James Rehak 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

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

This omission is not an oversight. Parameters are not affected by refactoring a class field because names parameters are unique to that method. This property allows a programmer to use the same name for a parameter in many different methods without any complier errors.

Refactoring is more than a simple find and replace all. Refactoring looks at the logic of the program to find where the renamed field is. A simple find and replace all reads through the entire text of the program and replaces the matching instances. If refactoring operated complete like that, then it would have renamed the same named parameters.

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

At first I was nervous making a big change, but using the refactoring tools makes backtracking much easier and preview certainly helps a lot. The initial push down operation caused errors because it pushed abstract methods that other classes use out of the abstract class.

I found the PushDown and PullUp refactoring operations very useful for correcting abstraction mistakes quickly, assuming the misinterpretation of the design is not too off. These refactoring operations can correct bad smells, such as God Classes, Divergent Changes, and Speculative Generality by moving class methods and/ or fields from its sub or superclass. This makes code more maintainable because the next person working on the code will be able to recognize the Hierarchy structure easier.

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

An interface contains general methods that all classes who implement it use. I created the interface IOwnable with the “theOwner” field and methods: setTheOwner(), getTheOwner(), getPrice(), isAvailable(), and getName() because all of those methods set and get aspects of an own-able object. The refactoring created an IOwnable.java file which is the interface. In addition to that, all instances where “Cell” was called to check an ownable aspect, “IOwnable” was called instead. This caused name replacements in 23 classes. The monopoly game passed all JUnit tests after refactoring as well as ran.

Extracting an Interface would be useful correcting design smells such as: God class, duplicate code, and Inappropriate Intimacy, however it could cause Speculative generality if the new ownable objects are never created. This refactoring makes the code more maintainable by allowing it to accept new ownable objects easier.

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

For this task, I decided to refactor just the for loop in the getRent() method (option 1). I think the first choice is better because it separates the actions of the methods more clearly. The getRent() method gets things and thought leaving the getMonoplies() method in fits the action of getting more than calculating. This leaves the calculateMonopoliesRent() method, just to calculate what is given to it.

Extracting a method can fix: Inappropriate Intimacy, Feature Envy, Too Many Parameters, and Method too long. This makes code more maintainable, because the methods do less things, which makes them easier to understand.

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

I used the Extract Local variable operation to create the String colorGroup from the cell.getColorGroup() method call with in the addCell() method. This decreases complexity by not having to call cell.getColorGroup() several times every time addCell() is called. This makes code more maintainable by allowing people to trace issues easier, since the method is called once and assigned to a method variable. This can fix smells such as: Duplicate Code, Method too long, and Shotgun Surgery.

Creating a local variable from repeated code is not always acceptable though. If the method call isn’t suppose to return the same value, for instance a Scanner call like input.next(), then using a local variable would cause only the first input String to be used.

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

Cell is an abstract class, which means other classes use its methods as their general methods. Modifying the signature caused errors in all classes that extends Cell, ex: PropertyCell, CardCell, etc… because they use the playAction() method because Cell contains all of the common methods each type of Cell uses. I went though and manually corrected all errors. The eclipse ide recommended a return false, so I made all instances return a false value, though it didn’t matter what Boolean value I picked. The game was not written to check any Boolean values from the playAction() method, so the value of the retuned Boolean should not affect the game.

Changing a Method’s Signature can deodorize the smell of too many parameters. Lots of parameters cause complexity, which makes code harder to understand and more likely to break when something is changed (poor cohesion).

**Task 13 – Detecting Design Smells.**

I first ran Feature envy check. There were five methods that were detected to be moved. From the GameMaster class I moved completeTrade() to TradeDeal Class as well as btnDrawCardClicked(), sendToJail() and movePlayer() to the GameBoard class because each of those Classes utilized those methods more than the GameMaster class did.

Jdeodorant suggested I move the purchase() method from the Player class to the Cell abstract class, this produced errors and was not a good idea because the Player class did not implement the Cell abstract.

Jdeodorant did not find any long methods to extract. The type checking bad smell checker revealed three issues. The issues were caused by large conditional statements within methods. The refactoring used polymorphism to replace the conditionals with a single line of code. This generated additional classes. There were some errors in the automatic code generation. In one of the new methods generated, a switch state was used, which introduced “a case expressions must be constant expressions” error. I fixed it by changing the switch to and if else.

The god class bad smell detector found 10 god classes that could be reduced by extracting smaller classes out of them. Unsurprisingly, half of the flagged classes were the unit tests to the flagged class. Most of the stench game from the GameBoard class, which has a lot of responsibilities. I extracted the Rent concept out of Property Cell, calling the new class PropertyRent. As well as the Roll Dice concept out of the Game Master.

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

Jdeodorant did not find any bad smells with my fire index calculator program! I could not find personally find any design smells, so I did not refactor anything.

**Task 15 – Summing it All Up.**

This was my first dive into “real” use of the refactoring tools. Before this project, I had only ever used it to rename variables, because I thought of better ones later. This project helped me learn to identify refactoring opportunities. I can see how refactoring after completing a programming implementation would be beneficial. Code can become not as cohesive as it could be the closer one is to finishing an implementation. Refactoring cleans that mess up.

Refactoring without tools can be daunting, changing a working program can introduce errors as well as be time consuming. The old adage, “If it ain’t broke don’t fix it” rings in the back of the mind. Fortunately the refactoring tools included in Eclipse take a lot of the pain out of refactoring.

I found Eclipses support tools to be very empowering. I particularly liked the preview function before refactored, it really helped identify what was going to be affected before committing to the change. I can see how relying on tools refactoring to make a design change (for instance an abstraction change) is much more efficient and safer than trying to manually adjust the files. As helpful as Eclipse’s refactoring tools are it is essential to understand the change that will being committed. Blindly committing to a refactor can introduce errors to the programs correctness. This is especially true when using JDeodorant. The use of Unit tests are extremely necessary for confirming the program is still correct.

JDeodorant has lots of potential. It can help make up for novice design mistakes, assuming the design is not too wrong. I found the automatic class naming of extracted Classes from a God Class to be quite lack luster. The extracted classes would be called whatever the source class was plus “product”. Fortunately, the names can be edited before refactoring. As helpful as JDeodorant is, not all of the suggested fixes were effective. Some of them introduced complier errors, while others could only be detected with unit tests.

Unit tests are extremely important when doing refactoring, because they allow us to see if we broke any thing after refactoring without having to run through the whole program. It is much more efficient and effective because the tests reveal exactly where any incorrectness in the program is. Without unit test, one would have to run the whole program, making sure every input and output is correct, a very daunting and time consuming task.