UECS2363 SOFTWARE CONSTRUCTION AND CONFIGURATION

Practical 3 : Principles of Code Refactoring

This lab practical will explore code refactoring using a Monopoly game written in Java (complete with JUnit testcases) developed by North Carolina State University (NCSU).
The game is solely for educational purpose and fully belongs to NCSU. Thus, distribution for self-profit or self-credit is strictly prohibited.

Import Project Files.
Download Monopoly3.zip from WBLE. Then, launch Eclipse and create a new Java project. Import the Monopoly3 files into the project by:

- Make sure that project has a folder named "src" in it. If not, select the project name, then right-click and choose New -> Source Folder. In the pop-up window, name it "src".
- Right-click on the folder "src" and choose "Import".
- Choose the Monopoply3 zip file, and then click "Finish". (There is no need to change any other options).

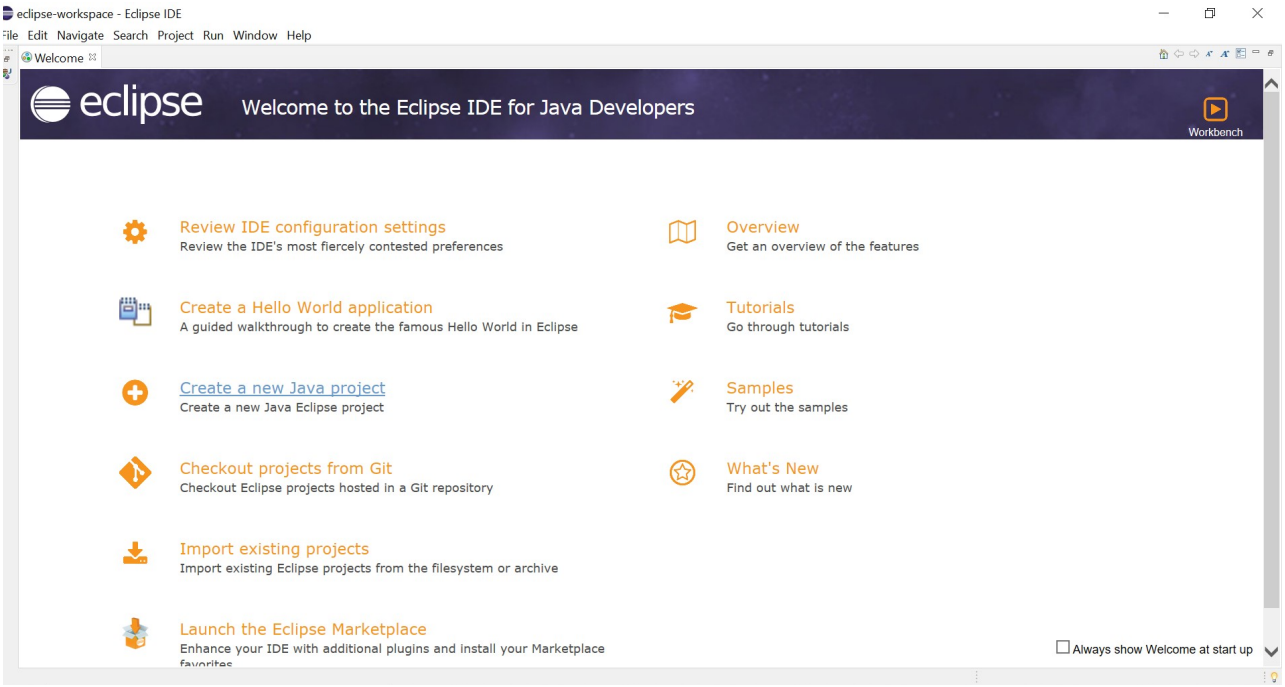

Figure 1: Launch Eclipse and select "Create a new Java project".

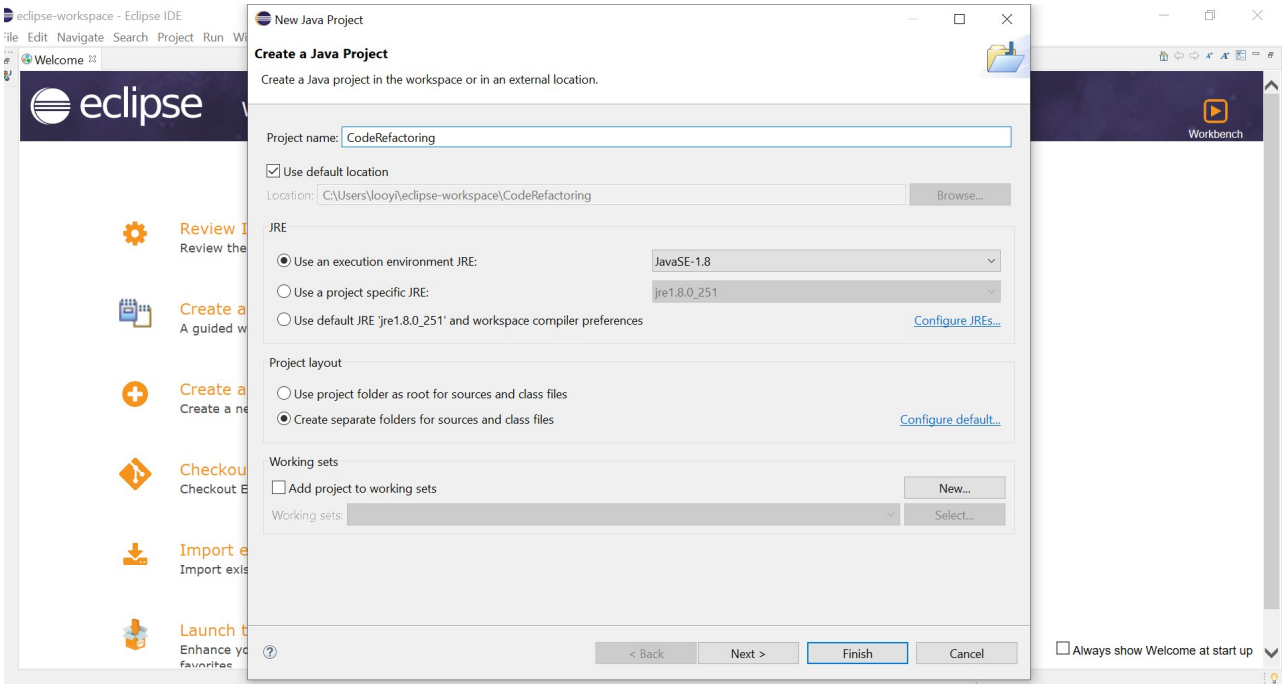

Figure 2: Create a new Java project according to own preference.

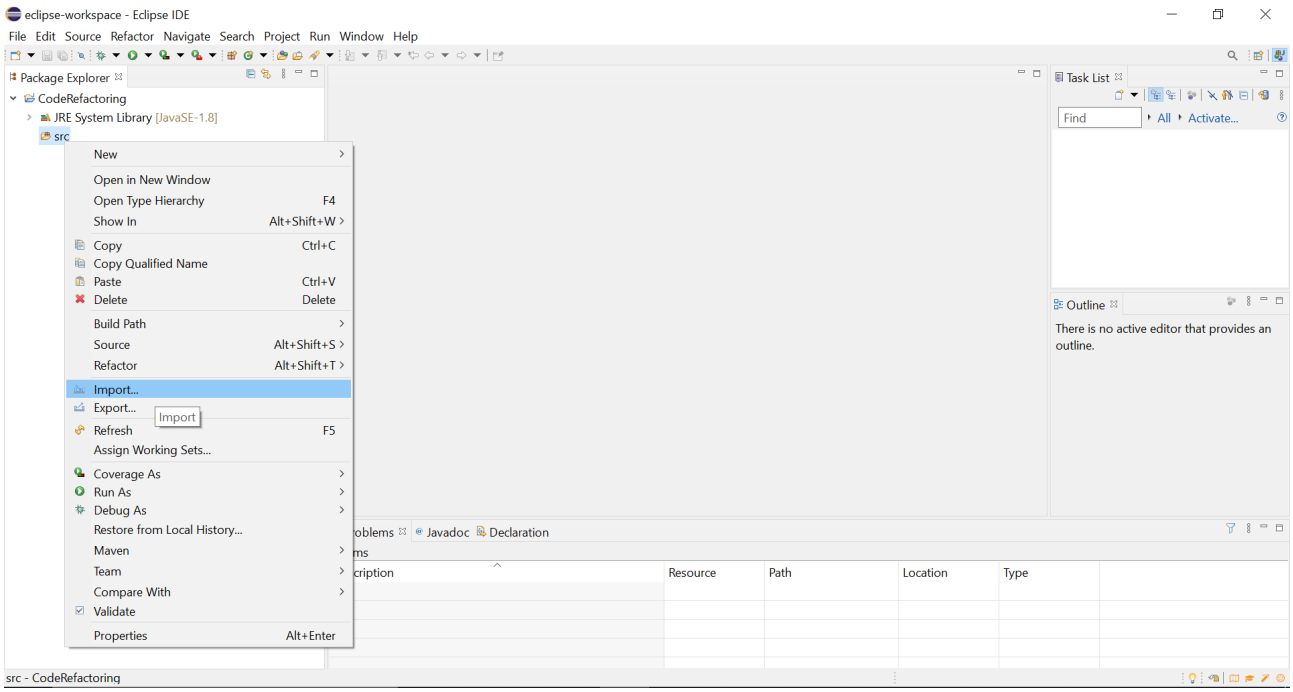

Figure 3: Right-click on the folder “src” and choose “Import” to import Monopoly project files.

