**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 renaming a class field using refactoring is a good way to maintain the original code and avoid the errors, so it is an oversight on the part of refactoring operation. The change on “owner” actually changed all the “owner” in the project or its related classes, this is different with the simple find and replace. Normally, it take lots of time manually to find the target you want to change and you may still miss it somewhere. In this way, you don’t have to worry about the location of targets. The computer does the work for you and it is accurate.

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.***

When an operation implemented in a superclass are used by only one (or a few) subclasses, we should use Push Down to move that operation to the subclass. When subclasses have methods that perform similar work, we should make the methods identical and then move them to the relevant superclass.

When I did Push Down of available, I did at same time for the isAvailable() and setAvailable(). The result is that the GameMaster.java and Play.java turned red. When I looked into the methods, they called the isAvailable() and setAvailable(). But they are not in their superclass Cell now. So I did Pull Up, there are lots of issue came out here including several errors came out. Only when I Pulled Up isAvailable() and setAvailable() at the same time, the original code was recovered. I ran the Junit test. It was fine.

I think Push Down [refactoring](https://en.wikipedia.org/wiki/Refactoring) is useful when a functionality in superclass is not commonly used by subclasses, just take it to the subclasses that will use it. Similar, if subclasses of the same superclass share a same functionality, we should pull this functionality up to the superclass. Then the subclasses just call it later.

**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.***

When multiple clients are using the same part of a class interface, or part of the interface in two classes is the same, we should move identical portion to its own interface to increase the maintainability.

Upon extraction of interface IOwnable in Cell.java, I saw a new class (interface) called IOWnable.java was created in the project folder. I have placed some methods into the interface including getName(), getTheOwner(), getPrice(), isAvailable(), setAvailable, and setTheOwner(). In the Cell.java, some codes were changed including “public abstract class Cell implements IOwnable” is declared on the top of the class. This means it implements the newly created IOwnable interface. In such case, I think Eclipse changes all usages of the original class to use the interface where possible.

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

I used the second way to extract the method. As a result, a new method called calculateMonopoliesRent (int rentToCharge) was automatically created, and the loop code and declaration of monopolies[] in getRent() were removed. When running the program, it works well and is the same as previous version.

The second way would be better, because the monopolies[] is used only in this method, why we leave the declaration of it to the other method. Additionally, the too many parameters in method is also a bad smell.

**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.***

It sounds very easy to extract a local variable. I just highlight one of the repeated expression and do refactoring. A new local variable was created. I call it colorGroup. All the expressions in the class were changed to colorGroup. I think this is a way to keep the repeated expression simple. Once when you want to change the expression in future, what you need to do is simple. Just change the declaration of the variable. Normally it should be OK to do this to a void method call. If there are parameters in the method call, and values of parameters are changing in different code fragments, the extracted local variable cannot handle this.

**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.***

When I am working on *Change* *Method Signature* refactoring operation and change the method to Boolean with a String parameter msg. I realized that this would change lots of classes. I selected all of them. So all the classed have been changed based on the method playAction. The errors came out as expected. They all request a return statement because the void method has been changed to Boolean method. I went ahead and changed them one by one, and run the program and Junit test. Everything is fine.

*Change* *Method Signature* is a good way to re-define the method, Not only it can change the type of method, it can also add additional parameters, remove parameters and even change the order of parameters. So it is a nice way for maintenance. Of note, it will change all the related methods in other classes. The change of the type declaration of the methods requests a further manual adjustment of the method.

**Task 12**

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

I installed the [JDeodorant](http://users.encs.concordia.ca/~nikolaos/jdeodorant/index.php?option=com_content&view=article&id=45&Itemid=63), and used it to check the bad smells. There are lots of targets coming out. First thing I did is checking Feature Envy. In the table of the Feature Envy view I can see 8-10 refactoring opportunities identified for the project. They are “Move method”. I clicked on the "Apply Refactoring" button and did refractoring for rows one by one. Most of identified bead smells were removed. But two of them generated errors in JUnit Test when being removed. SO I keep them there.

Using the similar approach, I did applied the refactoring for the Type Checking, God Class, Long Method. There are too many identified bad smells. I did some of them and make sure to pass the JUnit Test. If cannot pass the test, normally I did the undo for the changed class. Additionally, we I have worked on Duplicate code, but the Eclipse ask me to provide additional check clone tool. Since I don’t have a check tool installed so I skipped the check for Duplicate code.

Task 14 ***For this task, briefly describe the refactorings you made (if any) as a result of using JDeodorant.***

After I applied JDeodorant analysis on my fire danger project, surprisingly I didn’t find any identified refactoring chances in Featur Envy, Type Checking, God Class, Long Method. The reason could be my project is very small, or just one class in the project with all methods in it, or my code is perfect? I have no idea about this. Anyway, JDeodorant cannot do a regular job on my code. Therefore I didn’t push again the code to the Github.

**Task 15. *Summering it All Up***

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

Upon the practice in this project, I have learned couple of refactoring methods to improve the software maintenance including extraction of interface, method, local variable, and bush down/pull up of the classes. I also learned how to use JAutodoc and Javedoc to make documentation of the project. Additionally, I got some experience in finding refactoring opportunities using JDeodorant for bad smells.

The refactoring methods are highly related to the topic covered in the lecture. For example, in the lecture 4, we have learned the concepts of refactoring and bad smells of code, and knew the approaches on how to refactor the code and remove the bad smells. In this project, we actually got the chance to practically hand on it and solve the real issues and real bad smells through a real project. Before working on the code, I don’t really understand, after going through all the tasks of project, I think I basically got what the refactoring is for, and what the bad smells are.

* 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

I think the Eclipse provides a powerful set of automated refactorings that let you move methods, classes and packages, extract interfaces from concrete classes, push down-pull up nested classes into other level of classes, and create a new method from sections of code.

The limitation is that we must be careful when applying the refactoring, because errors can inadvertently be introduced, especially when refactoring is done by hand. In my case, when I am working on JDeodorant-identified bad smells, some of refactoring process produced errors. For example when you move something in one class it affects other classes, and the error cannot be automatically recognized and corrected.

So my impression for JDeodorant is not very good. Its functionality in identification of bad smells is excellent, but its ability to refactor the code and remove the bad smells is limited. This may be caused by complex of Java structure and personal habit in coding. As a result, I spend lots of time in refactoring the JDeodorant-identified bad smell one by one. I got errors frequently and have to undo the process. The thing is I have to do unit test after each refactoring to make sure the refactoring is OK.

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

As mentioned above, the refactoring may generate some unexpected errors, it is no way to guarantee the accuracy. So we must do unit test along with the refactoring process. In this case, If unit test fails, we still have a chance to do undo and go back to the original code.

In addition, I think the techniques we used now for refactoring may not mature enough. Keeping doing unit test is the only way to maintain the integrity of the project and find the un-expected errors.