Lunds tekniska högskola
Datavetenskap, Nov 30, 2010
Torbjörn Ekman and Görel Hedin
EDA260 Programvaruutveckling i grupp—projekt

Lab Exercise
Refactoring using Eclipse

1 Introduction

This lab will give you some hands-on experience of refactoring. During the lab, write down answers to questions, etc. Either directly in a printout of this text, or on a separate piece of paper. Be prepared to show this to your lab supervisor.

Remember what you learned in earlier labs:

- Pair program. Switch roles often. Both driver and navigator are active.
- Work in baby steps.
- Keep a list of things to do.
- Update often (just to get used to it—no one else will be working on your module)
- Commit often, and write good commit comments.

1.1 Check out module

Open Eclipse and check out the following module:

Host: `cvs.cs.lth.se`

Repository: `/local/cvs/eda260/refactorlab`

Module: `RefactorTeamXX` (where XX is a two-digit number. Ask your lab supervisor for which number to use.)

We will refactor this module in various ways to improve the design and readability, and also to learn a few common refactorings.

1.2 Study code

Take a brief look at the classes to get acquainted with the code base. We hope that the person who wrote this code left it in a clean state. Run JUnit to make sure all existing tests pass. Check the box if they did!

Green bar: ☐

1.3 Learning test

We suspect that the unknown class is actually a stack, but we are not sure. One way of finding out is to add a *learning test*, that is, a test that we write to learn something about what the code does. Add the following test that checks if inserted elements, using `put`, can be accessed in last-in-first-out order, using `get`:

```
@Test public void testMultiple() {
    Unknown unknown = new Unknown();
    unknown.put("First");
    Object second = "Second";
    unknown.put(second);
    Object third = "Third";
    unknown.put(third);
    assertSame("Objects aren't returned in LIFO order", third, unknown.get());
    assertSame("Objects aren't returned in LIFO order", second, unknown.get());
}
```

Execute the tests again. Does the unknown class behave like a stack?

Green bar: ☐

Now that we have the green bar, we should take a look at the test code and see if we can improve it by doing some refactorings. Think briefly about this, and write down a few possible refactorings:

Todo:

2 Create a fixture

The first refactoring we will do is to create a fixture. Both tests start by creating an instance of the `Unknown` class, and the call `unknown.put("First")`. We can factor this out code to a setup method. We can't do this in one single built-in refactoring, but we can do it in a number of smaller refactorings steps:

2.1 Move the fixture variable

First, refactor the `unknown` local variable in the first test using the built-in refactoring *Convert Local Variable to Field*. (You can select refactorings either in the Refactor menu or in the context menu.)

What did the *Convert Local Variable to Field* refactoring do? Write down some notes about this, or draw a diagram to illustrate what happened. (You can use the `Edit->Undo` command if you don't remember what the code looked like before the refactoring.)

Now, check that all tests still work:

Green bar: ☐

Now, we would like to refactor the second test in the same way. But we can't use the built-in refactoring now, because Eclipse will complain when a local variable is converted into an already existing field. Refactor it by editing the code manually instead.

Green bar: ☐

2.2 Create the setUp method

Now we want to create the `setUp` method. In one of the test methods, select the statements you want to extract (the initialization of `unknown` and the call to `put`, or just the call to `put` in case you already moved the initialization in the previous step). Invoke the *Extract Method* refactoring. Name the extracted method `setUp` as this is the convention for a fixture in JUnit. Make the new method public.

What did the *Extract Method* refactoring do? How was the test method you invoked the refactoring on changed? How was the other test method changed? Write down some notes or draw a diagram.

Green bar: ☐

2.3 Make the setUp method a Before method

Now, we want to make use of the `@Before` annotation, so that `setUp` is called automatically by the JUnit framework. However, there is no built-in Eclipse refactoring for this, so we manually change the code: Add the `@Before` annotation and remove the calls to `setUp` in the test methods.

Green bar: ☐

Update and commit the code.

2.4 Reflection

Reflect on what you have done. You have introduced a test fixture, using a series of small refactorings. It is often the case that the refactoring you want to do cannot be done in a single step. You need to break it down into smaller steps. Some of the steps might be supported completely or partially by tools like the Eclipse refactorings. Some refactorings might need to be carried out by editing manually. But after each small refactoring step, you should be able to run the tests and get the green bar. Write down a list of the refactorings you did, and note which were done manually and which were done with Eclipse tools.

