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

***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 (of if) this operation is any different than a simple find all and replace.***

Eclipse's refactoring operation's main purpose is to make changes easily without affecting the behavior of the program. Eclipse's refactoring operations allows you to preview all changes prior to carrying them out. It is during the preview that Eclipse will inform you of all potential problems/issues.

In review of the refactoring preview that was performed by eclipse, failure for Eclipse to rename the "owner" parameter to "theOwner" was NOT an oversight. Eclipse recognized "owner" as a parameter within the setTheOwner() method. Since the method Player() returns theOwner, and theOwner is passed into setTheOwer() method as "owner", the parameter was not renamed from "owner" to "theOwner". The value of "owner" is passed to "theOwner” within the setTheOwner() thus not effecting the behavior of the program. The Eclipse refactoring operation is different from executing a simple "Find All and Replace" action. If you were to execute the "Find All and Replace" action within an IDE, everywhere "owner" is found will be changed to "theOwner" throughout the entire program which would result in potentially changing the behavior of the program. This action can be very dangerous and can potentially introduce bugs/errors within the program that won't be discovered until the program is run. As a result, the "Find All and Replace" action should be used with caution.

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

***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?***

Eclipse's refactoring Push-Down operation moves the selected fields and methods from the class to a subclass. Eclipse's refactoring Push-Up operation moves the selected fields and methods to its super class or declare its method as abstract.

Using the "Push Down" action was much easier to complete by the fact that no additional actions needed be completed all classes/files that would have been impacted were automatically modified. The "Push Up" action was a bit more cumbersome where by each file that was previously modified by the "Push Down" action would have to modified when executing the "Push Up" operation. If a file/class was missed via by you not selecting it, then you would have to re-perform the "Push Up" action. I found that I needed to search via the explorer view to ensure that previously modified files via the Push down action were modified by the Push Up method. This would have been very time consuming and irritating if this would have had to be done for each subclass.

The "Push Down" refactoring feature could help to eliminate the class-level code smell known as "Refused Bequest". The "Refused Bequest" code smell is a result of inheriting code that is not wanted. To eliminate this code smell, you would need to write code to "refuse the bequest" which can result in the code being confusing and difficult to maintain.

The "Push Up" refactoring feature could help to eliminate introducing code smells into your code by placing fields methods into a super class/creating an abstract class which allows code sharing across related class. This would eliminate the application-level code smells known as "Duplicate Code" and "Contrived Complexity", and result in the code being more maintainable.

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

***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.***

You would want to extract an interface to allow the code that uses the classes to access the interface via a generic API. Depending on the size of the program you are working with, this could require the modification of every piece of code that interacts with this class. This is done automatically using Eclipse via the built-in refectory functionality "Extract Interface" which increases efficiency.

In performing this task, I found this to be a great asset as it allowed the developer the ability to preview the changes prior to committing the change to the code. This would save the developer time from having to modify the code by hand which could lead to potentially missing code that wouldn't be detected unless an error would be introduced as a result or Unit Testing was conducted.

Extracting an interface prevents the application-level code smell of "Duplicate Code" which is defined as similar or the same code in more than one place. It also prevents the class-level code smell "Large Class" which occurs when there are too many instances of variables resulting in the class having too many responsibilities. This would result in the code being difficult to maintain by eliminating the duplication of code, having the code located in a more centralized place so that if the code needs to be maintained/modified it is easily done by modifying the code in one single location verses having to modify the code in multiple place which in turn could result in the modifications to other areas of the program being missed resulting in errors.

When executing the "Extract Interface" refactoring operation, I selected to extract the following methods: getName(), getPrice(), isAvailable(), playAction(), setAvailable(), getTheOwner(), setTheOwner(), and toString(). The iOwernable.java file was created as a result of this action.

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

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

Extracting a method allows you to create a new method that contains the statements that have been selected, replacing those statements with a new method. This helps to clean up lengthy and complicated methods as well as allowing for reuse of this method by other parts of the program. This is done by taking large pieces of code and turning them into methods.

