Inf2: SEPP
Lecture 14: Construction I:
High quality code, object orientation

Cristina Adriana Alexandru

School of Informatics
University of Edinburgh

Last lectures

- Requirements engineering
- Design

This lecture

Construction: high quality code

- What is high quality code?
- Why is high quality code more important for large systems?
- How to write good code:
  - Bracketing
  - Indentation
  - Naming
  - Commenting
  - Javadoc
  - Use of OO features
  - Packages
  - Other advice

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.

Why is high quality code more important for large systems?

Fundamentally because other people will have to read and modify your code, e.g.

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

Even you in a year's time count as "other people"!

How to write good code

First some examples ...

Bracketing conventions

Which of the following is better?

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

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

Is it a good idea to mix these styles in one project, one file, one method?

Settle on a convention and follow it

Indentation

Which of these fragments is better?

A
```
for(double counterY = -8; y < 8; counterY+=0.5){
        x = counterX;
        y = counterY;
   if (y>0) counterY++;
    r = 0.33 - Math.sqrt(x*x + y*y)/33;
 r += sinAnim/8;
    g.fillCircle( x, y, r );
 }
```

B
```
for(double counterY = -8; y < 8; counterY+=0.5){
   x = counterX;
   y = counterY;
   if (y>0)
      counterY++;
   r = 0.33 - Math.sqrt(x*x + y*y)/33;
   r += sinAnim/8;
   g.fillCircle( x, y, r );
}
```

Be consistent about indentation. Don't use TABs

Naming

Which of these statements 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

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

Commenting I

Is the comment below helpful?

```
if (moveShapeMap!=null) {

    // Need to find the current position. All shapes have
    // the same source position, so just pick the first one.
    Position pos = ((Move)moveShapeSet.toArray()[0]).getSource();

    Hashtable legalMovesToShape = (Hashtable)moveShapeMap.get(pos);
    return (Move)legalMovesToShape.get(moveShapeSet);
}
```

Use comments when they're useful

Commenting II

How about this comment?

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

Avoid comments when they are obvious

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 system requires documentation aimed at the *users* of its component parts (e.g. methods, classes, packages).

Documentation held separately from code tends not to get updated, so including such documentation in stylised comments is a good idea.

Javadoc is a tool originally 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/8/docs/api/`

Doxygen (`http://www.doxygen.org`) is a popular corresponding tool for C++.

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

Use of OO features

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.

Make appropriate use of:

- Classes: grouping of behaviour, conceptual abstraction, hiding of implementation from users
- Inheritance: incremental extension without having to know about details
- Interfaces: decouple users from implementations

Revise classes, interfaces and inheritance in Java.

Packages

Recall that Java has packages which:

- allow related pieces of code to be grouped together.
- are units of encapsulation. By default, fields and methods are visible to code in the same package.
- give a way to organize the namespace.
  - They avoid problems if different software components of a large system happen to use the same class names.

Caution: 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.

How to write good code I (Summary)

- 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.
- Use OO features
- Use packages

How to write good code II

Good coding is also about

- 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 design practices (e.g. principles, patterns)

How to write good code III

- 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 overcomplicate the code to avoid writing something twice.
- Remove dead code, unneeded package includes, etc.
- 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.

And much much more.

Excellent books of advice exist, e.g. Code Complete

Reading

Essential: Search on Google and read/watch some introductory Javadoc tutorials.

Recommended: *Code Complete* 2nd Ed. Steve McConnell (one electronic copy available under Resources through the library)

Recommended: something that makes you confident you understand Java packages (e.g., see 'Java/programming resources' under Resources).