3 Better names

We will now do some refactoring also in the production code. Remember to commit often (but only when the bar is green.)

3.1 Renaming methods

As we are fairly confident that we are dealing with a stack, some renaming refactorings are appropriate. Use *Rename* to change the names of the `put-` and `get`-methods to `push` and `pop`.

Verify that Rename updates both declarations and calls in a consistent way. How many declarations and calls are affected by these two renames?

Re-run all tests.

Green bar: ☐

3.2 Renaming classes

The method names are better now, but the class name `Unknown` is still bad. Use the *Rename* refactoring to change its name to `Stack`. List the changes to the code that are done by this Rename.

Re-run tests to ensure that the name change didn't affect expected behaviour.

Green bar: ☐

Rename is not aware of our code convention for naming Test classes. Rename also the `UnknownTest` class and the field `unknown` to something more appropriate. Run the tests again.

Green bar: ☐

3.3 Move classes between packages

We would like to refactor the code now so that production classes and test classes are kept in separate packages. Create a new package `datastructures` and move the class `Stack` to that package. How did this refactoring affect the test class?

Run the tests again.

Green bar: ☐

3.4 Todo list

Is there anything more on your todo-list for cleaning up the code? Perform those changes, using the built-in refactorings when possible. What changes did you do?

Run the tests again.

Green bar: ☐

Now, you have clean nice code and all tests run. Do an update, and then commit the code to the repository. (Hopefully, you have already done so several times.)

4 Further experiments

You will now do some experiments to find out how a few more refactorings work. Since this is just experimental coding, there is no need to commit to the repository.

4.1 Extracted code uses local variables

Suppose you have the following method where the code uses a local variable.

```
public void aMethod(){
     Stack s = new Stack();
     s.push(3);
     s.push(4);
     s.push(5);
}
```

Use Extract Method to extract a couple of the `push` statements. How is the local variable `s` handled?

Now, add the following statements to `aMethod`.

```
Object result = s.pop();
s.push(result);
```

The first new statement assigns to the variable `result`. Use Extract Method on that statement. How are the local variables `result` and `s` handled?

4.2 Change Method Signature

Add a new parameter to the `push` method using the refactoring *Change Method Signature*. (This is definitely a thing you should not commit :-) Make the new parameter an integer with the default value 7. How does the new parameter affect existing calls?

4.3 Experiment more

Select three additional refactorings that you have not tried out yet (see the appendix). Construct small examples and try out the refactorings.

Refactoring 1:

Notes:

Refactoring 2:

Notes:

Refactoring 3:

Notes:

4.4 Reflect

Make sure you understand what the word "refactoring" means. Write down a definition of it here:

Refactorings in Eclipse

This is a description of some of the refactorings currently supported by Eclipse.

Rename The Rename refactoring will rename a class, interface, field, local variable, method parameter, or method. The declaration is changed and all uses within the project are changed to reference the new name.

Move Move a class from one package to another, move a method to another class, move a field to another class. References are updated accordingly, with the exception of references to classes moved from the anonymous package. When using the unnamed anonymous package references must be updated manually.

Change Method Signature Change the return type, access modifier, parameters, or name of the selected method.

Push Down Move a method, field, inner class to a subclass.

Pull Up Move a method, field, inner class to a superclass.

Extract Interface Create a new interface from a subset of the memebers in the current class.

Inline May be used to reverse operations like extract method, extract constant, and similar operations. Inline method will replace each call to that method with the method body.

Extract Method The selected lines of code will be extracted to a new method. A method invokation will be added to the location where the lines were extraced from.

Extract Local Variable Extract an expression to a local variable and optionally replace each occurance of the expression within the method to that local variable.

Extract Constant Extract the selected constant expression to a constant field. Optionally, all occurances of that expression may be replaced by the constant field.

Convert Local Variable to Field Turn a local variable into an field attribute.

Encapsulate Field Generate getter and setter methods for the selected field. The field will be accessed through its getter and setter when used from other classes. The class where the field is declared can optionally use the getter and setter as well.

Convert Anonymous to Nested Convert an anonymous class to an nested inner class.

Extract Interface Create an interface by selecting members in a class to specify the interface. The class will implement the interface and one may choose to change references to access the interface where possible.

Use Supertype where possible Change each use of the selected type with a super type where possible without violating the type rules. This may be used to use the super class instead of the selected class after pulling up a method or field to its super class.