! I followed your instructions for all Task 1 through Task 13 without any issues. JDeodorant really tripped me up!

I also find that GitHub desktop used in conjunction with Eclipse seemed to be problematic. Again, I did not have previous experience with GitHub, so it is a learning process as well for me. With Project 1, I was able to use it without any issues. However, with Project 2 it was giving me a lot of issues. I couldn’t get GitHub to sync up with my local repository, therefore I spent way too much time redoing this simple task before I was able to configure the GitHub desktop to work and was able to do all the commits to GitHub. Thank God for that! ☺ Conferring with some class mates, there seemed to be different issues, depending on what method you use. Some people did not even use the remote syncing at all, I’m not sure how that works, but I guess they got it to work!

Eclipse is powerful IDE in that it provides the tools to help coder/programmer to use their time more constructively and it saves a lot of time in refactoring code. However, it does have its limitations. If you leave it to Eclipse to do everything on its own and not manually checking/making sure that it’s doing what you intended for it to do, you could end up redoing or undoing what it did. Overall, it is still a good tool to use. Today there are so many tools out in the market, especially when it is free, it is unimaginable that anyone is still doing anything manually any more. Learning how to use these tools is another story. They’re all relatively easy to use, just takes a little getting used to. I am a little out of touch of doing some hands-on work, so it probably took me longer than someone who spends more time doing this or doing it on a regular basis.

I find that JUnit Test included in this program was very helpful in ensuring that all tests still run successfully after refactoring was applied to the code. I ran JUnit Test every time I modified/refactored some code. It is a reassurance that you didn’t break anything. Unit tests are important after making changes to the code. This is evident in real life when people actually spend time developing test cases for regression testing. Test cases were written to cover certain areas of the code and when code changes were made, they are sometimes being run over and over again to ensure that no errors were introduced to the baseline before the code changes were applied.

This project was very educational in putting some of the concepts learned in class to practical experience. Sometimes if you just learn things in theory and never got to practice you tend to forget. This reinforces the ideas and concepts introduced. Overall I really liked it. I have used Eclipse in the past, but did not get to use the tools introduced in this project. If I have more time to spend using it and getting used to it, I probably would like it better! ☺