COSC 603

Software Maintenance & Testing

Spring 2016

Project #2 – Refactoring and Design Smells

Task 6 - ***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.***

The omission by was not an oversight by Eclipse’s refactoring operation. This omission is not an oversight. The rename refactoring operation performed applies to the protected variable "theOwner" of type Player for the Cell class. The setTheOwner method takes a parameter of type Player. In this case, the parameterized variable is named "owner". This "owner" parameter's scope only applies to the contents of the setTheOwner method, therefore its name is really irrelevant. It could be named "apple" (which isn't advised because it would make the code a little confusing) and the setTheOwner method would still perform the same, setting the instance of the class variable "theOwner" to that of the parameter whether it's named "owner" or "apple" or “theOwner."

It's pretty clear from this example that the rename operation is not just a simple find all and replace. If it were, it surely would have renamed the parameter as well. Also if it were a simple find/replace in the Cell class, it wouldn't have updated the method calls in other classes. This refactoring operation also made changes to PropertyCell.java and RailRoadCell.java where they also have references to owner previously.

Task 7 - ***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.***

It was interesting to learn how to use the push down and pull up methods to refactor code. This is a pretty neat feature, and I will probably use this in the future now that I know what it does. Pushing down and pulling up could be useful for resolving divergent changes. For example, if you have a class with many subclasses like Cell has many subclasses, pushing down could be used to movie fields in Cell to a more applicable subclass. This ensures that all methods related to the field that is being moved are moved as well. It makes sure that changing the location of a field within the code also means you're changing the related methods and method calls as necessary to retain existing functionality. The type of code smell that the push down refactoring operation is used on are as refused bequest code smells. This is because the push down refactoring helps prevent your from inheriting code that you do not want. The type of code smell that the pull up refactoring operation is used on is the duplicate code code smells. This is because the pull up refactoring helps to elevate code duplication.

Task 8 - ***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.***

Learning how to refactor by “extracting an interface” was rather interesting. I had never done this before. The task itself was rather easy but it did teach me something completely new. This was a really quick and easy way to create and implement interfaces. I elected to move the methods getTheOwner and setTheOwner. The results of this change were a new class IOwnable was created. Cell now implements IOwnable. GetTheOwner and setTheOwner are abstract methods in IOwnable and the specifics of the implementation is still in Cell. This could be used so fix Primitive Obsession. In a way this forces constraints. You're abstracting out something about this Cell. Cell has an owner and so, cell must have a way to get and set the owner. By implementing IOwnable we are saying these are things that must be defined for each class.

The methods that were extracted into this interface were getName, getOwner, getPrice, playAction, setAvailable, setOwner and toString. After looking through the code, I didn’t notice any other files created other than IOwnable.java.

Task 9 - ***For this task, briefly describe your experience with this task including the method signature you extracted and why you chose this one.***

This was a neat trick help from having a method that got too large. This was very new to me as well. I liked seeing how the method signature was different in each case. I choose to include the string array declaration with the for loop when extracting the method.The string monopolies was not used anywhere else in the getRent method so it made sense to only pass the int rentToCharge. Passing the string too would be unnecessary. Again a really cool refactor feature in Eclipse.

Task 10 - ***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.***

Like all other tasks this was straight forward and was a great learning experience for me. It helped me see why someone would use this refactoring operation. The type of code smell this would be used for is duplicate code. This is because this refactoring operation in this case helps from having to call the getColorGroup() multiple times in the same method. In general this is an OK change to make. As long as the of the variable isn't going to change (and you need the updated value) between the first and second time it's called. I believe that does not affect the correctness of the a program in a negative way.

Task 11 - ***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.***

I found this the most dangerous action that was have performed so far in this project. You have to be very careful with this type of refactoring. Changing a methods signature changes how the method is interacted with. This forces other methods to change how it calls the method and refactoring can not intelligently add an appropriate variable for all scenarios through out your project. You must go back by hand and add in the appropriate value for each specific call. Changing a methods signature can help with the design smell of complex method calls, and could also fix the data clumping design smell. Changes occurred in more than just Cell because the specifics of the implementation of playAction are in the subclasses. Additionally the each playAction method in every subclass of Cell was altered to pass the msg parameter.

Task 13 - ***For this task, briefly describe the refactorings you made (if any) as a result of using JDeodorant***

I made a few different refactoring when I used JDeodorant. JDeodorant found a whole bunch of code smells that warranted the use of refactoring. I made some move method refactoring within the classes Tadedeal, and Gameboard. In these classes I created a new methods in the classes that uses the method the most, then move code from the old method to there. Turned the code of the original method into a reference to the new method in the other class or else remove it entirely. I used the extract a class within the classes PropertyCell, GameBoard, and Player. In these instance I created a new class and place the fields and methods responsible for the relevant functionality in it. I also used the extract a method in the GameBoard class. In this instance I moved code to a separate new method (or function) and replace the old code with a call to the method.

Task 15

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

I learned quite a bit more from doing this project then I did from project one. I had really never done much refactoring other then, the occasionally renaming of files or reorganizing a with in a project’s hierarchy. I really found learning about how to use refactoring in a practical setting instead of just hearing about its theoretical uses more useful. This was a really good exercise in how to quickly perform refactoring procedures that are commonly used to resolve "bad smells." Mostly I learned that these operations existed and that "refactor" wasn't just for renaming and moving classes. This could save a lot of time in fixing code. It makes improving the code, what we want, faster so, the development team can afford to spend time on enhancements like the users want.

I really like Eclipse's refactoring. Up until this project I had only really used it to move or rename classes. These additional functions of moving parameters, methods, classes are all very useful and I'm positive that I'll use them in the future, now that I know they exist and how to use them.

This project related to all the lectures we had on the topic of code smells and refactoring. In this project we got see practical uses of for detecting code smells and learned how to fix then using refactoring.

**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.**

While there are some limitations, and places where the developer still needs to specify parameters, methods, fields to be affected, I think these are necessary interventions. The code can't write itself, if it could we would be out of a job. It's important that even though these features or shortcuts exist we have say in what is affected and how the code changes.

I really like JDeodorant as it is a very useful tool for a developer. It allows the user to have a tool that helps to easily find were most of the bad code smells are in the application. Once those code smells are identified, it easily helps the user to make the appropriate code refactoring changes.

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

I had never really done any unit tests, utilizing a unit test tool like JUnit. I can definitely see the benefit as to why unit tests are so important when refactoring parts of the code. Refactoring is cleaning up a piece of code (e.g. improving the style, design, or algorithms), without changing (externally visible) behavior. The developer write unit tests not to make sure that the code before and after refactoring is the same, instead he/she write tests as an indicator that his/her application before and after refactoring behaves the same: The new code is compatible, and no new bugs were introduced.

However, unit tests are also useful to locate errors, so it can make sense to write unit tests for private parts of the software as well. These tests are expected to change throughout refactoring. If a developer wanted want to change an implementation detail (like the naming of a private function), he/she first updates the tests to reflect their changed expectations, then make sure that the test fails (your expectations are not fulfilled), then he/she change the actual code and check 4that all tests pass again. At no point should the tests for the public interface start to fail.