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

The omission of renaming the set method parameter is not an oversight because the scope of the parameter is local to the set method and leaving it as is will not break the code. The ‘Rename’ tool is different than the ‘find and replace’ tool because it is more intelligent, that is, it will only make changes that are necessary to ensure the code is not broken. The find and replace tool will blindly change all occurrences.

**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.***

I made the change using the push down refactoring tool, confirmed the result as was described in the instructions and submitted a commit. I then made the change using the push up refactoring tool, ran the JUnit test, which resulted in no errors, and submitted a commit. I realized that moving a superclass field down to its subclasses causes duplicate code, adds complexity, and can modify the behavior of the original code.

Pushing a field of a subclass to a superclass will reduce duplicate code and complexity smells. Pushing a field down from a superclass to a subclass will reduce poor cohesion and facilitate introducing additional subclasses.

**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.***

I made the change using the ‘Extract an Interface’ refactoring tool, ran the JUnit test, which resulted in no errors, and submitted a commit. I realized that some of the methods that I included in the extraction did not make sense, for example, playAction() is not associated with an ownable class. I undid the extract interface operation and ran the tool again and extracted only the methods that are applicable to the IOwnable interface. If a class is ownable than it has a price, it may have an owner, it may be available, etc. I added the following methods to the extracted interface: getName(), getTheOwner(), getPrice(), setName(), setTheOwner(), toString(), isAvailable, and setAvailable(). I left out playAction() since not all ownable class are associated with a play action. A file named IOwnable.java was created which included the code of the extracted interface. The IOwnable interface now overrides methods of the original Cell class interface.

Extracting an interface can help to reduce duplicate abstraction, when two or more abstractions have identical names or implementation or both. The interface forces any class that is “ownable” to have all methods defined by the interface. They must appear in its source code before the class will successfully compile. It embodies the subset of “ownable” classes in the IOwnable interface. Interfaces allow more flexibility for multiple unrelated object types.

**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.***

I made the change using the ‘Extract method’ refactoring tool, ran the JUnit test, which resulted in no errors, and submitted a commit. I noticed that including the string array declaration reduced the parameter list of the new method signature. This is because without including the declaration, the tool identifies that a string variable is used in the loop and will need to be passed into the method. When the string array declaration is included in the selection, the tool recognizes that the string argument is not needed. I included the string array declaration to reduce the parameter list of the new method signature because the method that the loop was extracted from declared the string array in the same way, by directly accessing the “theOwner” object.

**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.***

In Eclipse, I clicked on the GameBoard.java file in the Project Explorer, then clicked on the Outline window, and clicked the addCell() method to locate the method. I then I highlighted one of the cell.getColorGroup() uses and performed the ‘Extract local variable’ refactoring operation. The tool created a declaration statement using the cell.getColorGroup() method and replaced the two uses of the method with the new local variable ‘colorGroup’.

Creating a local variable from repeated code will reduce the duplication code and complexity design smells and narrows the changes required if the method were to change. It also reduces the branches or loops of a method and makes it easier to comprehend.

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

In Eclipse, I clicked on the Cell.java file in the Project Explorer, then clicked on the Outline window, and clicked the playAction() abstract method to locate the method. I then performed the ‘Change Method Signature’ refactoring operation adding a boolean return type and a string type parameter named msg. I noticed, using the preview feature, that the original method was deprecated in the Cell superclass and in each of the subclasses. I executed the refactor operation and then noticed the subclasses all had the same return type error in the playAction() method. I fixed the error and ran the Junit test, which resulted in no errors.

Changing a method signature by reducing the number of parameters reduces the too many parameters bad smell and can reduce the complexity bad smell. By reducing the parameter list you make the code easier to read and reduce complexity of calling and testing the method. It could be that the purpose of the method is impractical and should be refactored to better assign responsibility.

Changes were made in the subclasses of the Cell superclass because the subclasses extend the interface of the superclass. The subclasses inherit all the fields and methods of the Cell superclass. Nonabstract subclasses of Cell, such as CardCell, must provide implementations for the playAction method. After the refactor operation, the return type of the method in the nonabstract subclasses no longer agreed with the superclass.