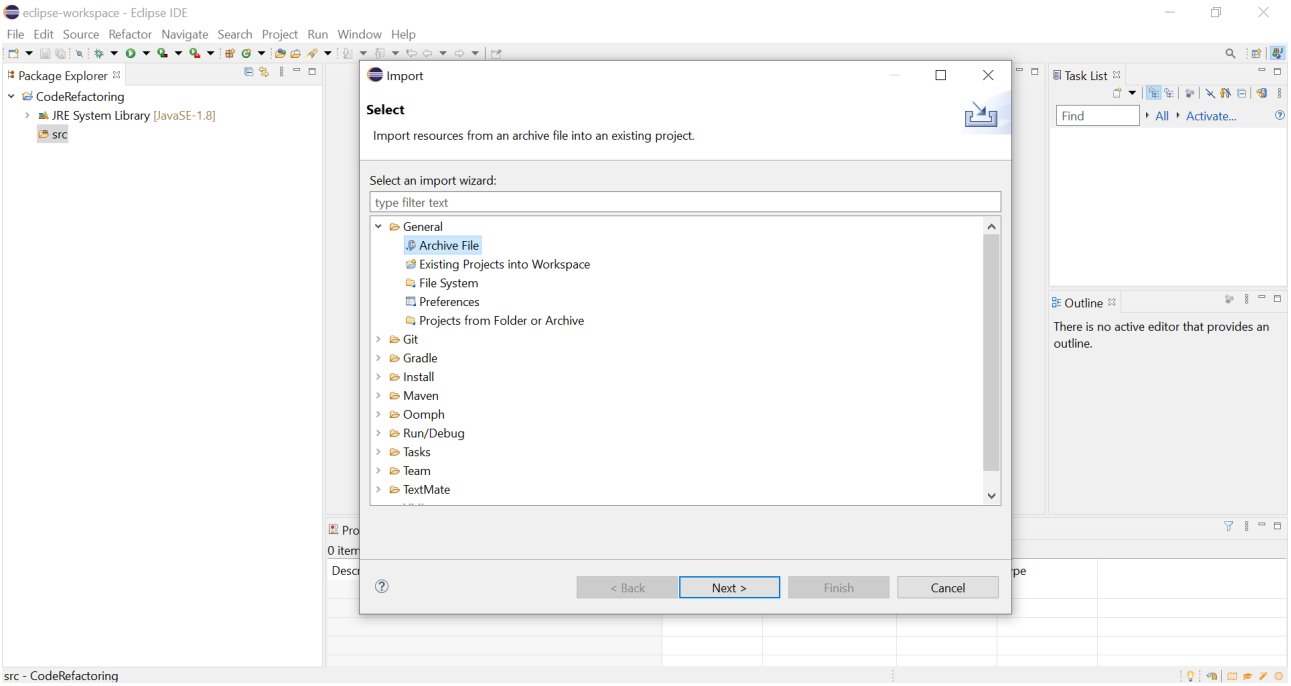

Figure 4: Choose “Archive File” option in the Import pop-up window.

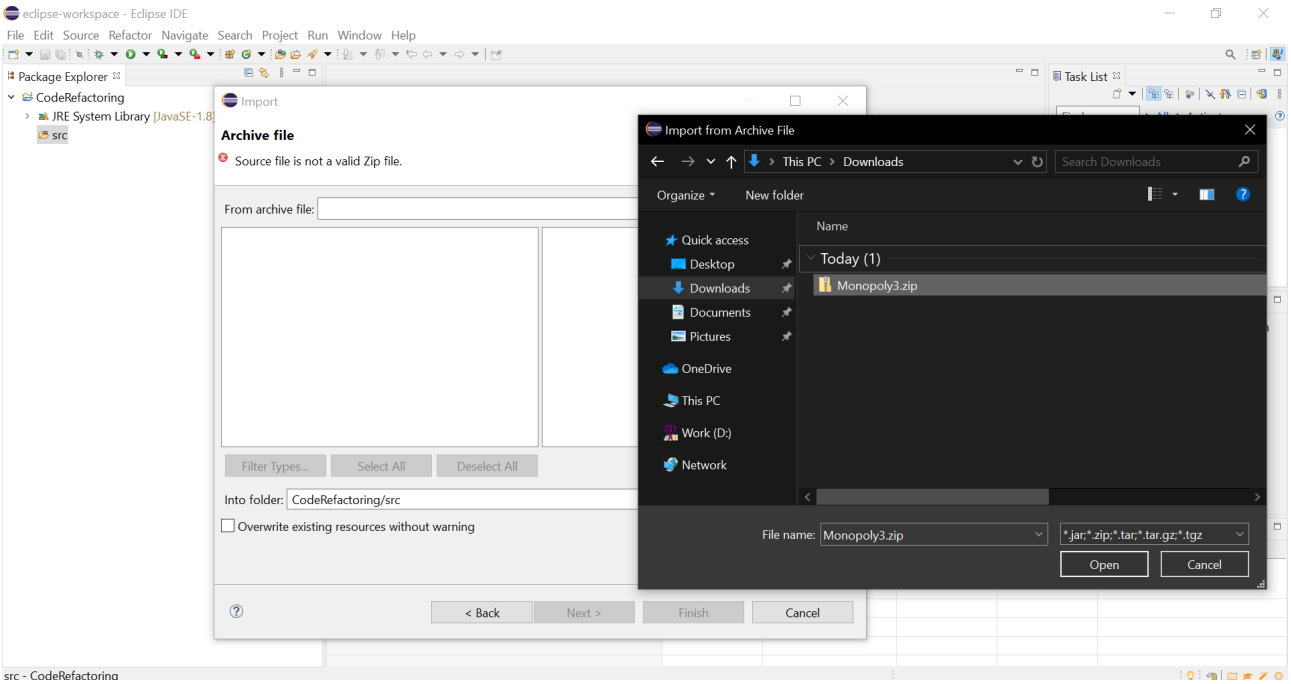

Figure 5: Browse for Monopoly3.zip downloaded from WBLE.

You should see the java files in packages under src but some of them will have red error-indicators.

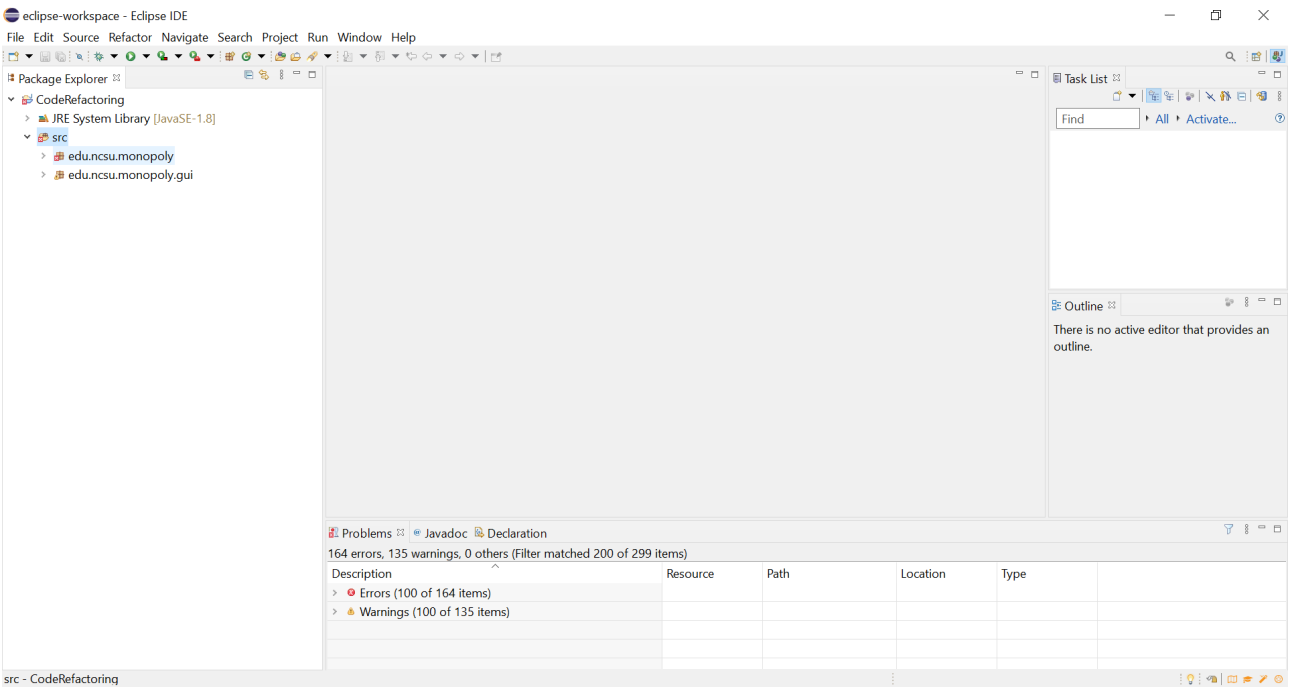

Figure 6: Errors in "...monopoly" package.

To fix that, you'll need to add JUnit to your project build-path after you import the files. Firstly, select "Project" menu, and then from the "Project" menu drop-down list, choose "Properties", then choose "Java Build Path" in the pop-up window.

