# Project 2

*COSC 603 Software Testing*

*Sigrid Berkebile-Stoiser*

*3/4/2016*

## Task 5

*Question: Why does Eclipse not change the input parameter “owner” of the setTheOwner function to “theOwner”?*

Eclipse understands the parameter scope of the variable. The input parameter of the setTheOwner method can have any name as long as it is assigned to the instance variable theOwner of the Cell class. This is the important variable that should have been correctly renamed. This refactor operation is different from a search and replace function because it intelligently identifies where the code has to be changed or not. When doing this with search and replace, we could introduce a lot of errors as variables might be changed that should not be changed, such as local variables named “owner” that are completely different from Cell.owner.

## Task 7

*Experience with this task:* I had problems with undoing the push down. When I tried pulling up the available field and the associated mutators, I seem to have made a mistake and in some of the classes, the mutators where not removed. So I had to remove them manually. Therefore, I would say that it is very easy to break the code if attempting this type of refactoring when working on a large project and being still inexperienced.

*Question: What design smells call for a push down or pull up refactoring in Eclipse?*

A pull-up of a method is useful to remove duplicate code. If you have many subclasses with methods that do the same thing, it is more efficient to implement this method in the parent which inherits it to all the children. If a maintenance programmer has to make a change to the method, he has to do it only in one place. Pushing down a method is useful if the method of a superclass is only used by one or very few of its subclasses. This improves class-coherence, i.e., methods are implemented where the developers expect them to see, and maintainers will be able to understand the code faster.

## Task 8

Experience with this task: I would actually not extract an interface to accomplish this task but rename Cell.java to “Property.java” and use subclasses of Property.java to describe all properties that can be owned by players of the game (such as cells).

To create the interface, I extracted the methods getPrice() and playAction(), as everything a player owns should do something in the game and cost something in order to make profit. I did not extract the setters and getters of the class (e.g., setTheOwner) because if I need these fields, I should rather subclass from Cell.java or a more general superclass of Cell.java that has these member fields. I did not see that any new files were created except for the IOwnable.java file. However, a couple of sub-classes of Cell.java were changed.

Extracting an interface is a good refactoring method for improving the generalization of the code. If several classes are using the same methods of another class, then it is a good idea to move these methods to a separate interface. The extraction makes the code more maintainable as the code becomes more logically organized. Grouping common actions into one place makes the code easier to understand.

## Task 9

Experience with this task: Extracting the method was fairly easy. As this type of refactoring makes changes locally, it was also easier for me to understand the consequences than when I tried out the pull up and push down operation. I extracted the **private** **int** calculateMonopoliesRent(**int** rentToCharge)signature, as it was shorter and the monopolies variable is not used anywhere else in the getRent() method. Thus, it makes sense to include it in the new calculateMonopoliesRent() method.

## Task 10

Experience with this task: the extraction of the local variable was easy to do and I did not experience any problems with it. This operation certainly makes the code more maintainable as it is easier to read for the programmer in case the new variable is named appropriately. However, care has to be taken that the new local variable does not interfere by accident with another variable of the same name somewhere else in the method. For example, a local variable of the same name could have been declared earlier and its value is lost when the new variable is declared. However, Eclipse might be able to recognize such a conflict.

## Task 11

Experience with this task: the change of the method signature did not cause any major compilation errors that were not easy to fix. However, I can see that this type of change can cause a lot of problems when not carefully done as each change has to be analyzed for logical problems and clashes of variable names.

The new method signature causes changes in other classes than Cell.java as many classes are inheriting the new method from Cell.java and need to implement the new signature. However, such a refactoring can make the code more maintainable if we have very long lists of input parameters which should be encapsulated into an object that is passed to the method instead of the long list. This makes the code easier to read and understand.

## Task 13

As suggested by JDeodorant, I refactored the method playerMoved in GameMaster.java by extracting a method. I actually thought that the method was very short (~ 20 lines) and refactoring was not necessary in my view. However, I tried to practice the refactoring capabilities of Eclipse a little more.

## Task 14

I did not follow any of the refactoring suggestions of JDeodorant as I liked my own design better than the suggested changes. However, I created a new object encapsulating the input parameters passed to the method that calculates the fire danger indexes in my program. This was done to eliminate the design smell “long parameter list”.

## Task 15

### Report

I learned from the project how refactoring can be automated and how code smells can be found without reading and analyzing all of the code first. Refactoring a project is important, especially if agile development methods are used which change the code very frequently and do not emphasize documentation. Thus, if tools help the developers find opportunities for refactoring and help insert the appropriate statements automatically, that is a big help. Also, for large projects, it is an error prone task to refactor completely manually as there are many interdependencies between classes and one small change lead to more changes in several other places. This could potentially take a long time to do by hand. Therefore, I can see how tool support saves time and reduces the risk of making mistakes when refactoring which will make the lives of maintenance developers easier.

The automatic detection of bad code smells is also a great way of finding opportunities for refactoring without being intimately acquainted with the code or having a lot of experience with identifying bad design. However, I found that it is probably not a good idea to blindly follow the suggestions of the tools but to determine on a case-by-case basis whether it really makes sense to refactor the code that the tool sniffs out.

I liked the support of Eclipse for refactoring as it offers a good preview of the changes and is fairly self-explanatory. Also, because Eclipse will find problems caused by the code changes automatically, the refactoring process is not too painful, at least with the small refactoring we did in the Monopoly game. However, I do not think that the automatic refactoring does the best job. Especially with extracting methods, I thought that more changes to the code were necessary to make the program more readable and understandable than just putting a chunk of code into a new method. I thought that re-writing the code in a different way to make the control flow with the new method more logical would have been better than automatic refactoring alone.

For me as a beginner, the JDeodorant tool was not extremely helpful. I did not agree with many of the suggestions for change when I looked up the code, especially with the proposed method extractions. The methods that were marked as too long were actually quite short in my view and would have become harder to understand with the method extractions instead of easier to understand. Also, some the solutions that were suggested were a little cryptic too me, such as “Replace Type Code with State/Strategy” for the design smell Type Checking. I also had problems with analyzing the code for duplicated code. JDeodorant prompted me to provide paths for files associated with an object cloning tool. I did not know what that meant and how to fix this issue as there was no example in the online documentation.

Unit tests are important when doing refactoring as they prove that the behavior as defined in the object interfaces and definitions is still the same after refactoring whereas the inner structure of the code has changed. Thus, if the unit tests run successfully, we know that we did not break the code. Certainly, refactoring can lead to the introduction of logical errors despite a successful run of the unit tests which have to be uncovered by other means.