I found this to be incredible simple however you need to be sure that all declarations are included in your method selection. I have chosen to include the string declaration in the method since I felt that it made the code readability cleaner. I felt it would be easier to just have the "int" passed in verses having to pass in the entire array. I felt it would make the processing of the code more efficient. The array needed to be incorporated into the method anyway so I felt it would be a better decision to include it within the method extract. This also helped to eliminate the method-level code smell of "Too Many Parameters".

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

***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.***

Eclipse makes creating a local variable from repeated code extremely easy adding to the efficiency of the programmer. Eclipse created a new variable from the selected expression, replacing the expression with a reference to the newly created variable.

This helps to eliminate the application-level code smell of duplicate code and coupling. This causes code to be difficult to modify as well as the adding to the complexity where each change that is made could introduce bugs that need to be fixed should the developer fail to perform unit testing.

With regards to methods/functions it could impact the correctness of a program. "Mutable local variables means that within a method execution you have changes of state. That makes it very difficult to change anything without breaking things". Immutable local variables would need to be passed to the method as parameters but with mutable variables, you need to make sure that the changes to those variables are propagated through the entire method/function. As a result, mutable variables can add complexity to methods structure by potentially many dependencies between the code in the method/function which can cause the code to break if something was moved/changed.

**Task 11 - Refactoring: Changing a Method's Signature**

***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 that just Cell.java and how this affected the definitions of any other classes besides Cell.java.***

Eclipse makes changing a method's signature extremely easy adding to the efficiency of the programmer. Eclipse checks the entire program to see which files are using the method that you are changing and offer a preview function prior to committing the changes. These files were impacted by this modification because the methods were called/used in other .java files. Eclipse was smart enough to search the project for those files that implemented the method. As a result, the return type of these files needed to be modified as the original method return type was "void" prior to the method signature change that was performed. After the signature modification, the return type was changed to Boolean so a return of either true or false was expected. Without the return type being returned, an error would occur. This feature allows the programmer the ability to quickly change parameter names, parameter types, parameter order, method return types in which all references to this method are updated. This is another extremely helpful feature to the programmer preventing those methods being missed potentially resulting in bugs being introduced into the program that may not be detected until after the program is ran if the developer failed to execute unit testing.

The "Change Method Signature" refractory operation helps to eliminate method-level code smells "Too Many Parameters", "Excessively Long Identifiers", "Excessive Short Identifiers", and "Excessive Return of Data". These code smells can make the code difficult to understand adding to the complexity to the program making it difficult to maintain.

**Task 13 - Detecting Design Smells**

***For this task, briefly describe the refactoring you made (if any) as a result of using JDeordorant.***

After installing JDeodorant, I was able to run a check against common code smells via the "Bad Smells" menu option that appear in the upper menu bar. Each of the code smells were run against this program. When checking the project for a particular code smell, I would highlight the root "scr" directory to ensure that all files under that directory structure were analyzed. Each time a particular code smell was executed, I would save the results of the findings prior to implementing the suggested refractory changes. I would also review the changes that were suggested prior to implementing the code changes. After all changes were made for each code smell, I ran the JUnit which resulted in NO errors being detected. Each code smell I checked for was present in this project.

Using JDeordrant, I was able to detect code smells that I would have not otherwise have picked up in viewing the source code of each file manually. This tools is extremely beneficial in helping to detect code smells that were introduced into the source code of the project after refactoring the code. This is especially beneficial for those project that have contain many files and/or have multiple people working on a project. For smaller projects, it is much easier to see if there have been any code smells introduced into the project, especially if you are the only person working on that project. However, when you are working with a group of developers each working on a piece of a project, there is the potential for code smells, to be introduced into the project without awareness. I was surprised to see how many files were affected and also the code smells that were introduced.