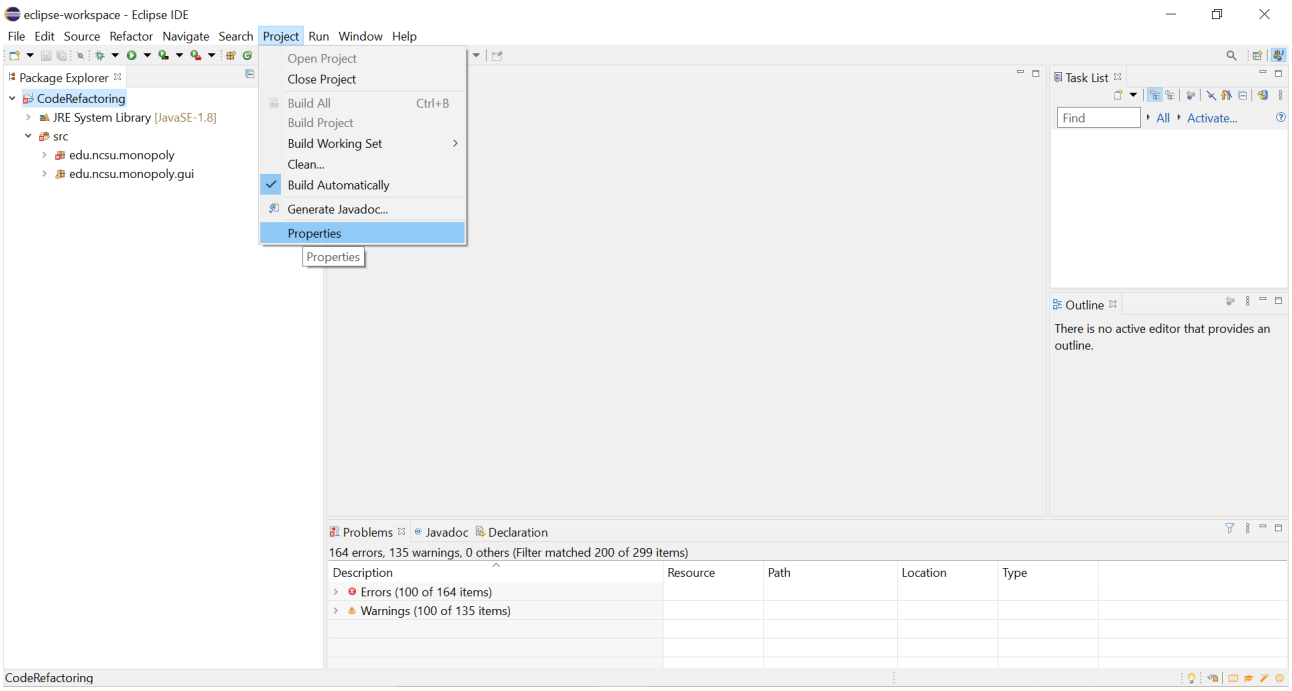

Figure 7: Select "Project" menu, and then from the "Project" menu, choose "Properties".

Prior to choosing "Java Build Path" in the pop-up window, choose "Libraries" menu then click "Add External Jars..." button. Browse for junit.jar in Eclipse plugins folder or in the junit subfolder of Eclipse plugins folder. (Usually, junit.jar is found in the plugins of Eclipse folder in C:\Program Files\eclipse\plugins\org.junit_X.XX.XX.jar)

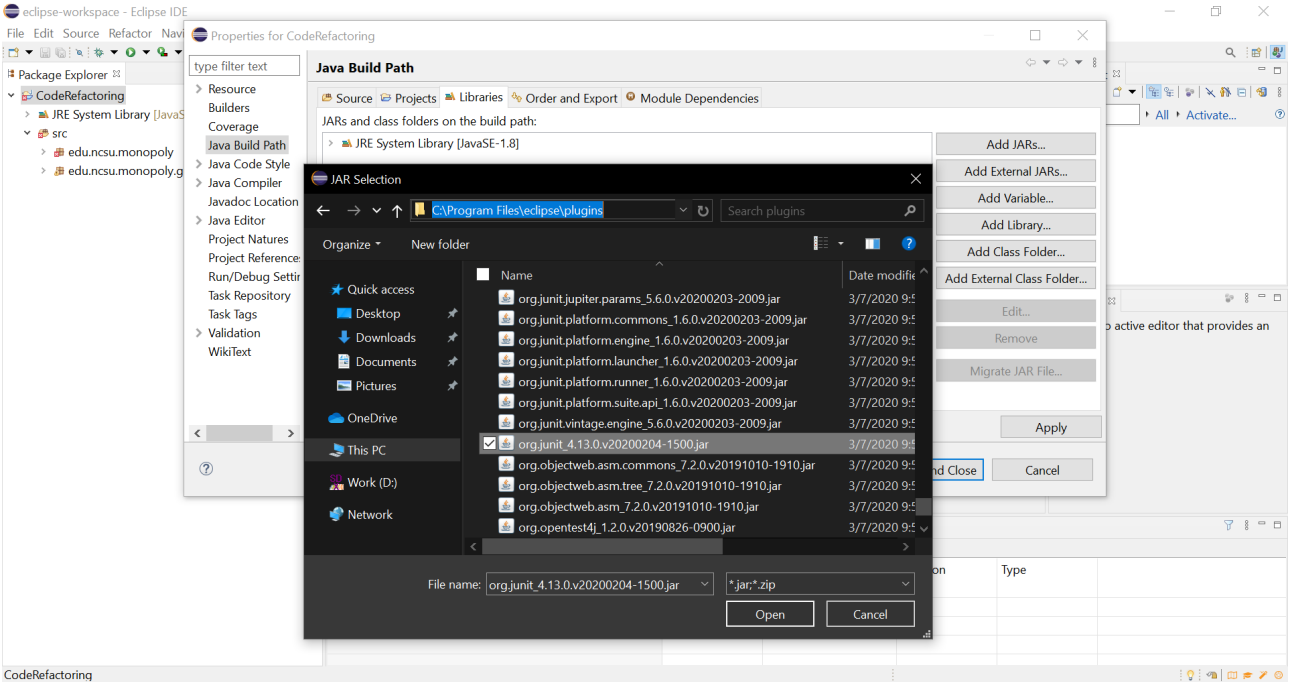

Figure 8: Add Junit into project build-path.

Once you click “Apply and Close” the the java files in packages under src with red error-indicators should be fixed by now.

Getting Started.

Firstly, let's make sure it works. Right-click on the “src” folder and from the “Run As” menu, choose “JUnit Test”. Is all well?

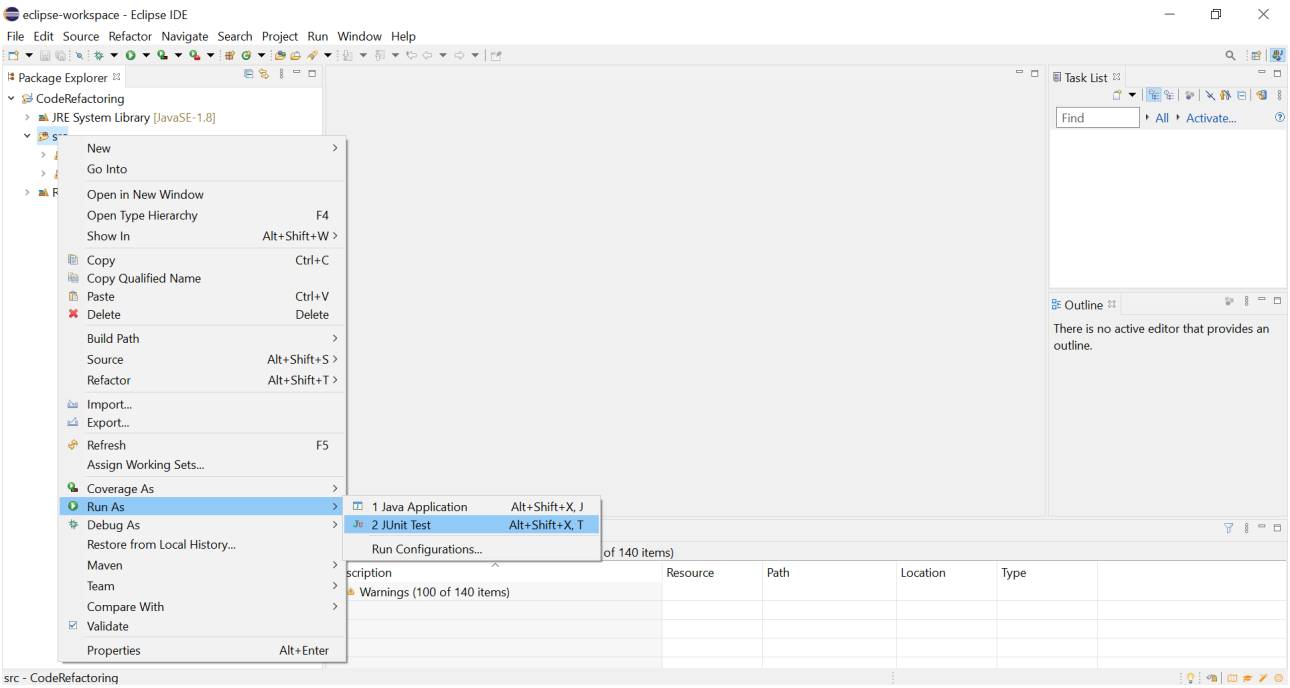

Figure 9: Running J-Unit test for the project.

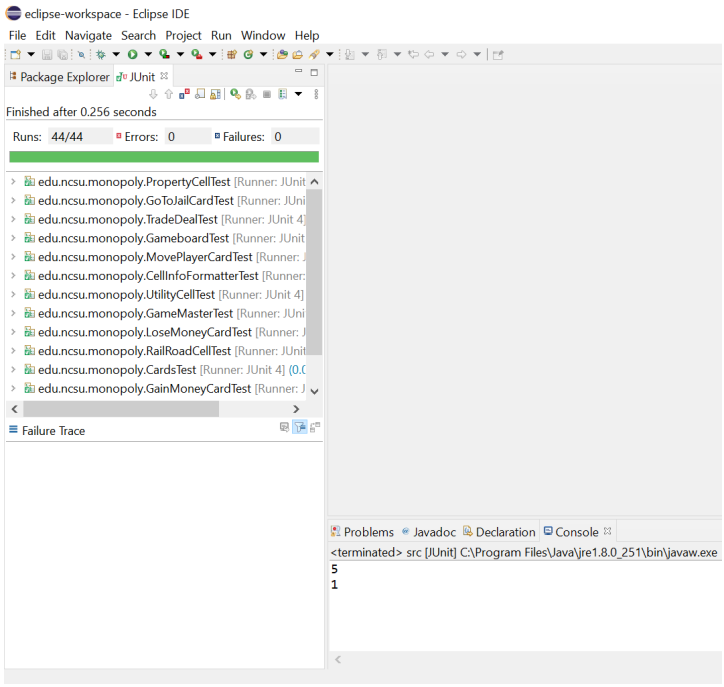

Figure 10: Successful J-Unit test.

