System development, integration and testing lab #8

Part 1 – Eclipse refactoring functions

Getting Started

Download the Lab2A file and import it into a new Java project in Eclipse (follow the procedure from the Unit Testing lab).

Let's make sure it works, first! Right-click on the src folder and from the Run menu choose Run JUnit Test. Is all well?

Go modify a test-case to make it fail by changing something. E.g., in CardTest.java in the the method testCardType() change the last assert to test for TYPE_CC instead of TYPE_CHANCE. Re-run the unit tests and see how the failure is handled? Un-do this error before proceeding.)

Renaming a Class Field

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

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

Did this really change everything? Use the Search->Java to find all references to a field named *owner* and see if anything was left unchanged. Then search for all references to a field named *theOwner* to verify all was updated.

But note that in Cell.setTheOwner (was SetOwner) the parameter variable is still named owner (not theOwner). Your refactoring just changed the name field defined in Cell; there may be other variables (parameters, local variables) with this same name. Global search-and-replace would have changed every string, but Eclipse's refactoring support knows the structure of your Java program and only does what you want.

BTW, you can undo all of these changes by choosing Undo from the Refactor menu! You can always undo your previous refactorings in Eclipse. (The menu shows the most recent one, but if you un-do that one you can then un-do the previous one.)

Changing a Class Hierarchy

Staying in abstract class Cell, note that there is a field named *available*. Choose the PushDown refactoring to move this from the superclass to all of its subclasses. Wait -- are there member functions that should be pushed-down too? Think about this carefully before carrying out the refactoring. Also, use Preview to see what would be changed.

After carrying this out, look at some of the subclasses (seen from the Hierarchy view you got by hitting F4) and see if the field has been moved into the subclasses.

But wait! Pushing down this method caused some errors to occur! See the red error-indicator icon next to GameMaster.java and Player.java? Or, do you see the list of errors in the Problem view? If you click on each error in the Problem view, you'll see we're trying to call setAvailable() or getAvailable() on a reference to an object of type Cell (the abstract class).

So this push-down wasn't a good idea (though it showed you how this can work). We could fix our mistake by choosing Refactor -> Undo. But before you do this, let's demonstrate the power of Eclipse's refactoring by doing this in a more "manual" way.

Let's un-do this by using Refactor -> Pull Up, the 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.

Ah ha! Eclipse recognizes this problem, and the window lets you say if you want the identically-named fields and methods in *other* subclasses of Cell to also be pulled-up. To select them all, click the check-box by Cell once to clear everything, and then again to select everything.

The changes you 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. Doing these sorts of things by hand would be tedious, time-consuming, and error prone. (BTW, are you wondering if Eclipse just trashed this code? Run the unit tests again.)

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 this made sense, and we decided to make the notion of owning something an interface.

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

Carry out this refactoring and make sure you understand how the code changed. What files were updated? What new files were created? You might find using the Preview option helpful here.

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. In our current project, there may not be a good place to show that this is useful, but let's see how it works anyway.

System development, integration and testing lab #8

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. Wait -- don't carry out the extraction yet.

Instead, cancel and go back and highlight the loop and the declaration of the String array right before it. Now do an Extract Method on that. Why is the signature of the function different that previously? Make sure you see what's happening here.

Go ahead and carry out one of these two refactorings. Examine the code in this Java file to make sure you see what happened.

Creating a Local Variable from Repeated Code

The Extract Local Variable refactoring allows you take an expression that might be repeated and create a local variable from that expresssion. Let's try this.

Go toe GameBoard.addCell(PropertyCell). See 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.

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

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

Changing a Method's Signature

As the article on refactoring with Eclipse notes, you can change the signature of a method but you must think carefully about doing this. And 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.

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

change the return type from void to boolean;

add a new parameter called *msg* of type String.

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?

Wrap Up

System development, integration and testing lab #8

Talk about what you like about Eclipse support for refactoring, and what some of its strengths and limiations are.

Here are the major points that you should take away from this lab:

Refactoring involves making structural changes to source code.

Even those that seem simple (e.g. renaming a class field) would be hard to do with normal IDE features (like search and replace). Refactoring without tool support is not practical.

Eclipse's refactoring support effectively understands Java program structure to allow you to make serious changes across many files in your project.

Having unit tests that show a system works allows you to refactor without fear of breaking your design.

Part 2 – The exercise itself

(individual work)

Download the Lab2B.zip file. The file contains six different exercises. Each exercise consists of a Java class (or classes) to be refactored and a corresponding JUnit test class. For each of the exercises do the following:

1) Read the code to understand, what it does
2) Identify bad smells in the class (e.g. the CsvWriter is an example of Divergent Change).
3) Refactor the class. If necessary, correct any bugs you'll encounter.
4) Run the unit tests to ensure no bugs were introduced

The solution to today's exercise should include the refactored code, information about the bad smells you identified and refactoring techniques you applied to remove each smell.