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

With the parameter still being owner and not theOwner in the setTheOwner() method is not an oversite on Eclipise because that variable had no relationship to the original class member “owner”

This operation is severely different than doing a simple find and replace. First, the preview showed me the old and new changes side by side. Secondly, this refactor tool seem to have gone through the rest of the connected files and looked for any calls to setOwner or getOwner and changed them to setTheOwner or getTheOwner. If I did a simple find and replace, I may have forgotten change a file and the program would not have compiled. This seemed to be a much faster tool than doing a find and replace.

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

This refactor tool was pretty straight forward. When I pushed down the *available* variable and methods any class that was a subclass to Cell class received *available* variable and methods. This refactoring tool could fix a few smells like “class is too Large” or “Duplicated code” if the class is too large I can always push something down and help make the class smaller. If all the subclasses have the exact same function and code within that function I should push it up to prevent duplicate code.

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

I did not know what an interface was, and after some research I understand that an interface are functions that all classes that are associated with it must have. The design smell that *extracting an interface* would help with is duplicating functions and maybe *shotgun surgery*. We created the *IOwnable* interface, so that means anything that I attach to it needs to have the setTheOwner and getTheOwner function. In this case only *Cell* class needs this function but if I were to add a *car* class, I need to create a setTheOwner and getTheOwner methods. This refactoring tool created the *IOwnable.java* file.

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

This was a pretty straight and forward task, if was fairly easy to understand what the refactoring tool was doing. With one option more parameters were being sent compared to the other option. I picked the one with the less parameters being sent because this should mean less items on the stack that needs to be passed around and helps fight the smell of *too many parameters.*

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

My experience with this task was that creating a local variable for many functions calls is very simple. This is removing the broken modularization design smell. This removed two calls of *getColorGroup().* A problem with this refactoring is, if this variable gets changed in the class, the method that set a variable is now out of date.

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

This task was a little more complicated because everywhere the void function *playAction()* was located it change it to wanting a return value and none of them had a return Boolean, causing complier errors. The reason why eclipse wanted the programmer to fix these error their selves is the refactoring tool does not know how to obtain the value to return, making the programmer to check them over. The reason why other classes need a change because they are extended by the *Cell* class. This can help maintain code because it could make a method more useful in the furture.

1. ***Task 13 – Detecting Design Smells.***

I used the *feature envy* and *God Class* tools to select a few spots to refactor the code. Feature Envy will tell me if there is a method that is being called more elsewhere. Feature Envy will suggest to use the *move method* refactoring technique to be used to solve these problems. God Class will suggest to use the *exact class* refactoring method if there are more connections in the extracting class compare to the source class. In game master class, Player was being extracted a lot. This makes a good refactoring choice because player was being use elsewhere a good amount of time.

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

Using JDeodorant on my project 1 code, there were no God Class or feature envy smell code found. It did find a good number of long methods that I decline to accept because I thought the move would break up the code too much and make it harder to follow. I had if / else statements for the wind speed, it wanted to break up that up in to two functions with the same if / else statement. To me this seem like it was duplicating code, which would lead to smell code.

1. ***Task 15***

**What I learned from this Project!**

I learned a great deal from this project, which includes refactoring, smelly code but not much about spaghetti code. With what I learned from the refactoring tools in Eclipse and from the lecture, I can defer which of these refactoring tools helps for which smelly code. Although this project did not teach us the rule of three.

The first refactor technique we used was the rename, this helps with the *comment* bad smell because it can change all the bad variable names in to a good one in one fell swoop. The changing a class hierarchy can help fix feature envy because if a method is calling something too high up or down it can be moved with ease. The last thing I am going to mention that I learned was that extracting interface seems to be a way to fix inappropriate intimacy smells.

**The Good the Bad the Ugly of Eclipse**

Eclipse has some great refactoring tools, from moving methods, rename variables to plugins like JDorderant. There are a couple strengths that were helpful, however there were some weakness. A strength was when I changed one item in on class, eclipse would change it in all other attached classes. Another strength was preview of the refactoring. Eclipse had some sort falls, the changing a method signature did break some portions of the code, however I think this was more for the programmer’s benefit. There are more pros than cons with the refactoring tools in Eclipse and will be very useful in the future.

A strength that I mention was eclipse changed code in other classes. For example, when I push a method down from a higher class, that method was now written in all sub classes. This refactoring tool saved me from having to copy and paste code that I created. This is also useful because if I created a bug in the original code, I just copied it to another part of my program. Another Strength was the preview option. This helped me understand what I was about to change by refactoring. Although this had a short fall, it only showed me the file I started with and not the connected files changes. The signature changing did break a lot of my code but for good reason. It was telling me that I needed to go to all the places that the function was being called and change how I was handling the returns. If it just put a return at the end of the function for me, I may have forgot to look at a file and fix them.

JDorderant has some interesting feature and useful aspects. The first thing was it did search the code for anything that could be smelly. This would not work for refactoring on the “Heroic scale” but it could help preventing code to get to that point. The Visual pane in eclipse was helpful to see if an extract class was using a method more than the source class, which indicates a good time to refactor. Some of the tools in JDeorderant will take some time to get use to like the duplication tool. I know in my project 1 code there are some duplicated functions that I can make in to one function but I do not know how to make JDorderant to find them.

**To Test Cases or not to Test Case!**

Test cases are a great and wondering thing to have when refactoring. Luckily for me, probably by design of the project, when I refactor the code there was not any unexpected changes to the code that would cause any of the test cases the fail. This could either be because eclipse did a great job refactoring or the test cases did not test everything. I am going to pick the former and my professor created wonderful test cases.

Having test cases are important while refactoring because refactoring can significantly change the code. For example moving a method up the hierarchy or changing a function’s signature. When changing a signature, it potentially changes a variable you do not mean too, causing a test to fail and when a test fail the program is not working as intended.