Let's play Monopoly!

Q: What happens if you chose the edu.ncsu.monopoly package and run your program? What was implicated?

Renaming a Class Field

Class "Cell" in "...monopoly" package is an abstract superclass with many subclasses. You can see this by selecting that class or file, and pressing "F4" button to see the class hierarchy. You can also see in the "Outline view" that "Cell" has a field named "owner".

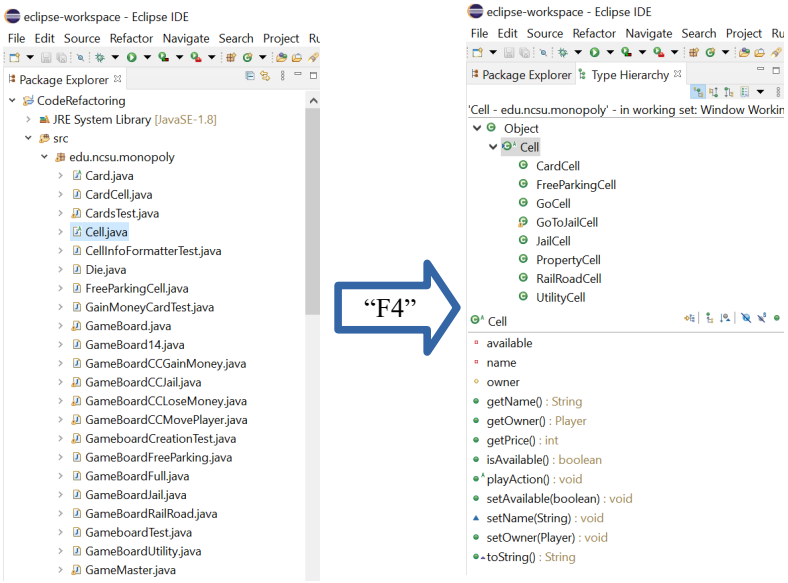

Figure 11: Choosing class "Cell" and pressing "F4".

Use “Rename” refactor operation to change the name of “owner” to “theOwner”. Select all the options in that menu (to change references, comments, and getters and setters). Press “Preview” to see what would change.

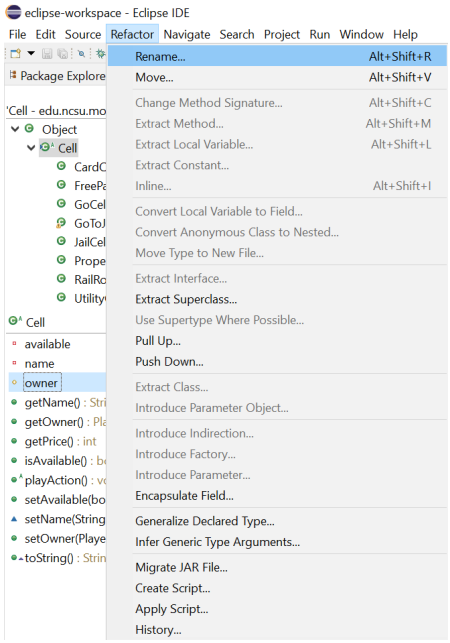

Figure 12: Choosing “Rename” in Eclipse’ refactor operation.

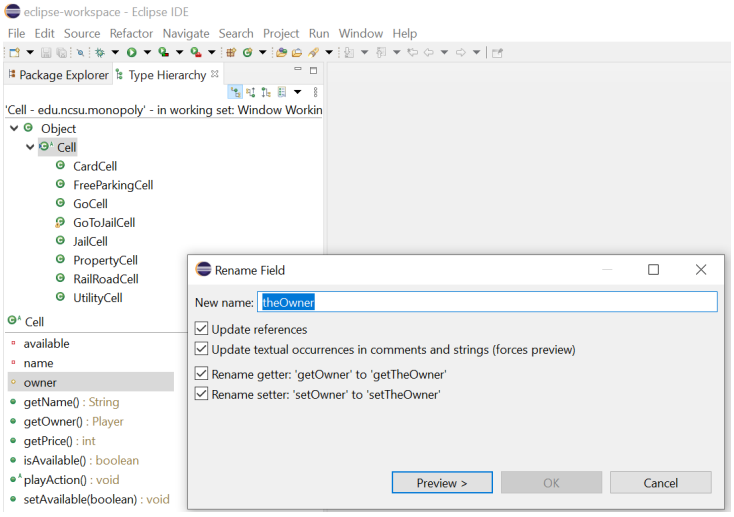

Figure 13: Change “owner” to “theOwner” for all occurences.

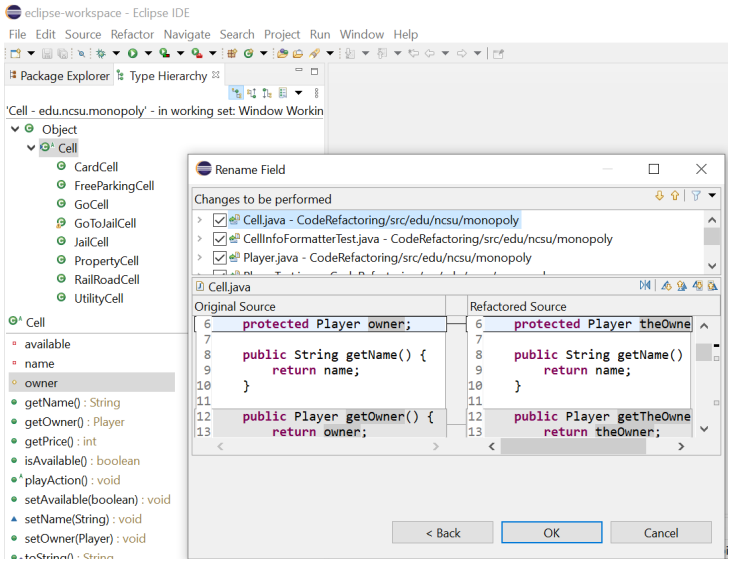

Figure 13: Preview changes.

Is everything changed? Use the Search->Java to find all references to a field named “owner” and check if anything was left unchanged. Then, search for all references to a field named “theOwner” to verify all was updated.

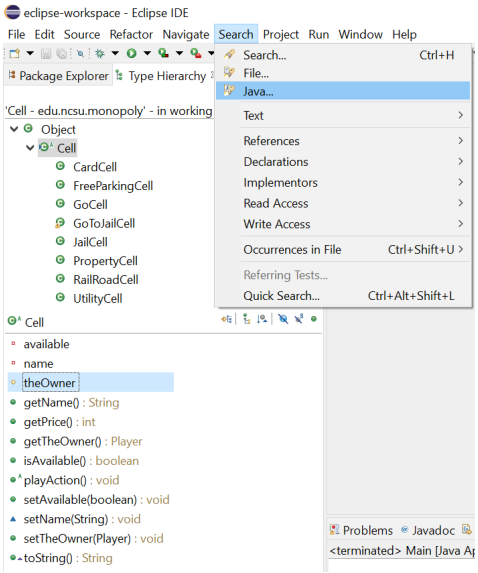

Figure 14: Search to verify changes.

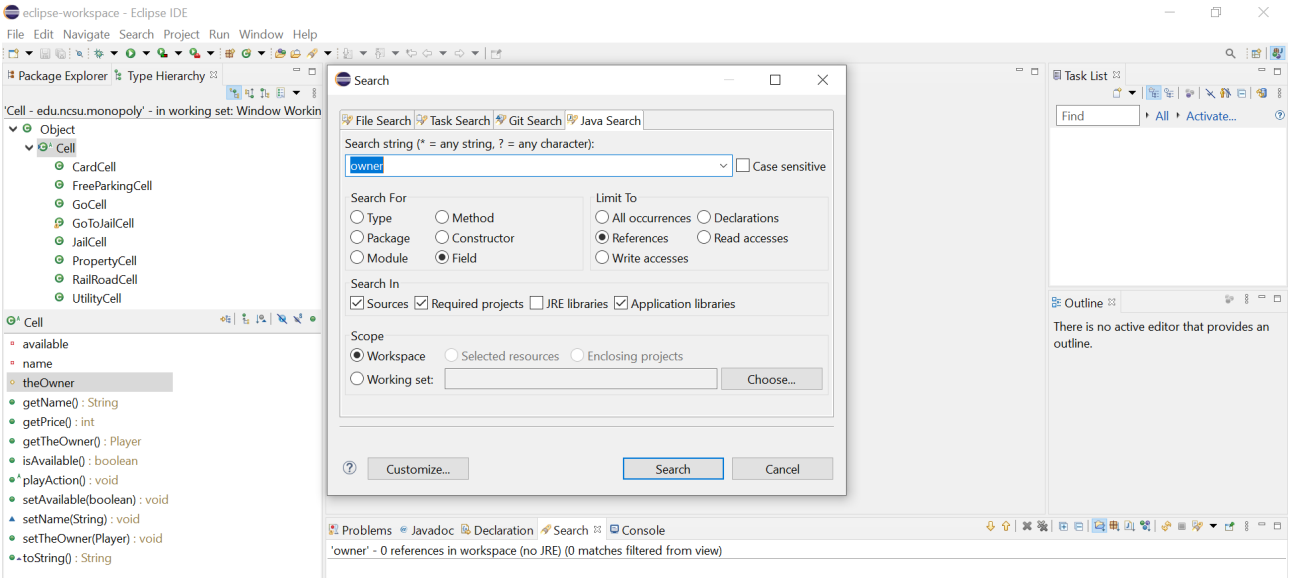

Figure 15: Verify updated “owner”.