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

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

In running each of the code smells listed within the "Bad Smells" menu options, the only code smell that was detected within my code was "Long Parameters". I saved each of the results as text files for later viewing, previewed the suggested refactoring changes, and then proceeded with committing the suggested changes. Since this the Project 1 program was only one file, I was able to do a good job in ensuring that there was no duplicate code. Since the program written wasn't very big (not having hundreds of lines of codes), "Feature Envy" and "God Class" code smells was not detected. As a result of executing the "bad smells", not many code smells were detected due to Project 1 being a very small program consisting of one file.

**Task 15 - Summing it all UP**

In completing this exercise, I have learned the importance of code smells and refactoring your code for maintainability. Maintainability is defined as "the ease with which a software system or components can be modified to correct faults, improve performance, or other attributes, or adapt to a changed environment". Various code smells make it difficult to maintain/change your code. The refactoring tools offered by Eclipse help the programmer optimize their time usage when developing programs. These allow the developer to become more efficient by minimizing time-consuming practices of manual refactoring code via the reorganization of classes for a better, more modular design.

With performing refactoring, I was able to see how different parts of the program effected and/or depended on other parts. I learned the importance of continually performing unit testing to ensure that when code refactoring is performed, that the program behavior is not impacted while improving the internal structure. The primary goal of refactoring is to make the code easier to maintain. Refactoring helps to improve human readability which is also essential in being able to maintain the code. One would not be successful at attempting to maintain code that is extremely complex and difficult to follow. For example, dealing with spaghetti code makes it very difficult to modify/add code without something breaking elsewhere in the project.

I have also learned that even if your code runs perfectly, there is still the possibility that "code smells" could have been introduced within your program. Code smells become more difficult to detect the larger the program is and the more people that you have working on the project. This was demonstrated when running the JDeordant on a larger program versus a smaller program. With a smaller program and being the only person that was working on the program, very few code smells were introduced. However, when working with the Monopoly program, due to the size, it was clear how code smells could have been introduced into the program without developer awareness.

Eclipse made it very easy for a developer to check the code for emerging "bad smells" in addition to making the use of automated refactoring tools to refine your code much faster than if you had done via a manual effort. However even though the Eclipse refactoring tool does make it easier for you to accomplish the refactoring of your program code, you must also use the tools with caution. There is the potential for Eclipse to refactor code that perhaps should not have been refactored causing you to have to make manual adjustments. For example, when using the using the "Push Down" action was much easier to complete where no additional actions needed be completed as it modified all classes/files that would have been impacted. The "Push Up" action was a bit more cumbersome where each file that was previously modified by the "Push Down" action would have to modify as well. If a file/class was missed via by you not selecting it, then you would have to re-perform the Push Up action. I found that I needed to search via the explorer view to ensure that previously modified files via the Push down action were involved/modified via the Push Up method. This would have been very time consuming and irritating if this would have had to be done for each subclass.

Using JDeodorant made detecting of "bad smells" much easier however, not all of the code smells that exist can be detected by this tool. As a result, it is still possible for code smells to continue to reside in your code and possible additional coded smells to be introduced should you be working on a large project with multiple developers. Even though JDeordorant does make it easier to detect and refactor common code smells within your code, you still need to be mindful that other code smells may still exist. The developer should continually refactor the code as it is maintained to help prevent/limit/reduce the number of code smells in your code.

Every change that you perform in the code needs to be tested. Although refactoring is extremely important, unit testing makes it possible to refactor your code safely ensuring the functionality has not changed. When performing any type of automated or manual refactoring of code, it is essential that you perform a unit test to rule out the possibility that any new bugs have been introduced. Doing so will ensure that program functionality has not been altered. Should a bug/issue be introduced via a refractory process, it will be easier to detect the cause of this issue by conducting/running a unit test verses waiting until after all modifications have been completed and then performing a unit test.

As a result of this exercise, I have a solid understanding regarding the importance of refactoring and also those difficulties that could be missed due to the lack of domain knowledge when maintaining a program you are not familiar. Refactoring should be considered a process which is performed continuously during the entire lifecycle of the software by the developer.