Construction:
High quality code for 'programming in the large'

Paul Jackson

School of Informatics
University of Edinburgh

What is high quality code?

High quality code does what it is supposed to do...

... and will not have to be thrown away when that changes.

Obviously intimately connected with requirement engineering and design: but today let's concentrate on the code itself.

What has this to do with programming in the large?

Why is the quality of code more important on a large project than on a small one?

Fundamentally because other people will have to read and modify your code – even you in a year's time count as "other people"! E.g.,

- because of staff movement
- for code reviews
- for debugging following testing
- for maintenance

(Exercise. Dig out your early Java exercises from Inf1. Criticise your code. Rewrite it better *without changing functionality*.)

How to write good code

- Follow your organisation's coding standards - placement of curly brackets, indenting, variable and method naming...

Coding standard example

- Suppose you are used to...

```
public Double getVolumeAsMicrolitres() {
      if (m_volumeType.equals(VolumeType.Millilitres))
            return m_volume * 1000;
      return m_volume;
}
```

- ... and you see

```
public Double getVolumeAsMicrolitres()
  {
    if(m_volumeType.equals( VolumeType.Millilitres))
      {
        return m_volume*1000;
      }
    return m_volume;
  }
```

- Even worse: mixed styles in one file - inevitable without standards!

How to write good code

- Follow your organisation's coding standards - placement of curly brackets, indenting, variable and method naming...
- Use meaningful names (for variables, methods, classes...) If they become out of date, change them.
- Avoid cryptic comments. Try to make your code so clear that it doesn't need comments.
- Balance structural complexity against code duplication: don't write the same two lines 5 times (why not?) when an easy refactoring would let you write them once, but equally, don't tie the code in knots to avoid writing something twice.
- Be clever, but not too clever. Remember the next person to modify the code may be less clever than you! Don't use deprecated, obscure or unstable language features unless absolutely necessary.
- Remove dead code, unneeded package includes, etc.

Which of these fragments is better?

1.
```
for(double counterY = -8; y < 8; counterY+=0.5){
    x = counterX;
  y = counterY;
  r = 0.33 - Math.sqrt(x*x + y*y)/33;
  r += sinAnim/8;
  g.fillCircle( x, y, r );
 }
```
2.
```
for(double counterY = -8; y < 8; counterY+=0.5){
  x = counterX;
  y = counterY;
  r = 0.33 - Math.sqrt(x*x + y*y)/33;
  r += sinAnim/8;
  g.fillCircle( x, y, r );
}
```
3. They are both fine.

Be consistent about indentation. Don't use TABs

Which of these fragments is better?

1. `c.add(o);`
2. `customer.add(order);`
3. They are both fine.

Use meaningful names.

- A good length for most names is 8-20 chars
- Follow conventions for short names
  - e.g. `i`, `j`, `k` for loop indexes

What else is wrong with this?

```
r = 0.33 - Math.sqrt(x*x + y*y)/33;
r += sinAnim/8;
g.fillCircle( x, y, r );
```

Use white space consistently

Use comments when they're useful

```
if (moveShapeMap!=null) {
    // Need to find the current position. All shapes have
    // the same source position.
    Position pos = ((Move)moveShapeSet.toArray()[0]).getSource();
    Hashtable legalMovesToShape = (Hashtable)moveShapeMap.get(pos);
    return (Move)legalMovesToShape.get(moveShapeSet);
}
```

and not when they're not

```
// if the move shape map is null
if (moveShapeMap!=null) {
```

Too many comments is actually a more common serious problem than too few.

Good code in a modern high-level language shouldn't need many *explanatory* comments, and they can cause problems.

"If the code and the comments disagree, both are probably wrong" (Anon)

But there's another use for comments...

Javadoc

Any software project requires documenting the code – *by which we mean specifying it, not explaining it*.

Documentation held separately from code tends not to get updated.

So use comments as the mechanism for documentation – even if the reader won't need to look at the code itself.

Javadoc is a tool from Sun. Programmer writes doc comments in particular form, and Javadoc produces pretty-printed hyperlinked documentation. E.g. Java API documentation at `http://docs.oracle.com/javase/7/docs/api/`

See Required reading for tutorial.

Javadoc example from tutorial

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param  url  an absolute URL giving the base location of the image
 * @param  name the location of the image, relative to the url argument
 * @return      the image at the specified URL
 * @see         Image
 */
public Image getImage(URL url, String name) {
        try {
            return getImage(new URL(url, name));
        } catch (MalformedURLException e) {
            return null;
        }
}
```

Rendered Javadoc (Eclipse)

Image java.applet.Applet.getImage(URL url, String name)

Returns an `Image` object that can then be painted on the screen. The `url` argument must specify an absolute URL. The `name` argument is a specifier that is relative to the `url` argument.

This method always returns immediately, whether or not the image exists. When this applet attempts to draw the image on the screen, the data will be loaded. The graphics primitives that draw the image will incrementally paint on the screen.

Parameters:

url an absolute URL giving the base location of the image.

name the location of the image, relative to the `url` argument.

Returns:

the image at the specified URL.

See Also:

java.awt.Image

Coding style

Good naming, commenting and formatting are vital components of a good coding style.

But good coding is about much more too. E.g.

- Declaration and use of local variables
  - Good to try keeping local variable scope restricted
- How conditional and loop statements are written
  - With *if-else* statements, put normal frequent case first
  - Use *while*, *do-while*, *for* and *for/in* loops appropriately
  - Avoid deep nesting
- How code is split among methods
  - Good to try avoiding long methods, over 200 lines
- Defensive programming
  - using assertions and handling errors appropriately
- Use of OO features and OO design practices (e.g. patterns)
- Use of packages

The relevance of object orientation

Construction is intimately connected to design: it is design considerations that motivate using an OO language.

Key need: control complexity and abstract away from detail. This is essential for systems which are large (in any sense).

Objects; classes; inheritance; packages all allow us to think about a lot of data at once, without caring about details.

Interfaces help us to focus on behaviour.

Revise classes, interfaces and inheritance in Java.

A common pattern in Java

```
interface Foo {
...
}

class FooImpl implements Foo {
...
}
```

Why is this so much used?

Use of single implementations of interfaces

All about information hiding

- Expect that users of `FooImpl` objects always interact with them at type `Foo`.
  - Maybe they are not even aware of `FooImpl` name.
  - Alternative means of constructing `FooImpl` objects can be set up
- Can use access control modifiers (e.g. *private*, *public*) on `FooImpl` members to define its interface
  - However interface and implementation still mixed together in one class definition
  - This was an early criticism of the OO take on the ADT paradigm
- With distinct interface `foo`, users need see nothing of `FooImpl`

Packages

Recall that Java has packages. Why, and how do you decide what to put in which package?

Packages:

- are units of encapsulation. By default, attributes and methods are visible to code in the same package.
- give a way to organize the namespace.
- allow related pieces of code to be grouped together.

So they are most useful when several people must work on the same product.

But

- the package "hierarchy" is not really a hierarchy as far as access restrictions are concerned – the relationship between a package and its sub/superpackages is just like any other package-package relationship.

Reading

Required: something that makes you confident you completely understand Java packages (e.g., see course web page).

Required: the JavaDoc tutorial

Suggested: *Code Complete* 2nd Ed. Steve McConnell