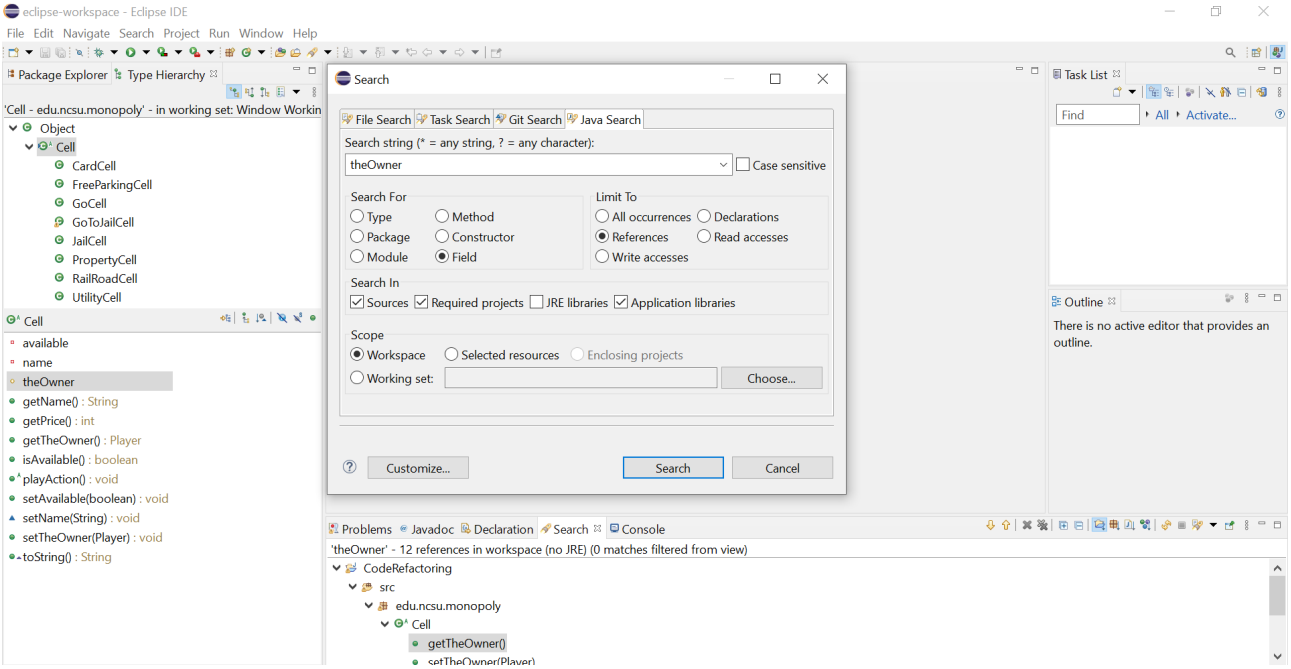

Figure 16: Verify change transitions to “theOwner”.

Browse to Cell.setTheOwner. Take note that the parameter variable is still named “owner” (not theOwner).

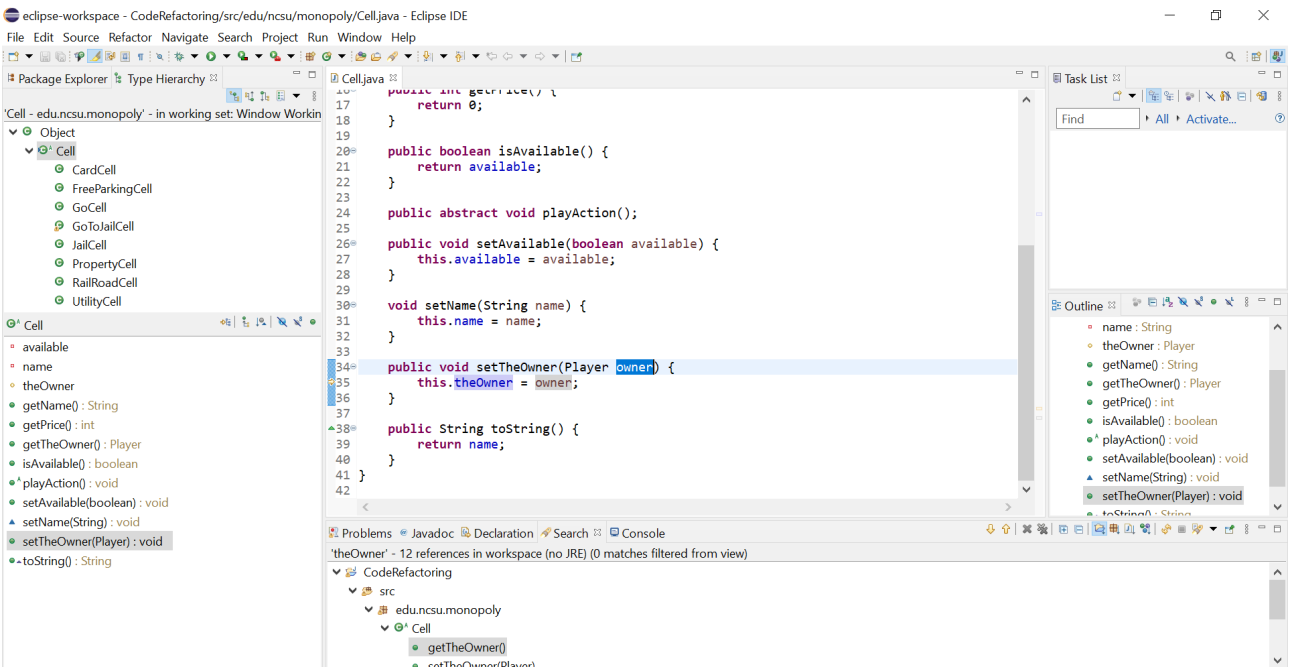

Figure 17: Cell.setTheOwner.

This implies that the refactoring just changed the name field defined in class "Cell" but there may be other variables (parameters, local variables) with this same name. Global search-and-replace method would have changed **every string**, but Eclipse's refactoring support knows the **structure** of the Java program and only does what the developer intended.

The changes made can be undone by choosing Undo from the Edit menu. Refactorings in Eclipse can always be undone. (Previously, "Undo" option for refactorings is available within "Refactor" menu).

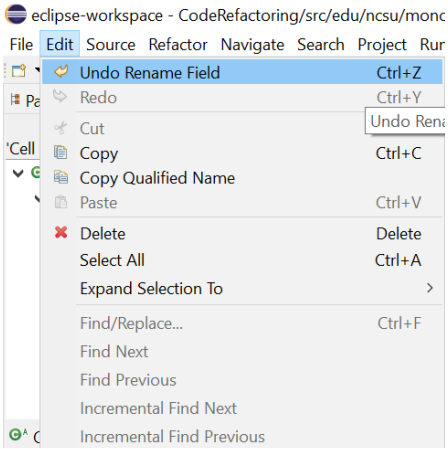

Figure 18: Undo refactoring for renaming "owner" class field.

Changing a Class Hierarchy

Continuing with class "Cell" in "...monopoly" package, choose the PushDown refactoring to move "available" field from the superclass to all of its subclasses. Are there member functions that should be pushed-down too? Think about this carefully before carrying out the refactoring. Use "Preview" to see what would be changed.

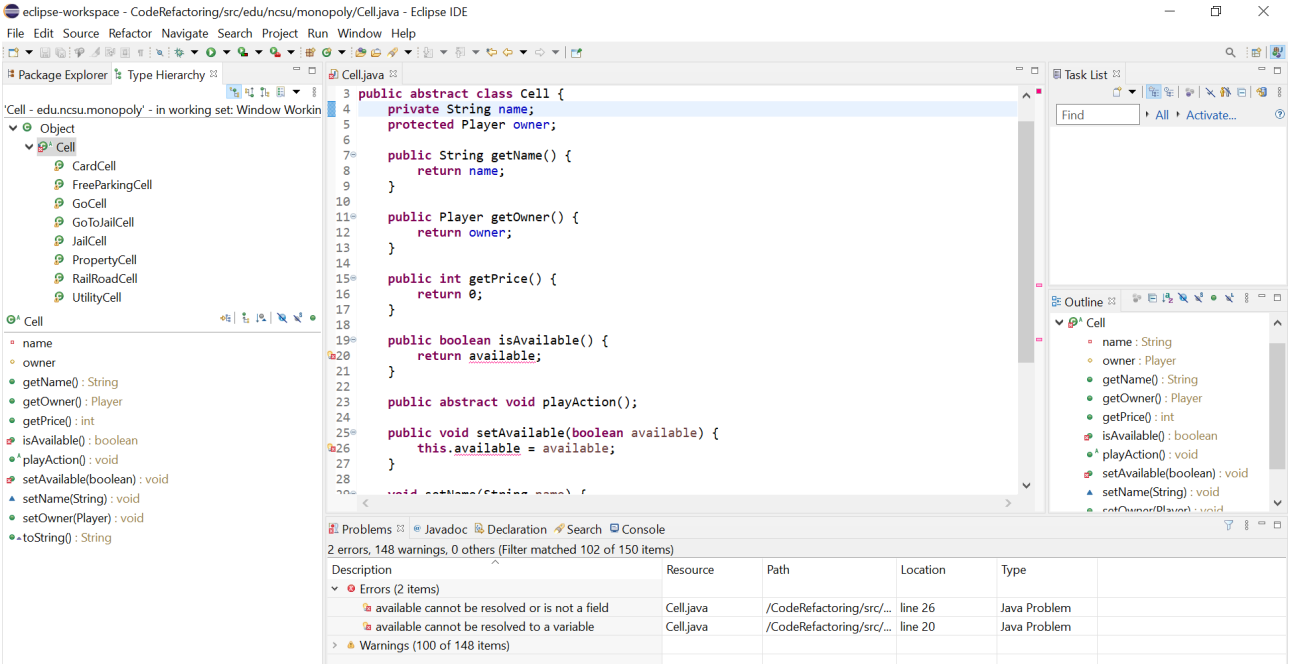

Figure 19: Apply "push-down" to "available" field.

After pushing down, some errors occur. The list of errors in the Problem view shows that setAvailable() or getAvailable() on a reference to an unfound object (pushed-down) in class Cell (the abstract class). Thus, this push-down wasn't a good idea (though it showed you how this can work). A fix can be done by choosing Edit -> Undo.