**Task 13 – Detecting Design Smells.**

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

I performed a Feature Envy smell detection operation with the JDeodorant pug-in. It found the Feature Envy bad smell for the completeTrade(), btnDrawCardClicked(), sendToJail(), and movePlayer() methods of the GameMaster class, in the purchase() method of the Player class, and in the displayInfo() method of the PlayerPanel class. I executed the refactor operation to move the completeTrade() method from the GameMaster class to the TradeDeal class and ran the JUnit test, which passed, and sent a commit. I refreshed the Feature Envy window and selected the next bad smell occurrence. I executed the refactor operation to move the purchase() method from the Player class to the Cell superclass and ran the JUnit test, which failed. The failure was due use of the ‘position’ variable in a declaration statement within the purchase() method. The Player class that the purchase() method was moved from used the ‘position’ class variable. After it was moved to the Cell superclass, the ‘position’ variable was no longer available. So I had to do a little research to learn how to revert the most recent commit with the EGit plug-in. After some review of the EGit user guide, I reverted all local and staged changes back to the current HEAD (or last commit on the master branch). I then ran the JUnit test and it passed. I continued with reducing the Feature Envy smell, but with more consideration regarding how the change will impact the code and if it makes logical sense. I decided it is more logical for the btnDrawCardClicked(), sendToJail(), and movePlayer() methods to remain in the GameMaster class than to move them to the GameBoard class (i.e. a game board can’t move a player, that’s the task of the game master).

I had to do some research to learn how to perform the Duplicated Code smell detection operation. I followed the instructions on the Eclipse Wiki:

<http://wiki.eclipse.org/Duplicated_code_detection_tool_(SDD)#CVS.2C_downloads>

I successfully executed the detection which resulted in identifying duplicated code in the constructor of GameBoardCCJail, GameBoardCCLoseMoney, GameBoardCCMovePlayer, and GameBoardCCGainMoney classes. The constructors on these subclasses had some identical statements. I decided to try to pull up some of the constructor body into the superclass constructor and call it from the subclass method. When I did this, it caused many errors to occur because the other subclasses call the superclass constructor but with a different signature. I reversed the change and decided that this smell was not significant enough to warrant refactoring.

I performed the God Class smell detection operation, which resulted in some smells listed. I created a PlayerProduct class to reduce the God Class smell in the Player class. I also created a GameMasterProduct class to reduce the God smell in the GameMaster class. Both refactorings passed JUnit tests and I submitted appropriate commits.

I performed the Long Method smell detection operation, which resulted in some smells listed. I extracted the gui() and player() methods within the GameMaster class to reduce the Long Method smell. I also extracted the owner() method within the Player class.

I performed the Type Checking smell detection operation, which resulted in some smells listed. I attempted to refactor one of the smells, but it resulted in errors. I then I reverted all local and staged changes back to the current HEAD (or last commit on the master branch). I then ran the JUnit test with successful results.

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

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

I performed a Feature Envy smell detection operation with the JDeodorant pug-in and it detected no smells. I performed the Duplicate Code smell detection operation and it detected no smells. I performed the God Class smell detection operation and it detected no smells. I performed the Long Method smell detection operation and it detected three (3) smells. I preformed the refactoring operation which extracted the tempValue() method and then ran the program and encountered no errors. I preformed the next recommended refactoring operation which extracted the tempInt() method and then ran the program and encountered no errors. I then refreshed the Long Method results list which showed only one bad smell. I started the last refactoring operation which tried to create another tempValue() method. I modified the name of the method to tempValue1() and continued with the refactoring operation. I then had to manually update the local variable from tempValue to tempValue1 and initialize the variable.

**Task 15 – Summing it All Up.**

What I learned from this project is how to identify bad code smells, what refactoring techniques to use to reduce them, and experience in implementing refactoring techniques. I saw how coupling can cause rippling effects and make it difficult to modify code. I saw how low cohesion can cause bad smells like Feature Envy and Long Methods.

Understanding of the code can be improved by ensuring that the code is appropriately documented and intuitively designed. This can facilitate awareness of execution behavior and cause-effect relationships and helps to conceptualize the result of the refactoring change being considered. It is better to avoid making a change by accurately predicting the negative result, than to make the change and have to undo the change.

What I like about the Eclipse refactoring tools is the built-in intelligence. Of course, if you are not aware of the impact of the change you want to make, it can really break your code and cause you to spend a lot of time cleaning up. However, if you know what you’re doing and are aware of what the tool is about to do, it can save a lot of time. The Eclipse refactoring tools point you in the right direction, but you still need to verify that it is the right direction.

I really liked the JDeodorant plug-in, one of my responsibilities as a government employee is to assess the quality of the code delivered by contractors. This tool would significantly reduce the time to determine the quality of code and also reduce the cost of refactoring it to improve quality. I’m going to look for a plug-in that I can use with our development environment. If I can’t find one, I may try to migrate our software to the Eclipse environment just so we can take advantage of the refactoring support tools.

I also experienced the convenience of unit testing. I was able to run a full test and see the results immediately after making a change to the software. This is very powerful in that it prevents you from going too far with changes without knowing if you’re introducing errors. The near instantaneous verification of software allows you to confirm that each change is valid, and makes it easier to maintain code and ensure quality.