ECE155: Engineering Design with Embedded Systems Winter 2013

Lecture 26 — March 18, 2013

Patrick Lam *version 1*

Objective. After this lecture, you will be able to:

- explain the goals of refactoring;
- given a particular code change, identify which refactoring (if any) was applied;
- carry out simple refactorings on code that we provide.

Refactoring

We've repeatedly mentioned the existence of refactoring, especially when talking about Agile methods. *Refactoring* incrementally improves code quality using local, semantics-preserving changes to the code.

The reference for refactoring is the following book:

Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, 1999.

Let's unpack the definitions of the terms we used to define refactoring.

- **Local**: Refactoring should not affect unrelated parts of the program; it should only change the parts of the program that you're refactoring. When you commit a refactoring to the repository, you should be sure to only commit the source files that got refactored.
- **Semantics-preserving**: This means that the behaviour of the refactored code should be identical to the behaviour of the original code. If some tool did the refactoring, you can usually be sure that it was semantics-preserving. Otherwise, your unit tests will help ensure that you didn't change anything.

Types of Refactoring. The Fowler book classifies the refactorings into the following broad categories:

- Composing methods: change what code belongs to which method;
- Moving features between objects: put fields, methods, classes in different places;
- Organizing data: change how data gets stored in your program;
- Simplifying conditionals: make your `if` statements easier to understand;
- Making method calls simpler: change how methods relate to each other;
- Generalizations: like moving features between objects, but related to the inheritance hierarchy.

We'll continue by looking at two examples of refactorings this time. Next time, we'll look at a list of refactorings and talk about refactorings in more generality.

Example: Extract Method. A popular example of a refactoring is the "extract method" refactoring, which splits out part of a method into an independent method. For instance[1]:

```
// (1) make sure the code only runs on mac os x
boolean mrjVersionExists = System.getProperty("mrj.version") != null;
boolean osNameExists = System.getProperty("os.name").startsWith("Mac_OS");

if ( !mrjVersionExists || !osNameExists)
{
  System.err.println("Not_running_on_a_Mac_OS_X_system.");
  System.exit(1);
}

// (2, 3) do all the preferences stuff, and get the default color
preferences = Preferences.userNodeForPackage(this.getClass());
int r = preferences.getInt(CURTAIN_R, 0);
int g = preferences.getInt(CURTAIN_G, 0);
int b = preferences.getInt(CURTAIN_B, 0);
int a = preferences.getInt(CURTAIN_A, 255);
currentColor = new Color(r,g,b,a);
```

We can instead split this into three sub-methods, which each make more sense as an independent unit rather than agglomerated into one method:

```
dieIfNotRunningOnMacOsX();
connectToPreferences();
getDefaultColor();
```

This is particularly useful when the code does the same thing many times ("code clones"); you can replace all of the clones with a call to a single method, which means that you only need to change the code once.

A method should have unity of purpose: it should do one conceptual thing. A good test is that you should be able to come up with a meaningful name for the method. Note that we are putting some design into the code through this refactoring.

Although the example here replaces all of the code with method calls, it's also common to pick out just part of a method and make a separate function from that, while leaving the rest of the method in place. Many IDEs support this sort of refactoring with one click.

Another example: Replace Magic Number With Symbolic Constant. Here's another example of a refactoring, which is quite applicable to your labs. Using symbolic constants is often better coding practice than putting numbers directly into the code. Consider this example[2]:

```
double potentialEnergy(double mass, double height) {
        return mass * height * 9.81;
}
```

You can refactor this to:

```
double potentialEnergy(double mass, double height) {
        return mass * GRAVITATIONAL_CONSTANT * height;
}
static final double GRAVITATIONAL_CONSTANT = 9.81;
```

There are two benefits to this refactoring: 1) the refactored code is easier to understand; and 2) if you might need to change the value of the constant, it's easier to do so in one place only.

[1] `http://www.devdaily.com/java/refactoring-extract-method-java-example`, accessed March 20 '11.

[2] `http://www.refactoring.com/catalog/replaceMagicNumberWithSymbolicConstant.html`, accessed July 4 '11.