Other than "Undo", Eclipse' "push-down" refactoring can be undone by Refactor -> Pull Up; logical opposite of Push Down. Choose one of the subclasses of "Cell", say, "Card Cell". Select "available" and run the Pull Up refactoring. But, you might see there could be a problem with fixing things this way: you don't want to have to run Pull Up for all the subclasses of Cell! Go ahead and proceed, and be sure to choose the methods to pull-up that you pushed-down earlier.

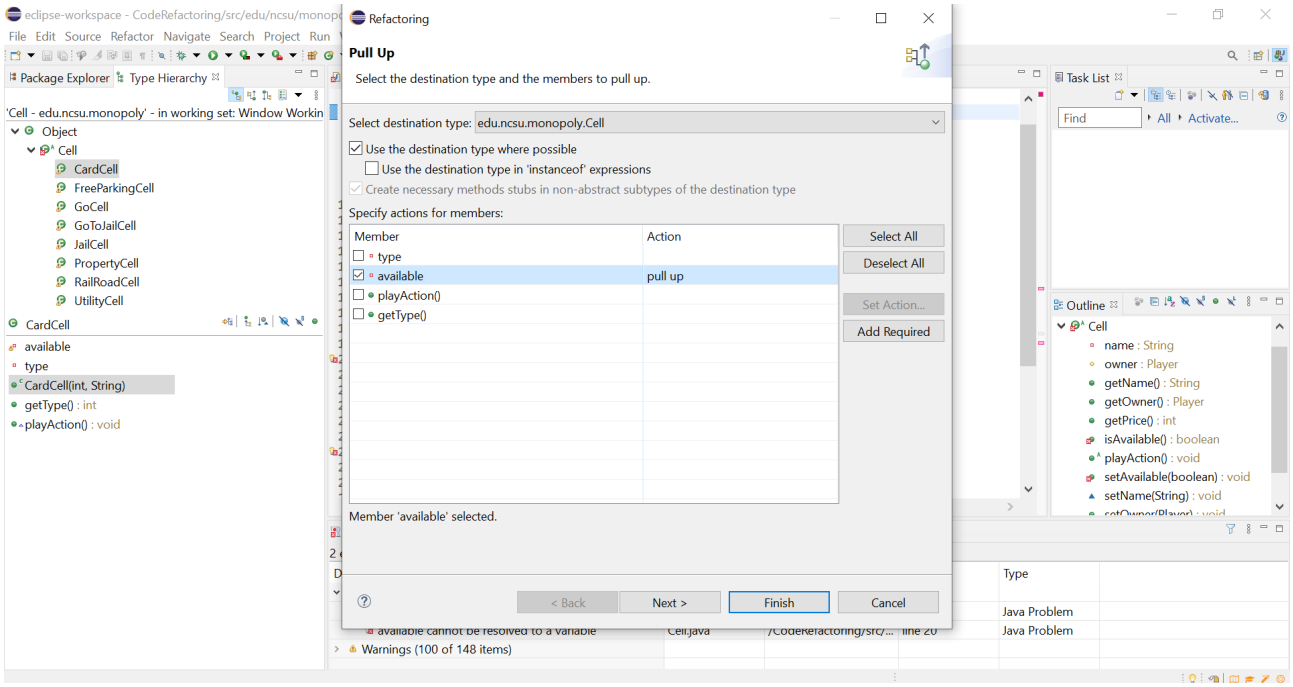

Figure 20: Apply "push-up" to "available" field in every subclasses that contain pushed-down "available" field.

After "pull-up" of "available from every subclasses in class "Cell", Eclipse recognizes this problem, and the window lets developer see how the "available" field had been declared again and again where the deletion of overlapping field will fix the issue.

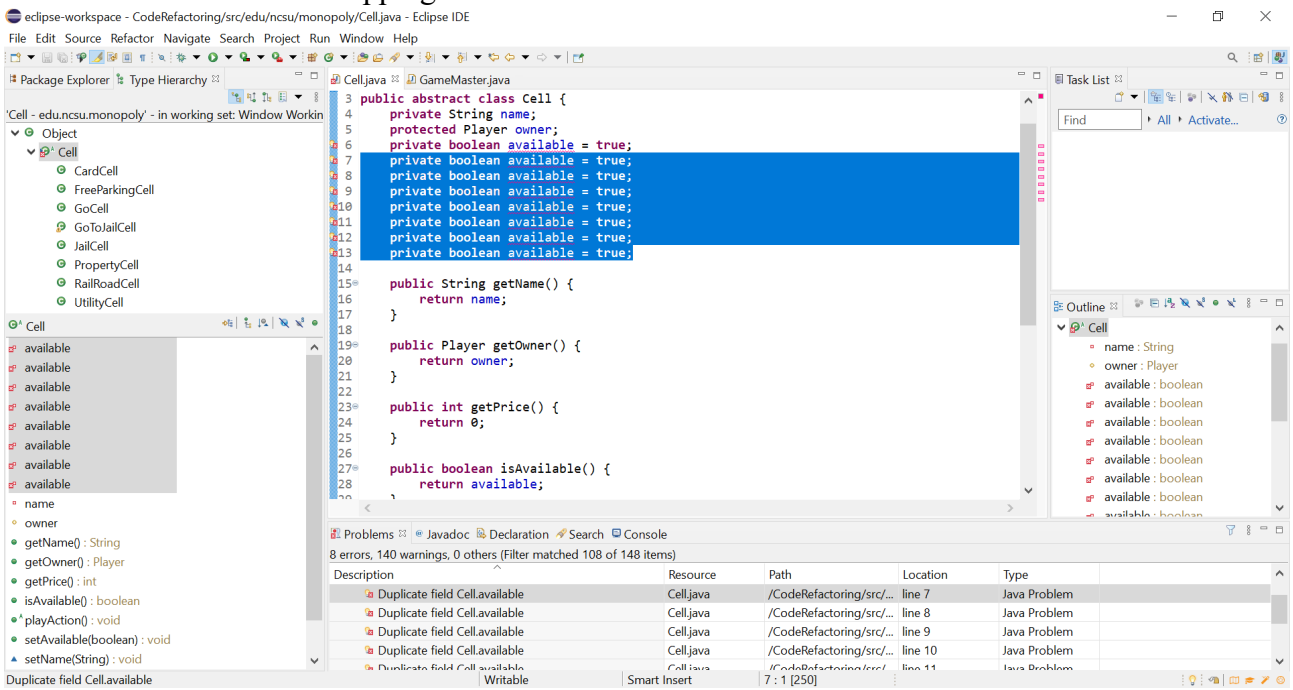

Figure 21: Delete the overlapped declaration of "available" field after pull-up.

The changes made here in the class hierarchy demonstrate how smart Eclipse can be about your Java programs and what kinds of large-scale changes it can make throughout your files. Changing the hierarchies manually without a tool would be tedious, time-consuming, and error prone.

Extracting an Interface

Abstract class “Cell” has a field called “owner” because players can own squares on a Monopoly board. What if players could own other things? Let's say developers decided to make the notion of owning something an *interface*.

Choose class “Cell” and then bring up the “Extract Interface” refactoring operation. Name the new interface “IOwnable”. What members of “Cell” should move into this interface?

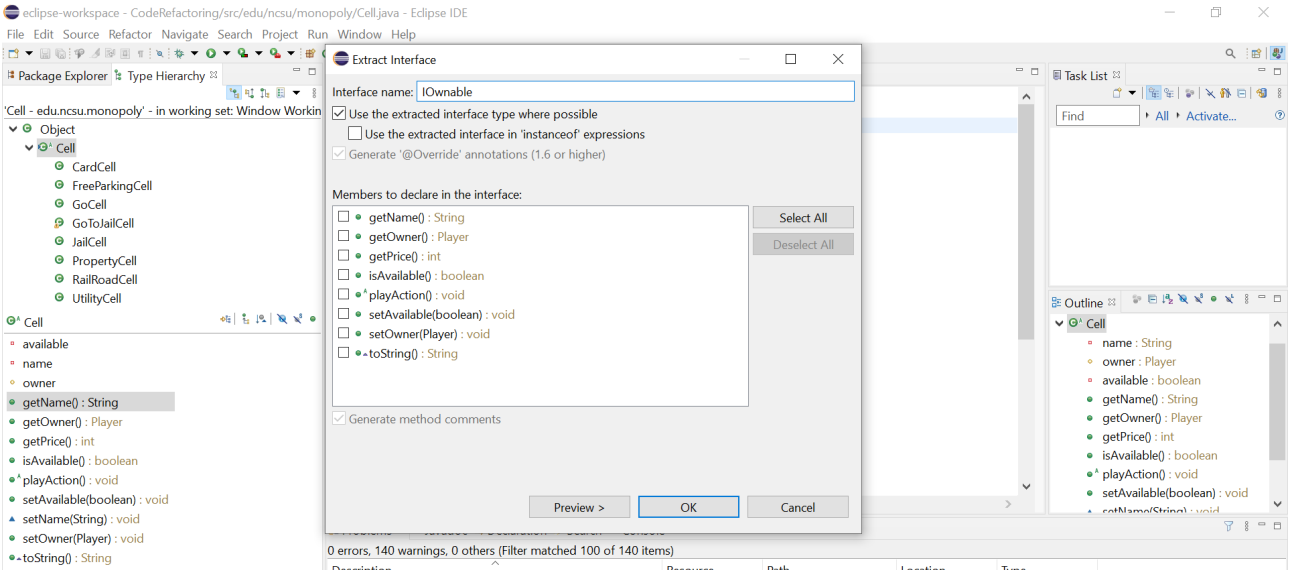

Figure 22: Delete the overlapped declaration of “available” field after pull-up.

Q: If developer chooses to select all from “Cell”, then how the codes will be changed?

