Jlaja Pandora Podier

Robert Bathmann

COSC 603: Software Testing and Maintenance

Spring 2016

Project 2 Refactoring and Design Smells

Task 6

No, renaming the parameter is not an omission on the part of Eclipse. The name of the parameter for the setter method is not related whatsoever to the field variable, except in similarity in name.

The rename command is much different than a find and replace. Eclipse finds all usages of the identifier (field, parameter, method, etc.) and updates each usage with the new name. A find and replace would have also renamed other identifiers that may also have been called owner.

Task 7

Eclipse was able to successfully push down and pull up the available field and its getter and setter. Previewing the change helped identify if the correct actions were being taken.

This type of refactoring is effective at remediating the code duplication, large class, and downcasting design smells. Pulling up the repeated available field and methods was able to reduce duplicated code, which is troublesome for software maintenance because a change made to one code section may not be made in the duplicates. Pushing down may reduce the size of a super class, but should only be performed when at least one subclass does not require the member that is being pushed down. Finally, we saw that pushing up a member can solve the need to downcast when all subclasses can share the same member. This is what caused the compilation errors after pushing down the available field.

Task 8

The getTheOwner and setTheOwner methods were extracted to the IOwnable interface (IOwnable.java). Extracting an interface can solve the downcasting code smell since ownable items no longer need to inherent from the Cell class. This could make the code more maintainable since new ownable items can be more easily added to the game. A new item class would simply need to implement the IOwnable interface. However, the classes that used the owner setter and getter were updated to use the IOwnable interface instead of Cell where possible. This assumes no future use of Cell's other methods or fields will be used in those areas of the code.

Task 9

I chose to include the String array declaration as part of method to extract. Since the owner is a field, the new method would have access to the owner and be able to get an array of the monopolies itself. The calling method no longer needs to care about this detail. The parameter name rentToCharge was a bit misleading since that is what we are attempting to calculate. Therefore, we changed the name to originalRent and adjusted the new method accordingly.

Task 10

Extracting a local variable can make the code more maintainable by reducing the amount of duplicate code. While this refactoring only eliminated one call to a getter, extracting a local variable is often more useful if the value is unnecessarily calculated multiple times which can increase the chance of errors in the code.

This refactoring is not always a good idea and may impact the correctness of a program, especially for reference types. Consider if between usages of the new local variable, the setter for colorGroup in the cell class was invoked, thereby changing the color group. The local variable would still reference the old value of the color group and may lead to an error.

Task 11

Changing a method signature could eliminate feature envy if the method makes too many inappropriate calls into other classes that would better be handled as parameters. It could also reduce the number of parameters if not all are deemed necessary. Additionally, it could eliminate the need to downcast if the return type is changed to a subclass of the previous return type.

Other are changed due to this refactoring. All the subclasses of Cell were modified to change the signature and add a return statement. Also, any class that invoked the playAction method had to change since a String parameter is now a required argument. Eclipse modified those classes to supply null as the argument. Since the playAction method doesn't do anything with the parameter this is acceptable. But if the string was to be used, all usages of playAction would need to be updated with a meaningful string argument.

Task 13

Added a new class PropertyCellProduct after conducting the JDeodorant Bad Smell refactoring in which we used the God Class and extracted the getters and setters for the rent and the calculation for rent monopolies

Task 15

Project 2 was based on a Java-based Monopoly program that was developed at North Carolina State University. This program focused on maintenance and how to improve a program using refactoring. Some of the refactoring scenarios used was renaming a Class Field, Changing a Class Hierarchy, extracting an Interface, extracting a method from code, creating a local variable from repeated code, changing a method’s signature, and removing bad design smells by refactoring. It also focused on providing comments throughout the code via Javadocs. The JUnit test cases provided a sanity check to ensure that as changes were made to the code, the functionality was not impacted. We also made modifications to JUnit test cases to better understand the impacts to the test case and also the potential failures.

Refactoring helps with software maintenance by making the code easier to understand and provides a better structure. Having maintainable code is a topic that’s repeatedly discussed in class in which you want to ensure that the code you develop is easy to understand and change. This project allowed us to utilize the tools within Eclipse to modify and improve the code easily.

Eclipse's support for refactoring is comparable to what is provided by other Integrated Development Environments. Its ability to find references and usages throughout the project makes the refactoring tools a strong alternative to a simple find and replace. The ability to preview all changes is helpful to understand how exactly a refactoring will affect the project. If the previewed changes don't seem to make sense, that is a good indication that the refactor is not necessary.

JDeodorant seems to be a great tool if a large, unfamiliar project needs to be better understood or refactored. The tool can present a snapshot of potential problem areas for further review. However, JDeodorant does not understand the conceptual ideas behind the code - only its structure. The tool provided many suggestions that did not make sense logically.

We think that JDeodorant is a tool that can be used occasionally to identify refactoring suggestions. Programmers that develop or maintain the code daily would most likely already have a good idea of where refactoring is needed.

Unit tests are important when doing refactoring because it ensures the program can compile and there are no errors in the methods when code changes. It verifies that the changes made to refactoring makes sense and the changes did not break other parts of the application, so you need a strategy to handle regression testing for the changes made. Unit tests are not 100% fool proof because you can make changes and the unit test cases passes, but that does not mean that you didn’t break something else in the code.