Inf2C tutorial SE2: Writing Good Code

1 Before the tutorial

Read through this tutorial sheet.

1. Can you find code in your project (the one you're doing the assignment on) that could be improved following suggestions here or in the lectures? Or, can you find a piece of code that's hard to understand, even if you can't see how it could be improved?
2. Go back to a piece of Java code you've written in Inf1 or Inf2. Can it be improved? How?

Take with you to the tutorial one or more printed pages of code with things you think could be improved highlighted. Be prepared to explain to the group what you think the problem is, and if possible, to suggest improvements.

2 Code inspection checklist

In a *code review*, a group of people study a section of code looking for possible problems with it – ways in which it might not be doing what it should now, or ways in which it might be hard to maintain in future. A few of the things a code review will look for are:

1. A one- or two-sentence description of each public method and class, possibly in Javadoc. (If it's not possible to summarise the functionality in one or two sentences, the method or class may be doing several unrelated things and may need to be split.)
2. Bad smells in the code (see next section)
3. Bad names (of classes, methods, attributes etc.), including:
   - names that don't explain what the thing is, e.g. `c`
   - names that don't adhere to coding standards (e.g. universal convention in Java: all classes start with a capital letter e.g. `Customer`, all instance variables and methods with lower case, e.g. `balance`, `getBalance`);
   - names that include type names, e.g. `customerArray` (exercise: why?). More subtly, `order.add(item)` is better than `order.addItem(item)`: if you trust people to name *their* variables meaningfully, you can avoid redundancy in *your* names.
4. Off-by-one errors in `for` loops.
5. Objects compared using `==` instead of `equals` (the latter is almost always what's wanted).

6. Possible dereferencing of null pointers.

7. Exceptions: are they all properly handled, not by blank catch sections that should have something in them?

8. Resource problems: e.g. are the acquisition and release of locks, database handles etc. correct?

9. Unit tests: are all present that should be? Are they correct?

3 Bad smells in code

Sometimes code "smells bad" – there's something about it that will make an experienced developer suspect that it's of poor quality, even before looking at what it's supposed to do. Kent Beck and Martin Fowler have identified a collection of "bad smells" in code. Here are a few of them.

Comments Comments at the beginning of a method that specify what a method does, e.g. in Javadoc, are not a bad smell, but a method which is densely commented inside its code is suspicious. Often it's a sign that the code is hard to read – could it be improved? A block of comment explaining what the next section of the method does often indicates a section of code that would be better separated out into its own method (with the comments then becoming the (Javadoc) specification of that method. A comment that explains what should be true at a point in the code should be replaced with a Java *assertion*. Good uses for comments (within reason) include noting when you're not sure whether to do something one way or another way, or explaining why you've decided to do it this way instead of some other way that might have seemed more obvious.

Long method Methods in a good object-oriented program almost always fit onto one screen: many methods will be just a few lines. This makes the code easier to understand and reuse, *provided that* you use well-chosen method names, which explain what the method does. If a method is long, look for ways to simplify and restructure it, usually by separating out part of the code (the body of a loop? an if or else clause? a not-very-closely-connected chunk?) into its own method. This smell often goes with the one above. E.g., if the code of a long method has a comment "Next we wizzle the froboz", it's probably best to separate out the next chunk of code into a separate (private) method, maybe called `wibbleFroboz`: then calling the method replaces the comment. It's OK if `wibbleFroboz` is only called this once, but if you pick meaningful chunks of functionality to separate out in this way, you often will find that they're needed again later.

Long parameter list It's hard to remember a long list of parameters to a method and what order they go in. Using global variables is worse, of course (why?) – but think twice about passing in something that the method could compute. E.g., don't pass in two parameters which will be got from the same object: pass the object in instead, and let the method get both pieces of data when it needs them.

Duplicated code Or, just as common and more difficult to deal with, *nearly* duplicated code. Is there a reason for the differences? (Or is it a mistake that they're not exactly the same?) Can you replace the nearly duplicated code by a private method, maybe with a parameter to account for the variants?

Large class Look at the largest class in your system – by any metric, e.g. most lines of code, most instance variables, most methods. Is it coherent, or would it be clearer to split it into more than one class? Is there code duplication?

Switch statements Java has a switch statement, but it's almost always the wrong way to solve the problem. Could polymorphism do the job instead?

Speculative generality E.g., methods that don't actually do anything but are placeholders for maybe doing something in future, or parameters that aren't yet used. They add complication for no value: if they ever are needed, it's easy enough to add them then. Keep it simple. You Ain't Gonna Need It.

4 Refactoring: improving existing code

Refactoring code is improving its quality without changing its functionality. It includes minor changes e.g. to improve variable names, and more major changes to restructure the code and improve its design. It can remove bad smells in code. Some refactorings are one-offs, but many are common, e.g., factoring out a chunk of a long method. These can be described systematically, and ultimately automated, at least to some extent: that is, a tool (e.g. Eclipse) can give the programmer an easy way to choose a refactoring to apply, and can then automate the actual edit of the code. See `http://www.refactoring.com` for more information.

Moral: *if* you keep your code clean, readable and well-designed at all times, *then* it's easy to change the functionality when you need to. And *then* you don't need to put things in now in case you need them later. And *then* it's easier to keep your code clean, etc....