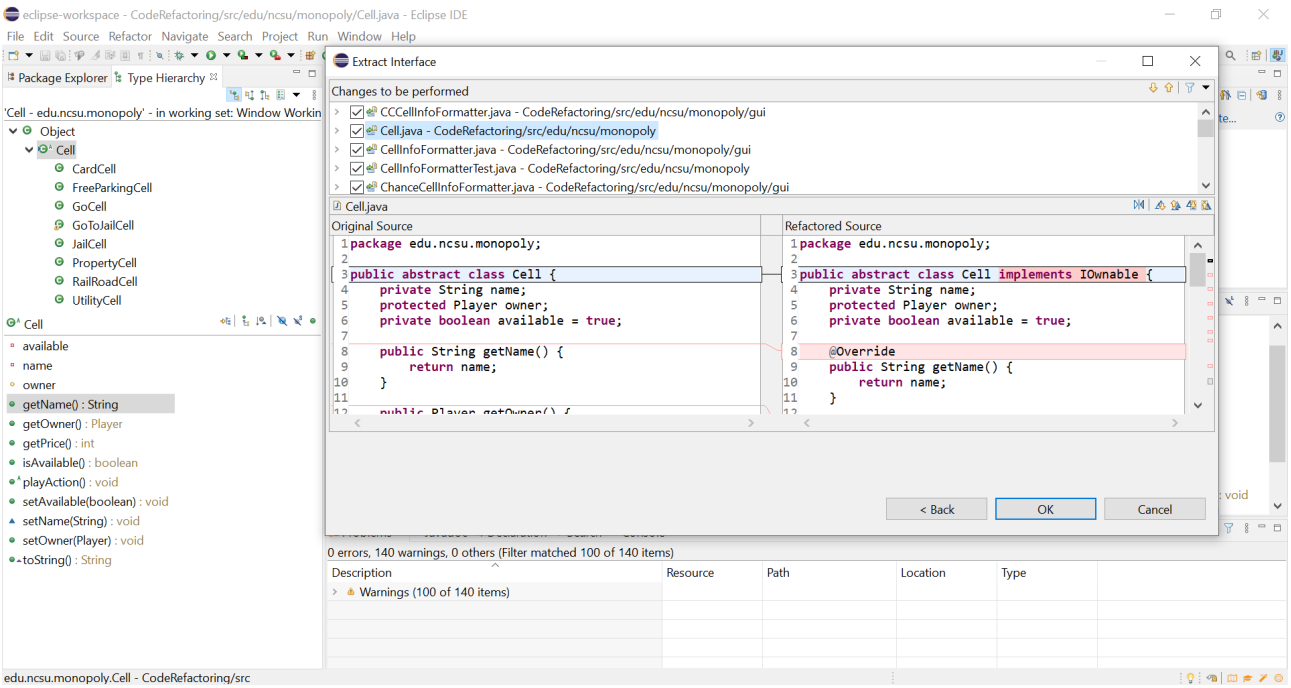

Figure 23: Check changes using “Preview”.

Extracting a Method from Code

A useful refactoring is to take a chunk of code and turn it into a method. Then it can be reused elsewhere. You can use this to remove instances of duplicated code too.There might not be a good example to be found within this project, but let's see how it works anyway.

In the class “PropertyCell” there is a method “getRent()”. Let's take the **first for-loop** and make it a separate method. Highlight the loop itself and then choose “Extract Mehod” from the refactor menu. You might name the new function something like calcMonopoliesRent(). Look at the menu that comes up and make sure you understand what the options are and why it's asking for these things.

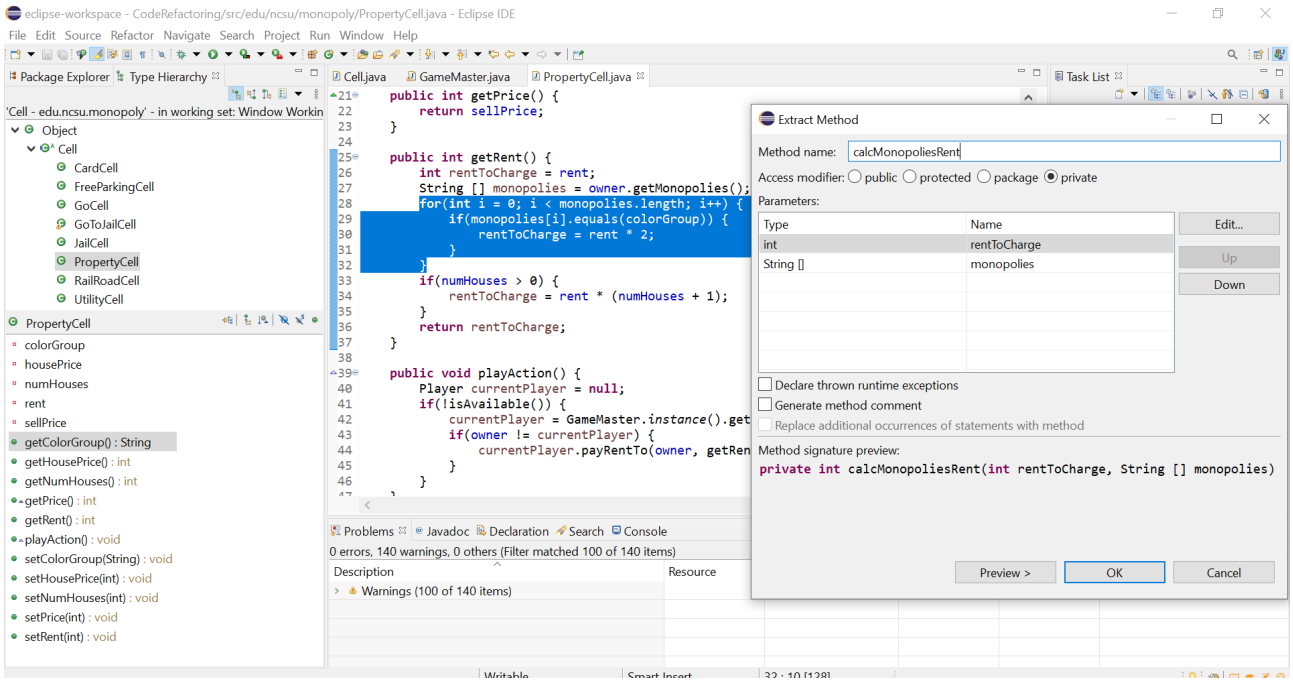

Figure 24: Method signature preview 1.

Cancel the action and go back to highlight the **loop and the declaration of the String array right before it**. Now do an “Extract Method” on that. Compare the signature of the function with previous extraction.

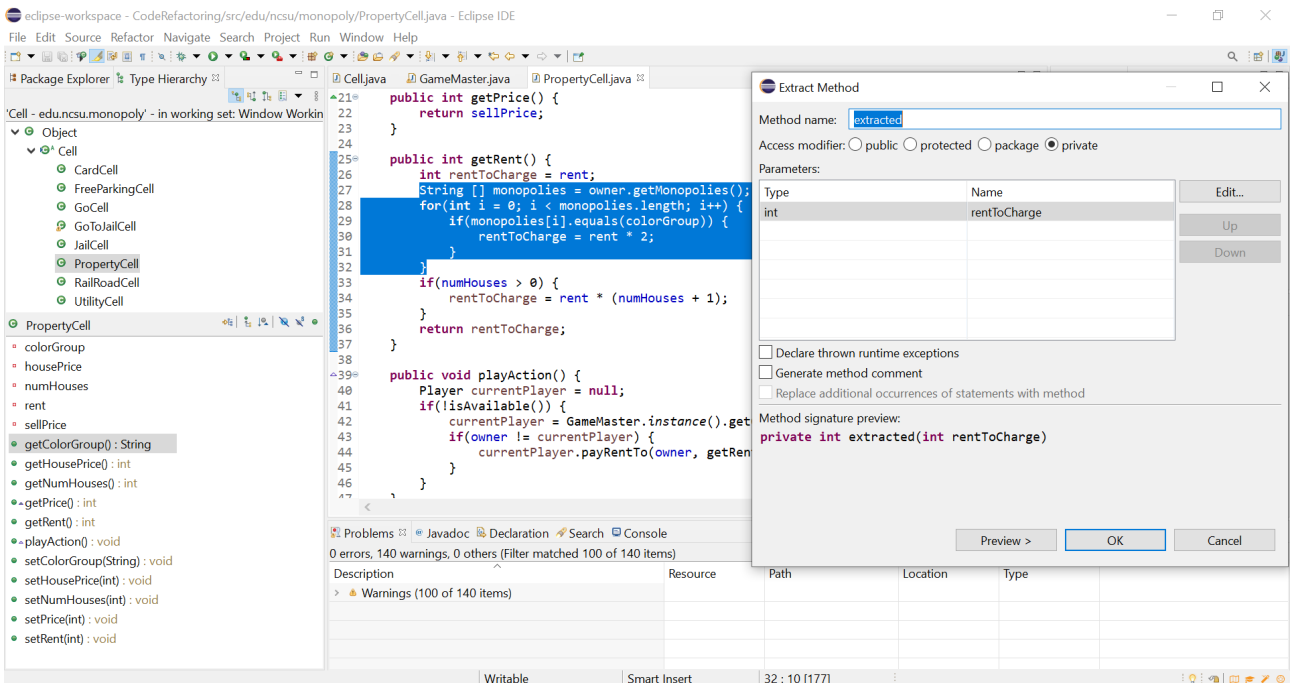

Figure 25: Method signature preview 2.

Creating a Local Variable from Repeated Code

"Extract Local Variable" refactoring allows developer to take an expression that might be repeated and create a local variable from that expresssion.

Go to "GameBoard.addCell(PropertyCell)". Note that the expression "cell.getColorGroup()" is used **twice**. Highlight one of those usages and then "Extract Local Variable" from the refactoring menu. Note that Eclipse suggests names for the local variable.

