Refactoring in a nutshell

Object-Oriented Programming and Design B2-2011
Department of Computer Science at the University of Copenhagen (DIKU)

Refactoring is a process of revising your code before you submit it as *done*. Many people get overexcited when their code finally "just works" (you'll have this a lot as you learn the art of programming), and move on to the next problem. What many don't realise is that in the real world their code is going to be maintained by other programmers, or, if they should be so unlucky, themselves[1].

The intent of refactoring is to make code more readable. While an implementation that works may be good, an implementation that works and can be read by a human being is even better, since most likely doesn't work exactly as expected, in which someone will have to do a bugfix.

Ideally, you should refactor your code every time before submitting an assignment. We're certain that most of your instructor's comments could be avoided if you just refactored your code to be *readable*.

This material covers both some basic concepts in refactoring, as well as the conventions we'd like you to follow from now on in this class. You might be familiar with another convention, or you might in the future find yourself in an environment where other conventions are applied. This is by no means a silver bullet, but it is an exercise in adjusting one self to a code-base-wide set of conventions that let's all the programmers easily get an overview of the system, regardless of their background with the system.

1 Naming and commenting

In this class we use a very explicit style of naming and a very conservative style of commenting. This means that various program constructs are named as logically consistently as possible while comments are only written if they *absolutely* have to be there. That is, there is seemingly no easy modularisation and/or naming strategy that can let you avoid the comment.

The reason we chose this strategy over vicious commenting is that names of fields, classes, methods, etc. are easy to change if they are not descriptive enough. Indeed, a good editor provides you with the functionality to rename an element and simultaneously change the name everywhere else it occurs[2].

Comments on the other hand, are hard to change. They are written in human-readable language and have no formal structure. Your editor has no chance of helping you out here. As a result, comments often outlive the implementation they describe, since programmers are:

1. Too lazy to revise comments.
2. Too scared to delete comments since they probably are *very* important.

A comment that has outlived the implementation it describes is, literally speaking, pointless.

The other reason for this strategy is that we would rather have that you code in such a way that you understand every single character of your code and don't have to resort to such a boring thing as explaining it to yourself and others through unreliable human-readable comments. In other words, code doesn't lie, comments might.

In general, seek to make your method, class and field *signatures* clear, rather than the comments around them, and use comments only to make the hidden things clear. Indeed, a well-formed signature doesn't need a comment.

As such, *we disallow use of any abbreviations what-so-ever*, except the most obvious ones, such as API, IO, GUI, etc. This is because abbreviations can often be ambiguous. We want specificity, not ambiguity. We especially discourage something like Hungarian Notation (Google it if you must). There's absolutely no reason to use it (given the initial goals of the notation) in a language like Java.

[1] An active programmer can look back at his own code 6 months later and think "what in the world was I thinking?"

[2] For Eclipse, see http://showmedo.com/videotutorials/video?name=IntroductionToEclipseWithJava3_JohnM&fromSeriesID=6.

This leads inexplicably to very long names for various constructs. If you're using a good editor this is not a problem as you have auto-completion[3]. Besides, after reading the long name a few times, you might come up with a shorter equivalent, in that case – rename it, immediately. As a last resort, you can keep the name short if you provide additional commentary.

We would like you to write all the comments and names in English. This is because Java has a lot of English keywords, and blending languages is odd. Besides, there's no guarantee that the person that will maintain your code will be a native Danish speaker.

Don't try to be funny when coming up with names, it's only the better comedians that can come with lasting jokes.

Use the following conventions for the various types of methods:

Accessor methods Should be prefixed with `get`, e.g. `getSize()`, unless they return a value of type `boolean` , in which case they should be perfixed by `is`, e.g. `isEmpty()`.

Mutator methods Should be prefixed with `set`, unless the method does more than change a single instance variable. For instance, recall the method `advance()` from assignment 1, it changed an instance variable representing the position of a cow in the race, and possibly also another instance variable representing the winner of the race, if a winner was determined as a result of advancing some cow.

