Project 2

**Task 6** – This omission is not an oversight on Eclipse’s part. While the variable passed into the setTheOwner method is conceptually the same as the private theOwner variable, it is still a separate variable and a different object. When a variable (or function or class) is renamed, only the occurrences of **that specific** variable (or function or class) should be renamed. Items which may be functionally equivalent, but are not the same item should not be renamed.

**Task 7** – This task shows the versatility of Eclipse and its ability to refactor. Refactoring via eclipse not only saves time for mass updates, it also helps avoid user errors for such updates. Fields and methods should certainly be pushed up if they are used by every subclass. They should also be pushed up if a large number (but not all) subclasses use the given methods without a need for overriding the method if it was added to the superclass. Methods should be pushed down if they do not belong to the superclass logically. Methods which are only used by a few subclasses and methods which are very often overridden should be pushed down. This makes the code more maintainable since related functions and variables are easier to find. Making a change to the variables also becomes easier as a properly refactored code base would be less susceptible to errors and quicker to make changes in. For example, changing the type of a variable in a superclass would be inherited by subclasses saving time. Changing the type of a variable only defined in the subclass would not affect other subclasses which may have a variable of the same name which should not change.

**Task 8** – The methods setTheOwner, getTheOwner, isAvailable and setAvailable were refactored as a part of this interface. These methods relate to ownership and whether the object can be owned (availability). Other methods, such as getPrice were not refactored. A user does not necessarily have to purchase an object to own it. An object may not have a price and yet be owned by a user. getPrice should belong to its own interface. In this task a new java file called IOwnable was created. Extracting an interface makes sense when a group of methods describe a logical action. By extracting an interface, we can apply (inherit) the same interface on various objects which all share that action. For example, while cells are an object which can be owned, pieces representing players can also be thought of as being “owned” by a player. In this case, a class representing the pieces should inherit the IOwnable interface and in turn implement the same methods (setTheOwner, getTheOwner, isAvailable and setAvailable)

**Task 9 –** The method which was extracted was marked as private by default during the refactoring and preview state. I left this method as private since no other class is expected to call it directly. I ended up choosing the method signature which did not pass in the parameter monopolies. In one approach, the parameter would be defined in the original method and passed into the new refactored method. I chose the approach of defining the parameter in the refactored method since the getRent method does not use that parameter; the parameter would only be used for the new calculateMonopoliesRent method. Declaring and initializing the variable within the new method would increase cohesion of the new method and decrease the coupling (by decreasing the number of parameters passed in) between the two methods.

**Task 10** – In this task, not only was a local variable created, but all references matching the selected code for refactoring were also replaced with the new variable. This change was only performed for the function where the code was selected. Creating local variables helps with duplicated code smell. This could be useful to maintain if the function which was called changed. For example, if the function now required a parameter to be passed in, this change would only need to be done once where the variable is declared instead of changing it at multiple places. Refactoring a method into a variable is not always a good idea since methods often accept parameters and give different results for different parameter values. If a method called on line 5 uses a parameter, it is possible that the value of the parameter changed between lines 1 and 4. Using the same method but before line 1 (before the value of the parameter passed in changes) would result in the method returning a different value than before.

**Task 11** – There were many more manual changes with this task than the tasks before. I ended up return true for every implementation of the playAction method. Not knowing what the return variable would be used for and how it should be calculated made me question whether returning true on all occasions was the correct decision. Cell.java defined cell as an abstract class which defined the playAction method but did not implement it. All non-abstract subclasses of the class cell would need to implement that method. Since the return type of the method changed, every implementation of the method had to be changed to return a value of that type. Refactoring method calls might be useful for “shotgun surgery” where the code is highly fragmented and becomes easier to code/maintain by aggregating to a larger class.