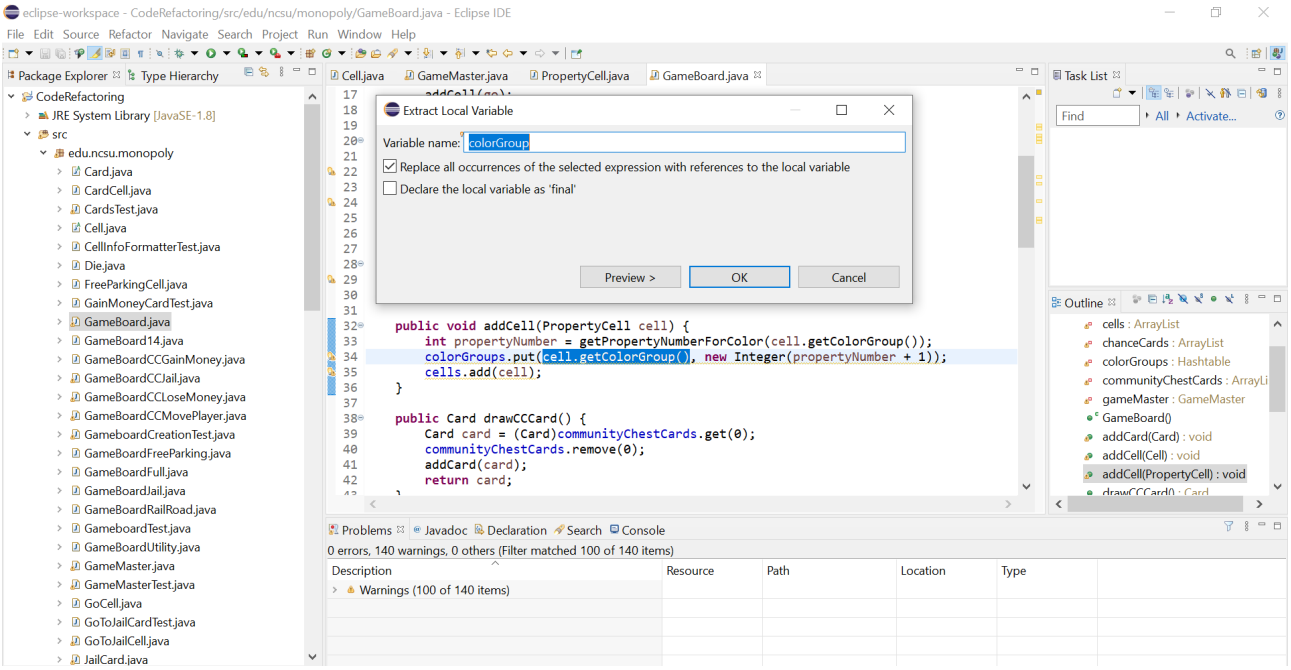

Figure 26: Extract Local Variable.

Explore what options are offered, and carry out the refactoring and make sure you understand what has changed.

Q: Is it always alright to do this to a function call like this? Could it affect the correctness of the program?

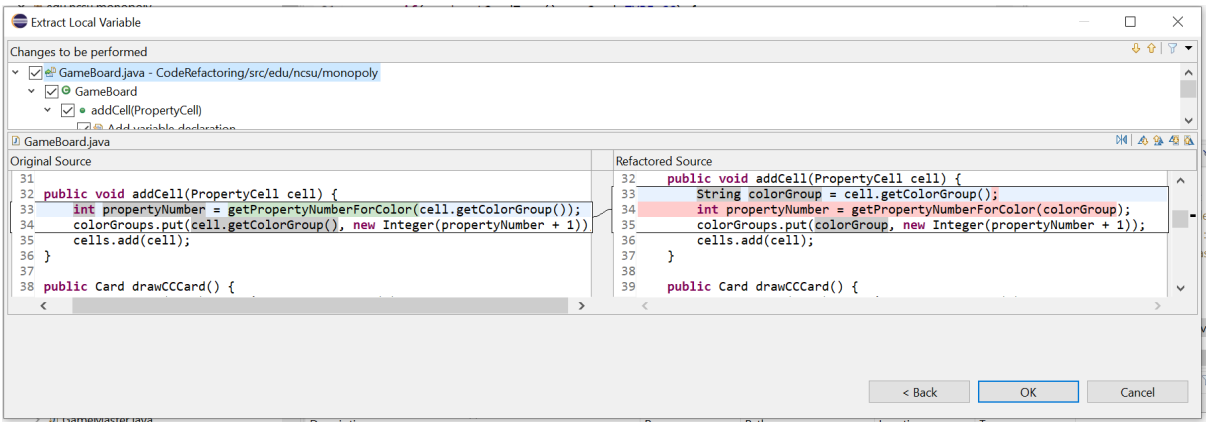

Figure 27: Method calls are replaced with local variable calls.

Changing a Method's Signature

Signature of a method can be changed but one must think carefully about doing this. Eclipse will certainly not be able to make all the logical changes that are required; you'll have to do more work after the refactoring operation is completed.

However, let's see how this would work. In class “Cell”, select the abstract “playAction()” method, and use the refactoring Change Method Signature to:

1. change the return type from void to boolean;
2. add a new parameter called msg of type String.

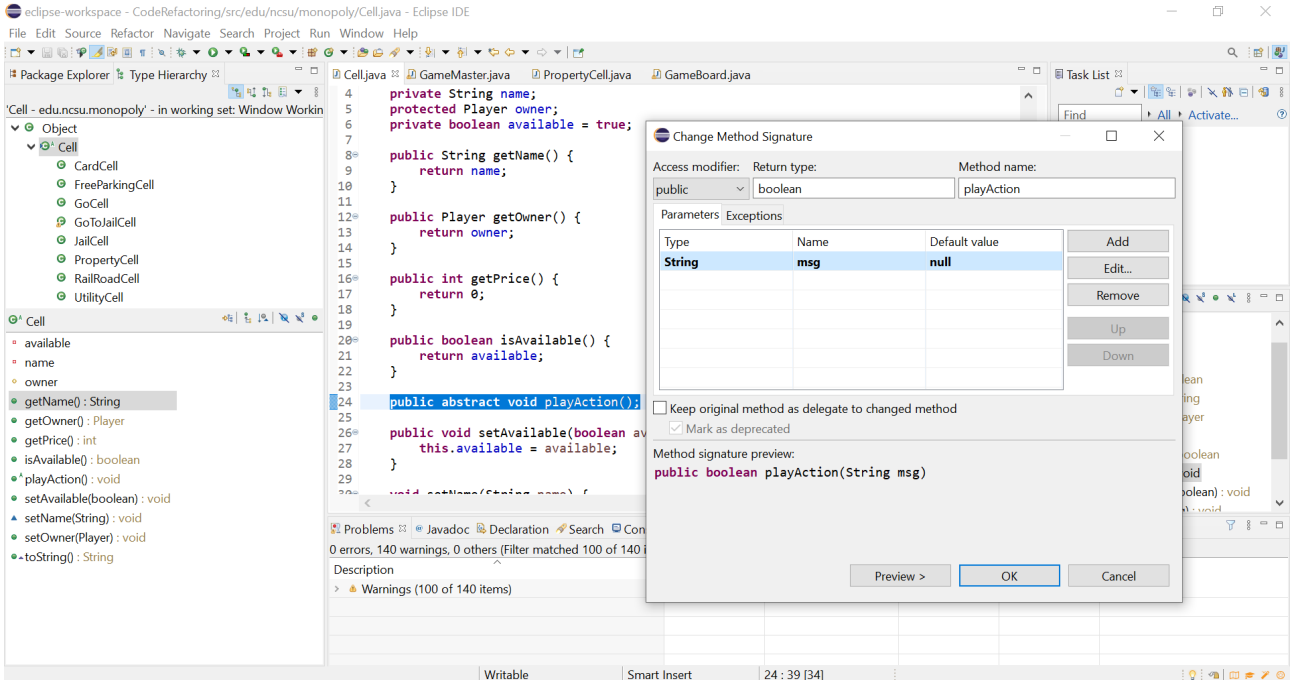

Figure 28: Specifications of change for playAction() method signature.

Use Preview to see where and how things are changing. Why are things changing in other classes besides “Cell”? How does this affect the definitions of any other classes besides “Cell”?

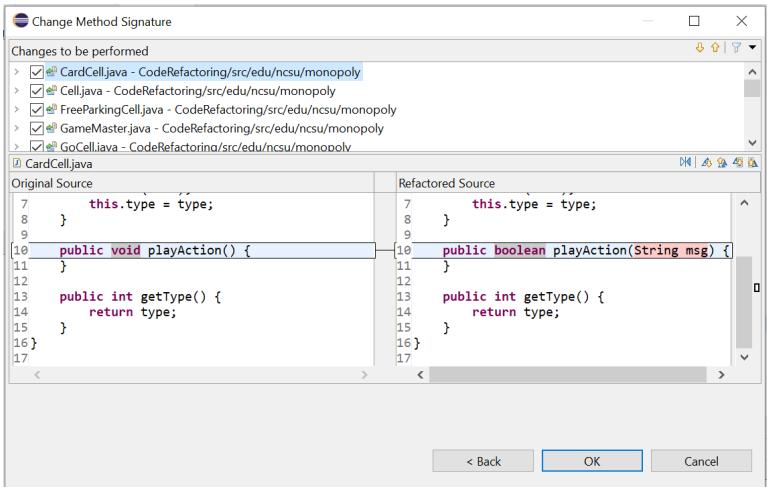

Figure 29: Other classes that calls playAction() method, need to simultaneously change the method signature.

UECS2363 SOFTWARE CONSTRUCTION AND CONFIGURATION

Summary

1. Refactoring involves making structural changes to source code without changing the external behaviours of the application/software.
2. Even simple refactorings (e.g. rename class field) would be hard to do with normal IDE features (e.g. search and replace). Refactoring without tool support is NOT PRACTICAL.
3. Eclipse' refactoring support effectively understands Java program structure which allows developer to make serious changes across many files in the project.
4. Having unit tests that show a system works allows a developer to refactor without fear of breaking the design.