In addition to the above comments, we'd like you to make use of the `camelCase` naming convention. This is implies the following naming strategy for the respective Java constructs:

- `someSortOfField`
- `SomeSortOfClass`
- `someSortOfMethod`
- `someSortOfVariable`
- `SOME_SORT_OF_CONSTANT`

2 Code modularization

1. Keep the length of your code lines under 80 characters. It's far more preferable for you to break your code into more lines than to have long lines which do a lot on a single line. Instead, use auxiliary variables to aid readability, and don't worry, the compiler (should) compile them away for you.

2. If your method is longer than 5 lines, consider splitting it into helper methods. Indeed, one often feels the temptation to vertically separate the "stages" of a method, and add comments in between them – this is *exactly* when you should define a few helper methods with a meaningful names, and let the body of the method just be those stages. Again, if you have a good editor, it will help you out[4].

3. A method should do one thing only – exactly what it's name implies. If these two don't match, either change the name or change the method.

[3]For Eclipse, see `http://showmedo.com/videotutorials/video?name=IntroductionToEclipseWithJava2_JohnM&fromSeriesID=6`.
[4]For Eclipse, see `http://www.youtube.com/watch?v=7KDruqCzdpc`.

3 Indentation and bracketing

1. Use indentation of size 4.
2. Use spaces instead of tabs[5]
3. Place `{` and `}` on separate lines, as you see done throughout the code snippets in this document.
4. Always place `{` and `}` around the body of an if statement or a loop, even though they are optional for one-liner code blocks. This is because you often get overexcited that you could do something with a one-liner only to later realize that you forgot to do something else in the body of the if statement or loop. Adding another code line at the same indention level does not have the effect that you'd expect. Indeed, only the first line is taken as the code block for the if-statement/loop, and the second line is executed after the if-statement/loop.

4 Static, instance and local

The following rules are used to seek clarity about what type of variables and methods are used in various contexts.

1. *Always* prefix instance methods and variables with `this`.
2. *Always* prefix static methods and variables with the name of the class in which they are defined. This convention is in place because static methods are not *dynamically dispatched* the way that instance methods. Ask your instructor for an explanation if you're not sure what this means.
3. Hence, local variables are not (nor can they be) prefixed by anything.

5 Instance variables

Always instantiate your instance variables in the constructor, and not when you define them.

6 Setup in Eclipse

We've attached a settings file as `oopd-2011.xml` for you to use in your Eclipse installation if you want to adhere to these conventions painlessly. You import the setup by navigating to `Window -> Preferences -> Java -> Code Style -> Formatter -> Import`.

7 Other tips and tricks

7.1 `boolean` condition is a `boolean` value

If you have a method that returns a `boolean` and ends with an if-statement that either returns `true` if the boolean condition holds or `false` otherwise, then you can spare the if-statement.

```
boolean checkTheVariable(boolean variableIsTrue)
{
  if (vairableIsTrue)
  {
    return true;
  }
  else
```

[5]This ensures that when you port your code to another computer all indentation remains the same and is not dependent on what the editor (on that machine) is defined to.

```
  {
    return false;
  }
}
```

Can be written shorter as:

```
boolean checkTheVariable(boolean variableIsTrue)
{
  return variableIsTrue;
}
```

7.2 `null` value is `null`

Much like the above case:

```
public Cow winner()
{
  if (this.winner == null)
  {
    return null;
  }
  else
  {
    return this.winner;
  }
}
```

Can be shortened down to:

```
public Cow winner()
{
  return this.winner;
}
```

7.3 `return` stops the method

If you have the code sequence

```
public Cow getCowAtIndex(int index)
{
  if (index == 0)
  {
    return this.firstCow;
  }
  else if (index == 1)
  {
    return this.secondCow;
  }
  else if (index == 2)
  {
    return this.thirdCow;
  }
  else if (index == 3)
  {
    return this.fourthCow;
  }
  return null;
}
```

You do not need the `else` part as return already stops the method, and no further work is done! So you can shorten the code to the following:

```java
public Cow getCowAtIndex(int index)
{
  if (index == 0)
  {
    return this.firstCow;
  }
  if (index == 1)
  {
    return this.secondCow;
  }
  if (index == 2)
  {
    return this.thirdCow;
  }
  if (index == 3)
  {
    return this.fourthCow;
  }
  return null;
}
```

P.S. Yes, we forgot to document why we return `null`!

7.4 Empty methods

If they have to be there (usually they don't), place a comment explaining why.

7.5 Auto-generated code and comments

Delete it all!