Lecture 14: [Construction I] High Quality Code & Object Orientation

Inf2: SEPP

Adriana Sejfia
School of Informatics
University of Edinburgh

Last lectures

Building on what we've covered so far...

- Requirements Engineering — understanding what the system must do
- Design — planning the structure and architecture of the system

This Lecture: Construction — High Quality Code

- What is high quality code?
- Why is high quality code more important for large systems?
  - Bracketing conventions
  - Indentation
  - Naming
  - Commenting & Javadoc
  - Use of OO features
  - Packages
  - Other practical advice

What is High Quality Code?

- High quality code does what it is supposed to do…

What is High Quality Code?

- High quality code does what it is supposed to do…
- …and will not have to be thrown away when requirements change.

What is High Quality Code?

- High quality code does what it is supposed to do…
- …and will not have to be thrown away when requirements change.
- Intimately connected with requirements engineering and design.

What is High Quality Code?

- High quality code does what it is supposed to do…
- …and will not have to be thrown away when requirements change.
- Intimately connected with requirements engineering and design.
- Today we focus on the code itself.

Why high quality code matters (for large systems)

- Other people will have to read and modify your code:
  - Staff movement — team members come and go
  - Code reviews — peers inspect your work
  - Debugging following testing
  - Maintenance over months and years
- Even you in a year's time count as 'other people'!

How to write good code

Starting with some examples…

Bracketing conventions

Style A

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

Style B

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

Bracketing conventions

Style A

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

Style B

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

✓ *Settle on ONE convention and follow it throughout the project.*

Indentation

```
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 );
}
```

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

Indentation

❌ **Misleading (if body on same line)**

```
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 );
}
```

✅ **Consistent indentation**

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

✓ *Be consistent. Don't rely on default TABs — use spaces.*

Naming conventions

```
c.add(o);
```

```
customer.add(order);
```

Naming conventions

❌ **Cryptic names**

```
c.add(o);


// What is c? What is o?
// Reader must trace the whole file
// to understand this line.
```

✅ **Descriptive names**

```
customer.add(order);


// Intent is immediately clear.
// No mental overhead for the reader.
```

✓ *8–20 chars is a good target. Follow conventions: i/j/k for loop indices.*

Consistent whitespace

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

Consistent whitespace

❌ **Inconsistent spacing**

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

✅ **Consistent spacing**

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

✓ *Apply spacing rules uniformly — around operators, inside calls, etc.*

Commenting — when comments help

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);
}
```

Commenting — when comments help

Comments are valuable when they explain *why*, not *what*

Use them sparingly when they're useful

```
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);
}
```

Commenting II

Is the comment below helpful?

```
// if the move shape map is not null
if (moveShapeMap != null) {
    ...
}
```

Commenting — avoid the obvious

- Comments that simply restate the code add noise, not clarity:

Commenting — avoid the obvious

- Comments that simply restate the code add noise, not clarity:
- Too many comments is a more common problem than too few.

Commenting — avoid the obvious

- Comments that simply restate the code add noise, not clarity:
- Too many comments is a more common problem than too few.
- Good code in a modern high-level language shouldn't need many explanatory comments.

Commenting — avoid the obvious

- Comments that simply restate the code add noise, not clarity:
- Too many comments is a more common problem than too few.
- Good code in a modern high-level language shouldn't need many explanatory comments.
- "If the code and the comments disagree, both are probably wrong." — Anon

Commenting — avoid the obvious

- Comments that simply restate the code add noise, not clarity:
- Too many comments is a more common problem than too few.
- Good code in a modern high-level language shouldn't need many explanatory comments.
- "If the code and the comments disagree, both are probably wrong." — Anon
- But there is another valuable use for comments: documentation (Javadoc).

```
// if the move shape map is not null     ← redundant!
if (moveShapeMap != null) {
    ...
}
```

Javadoc

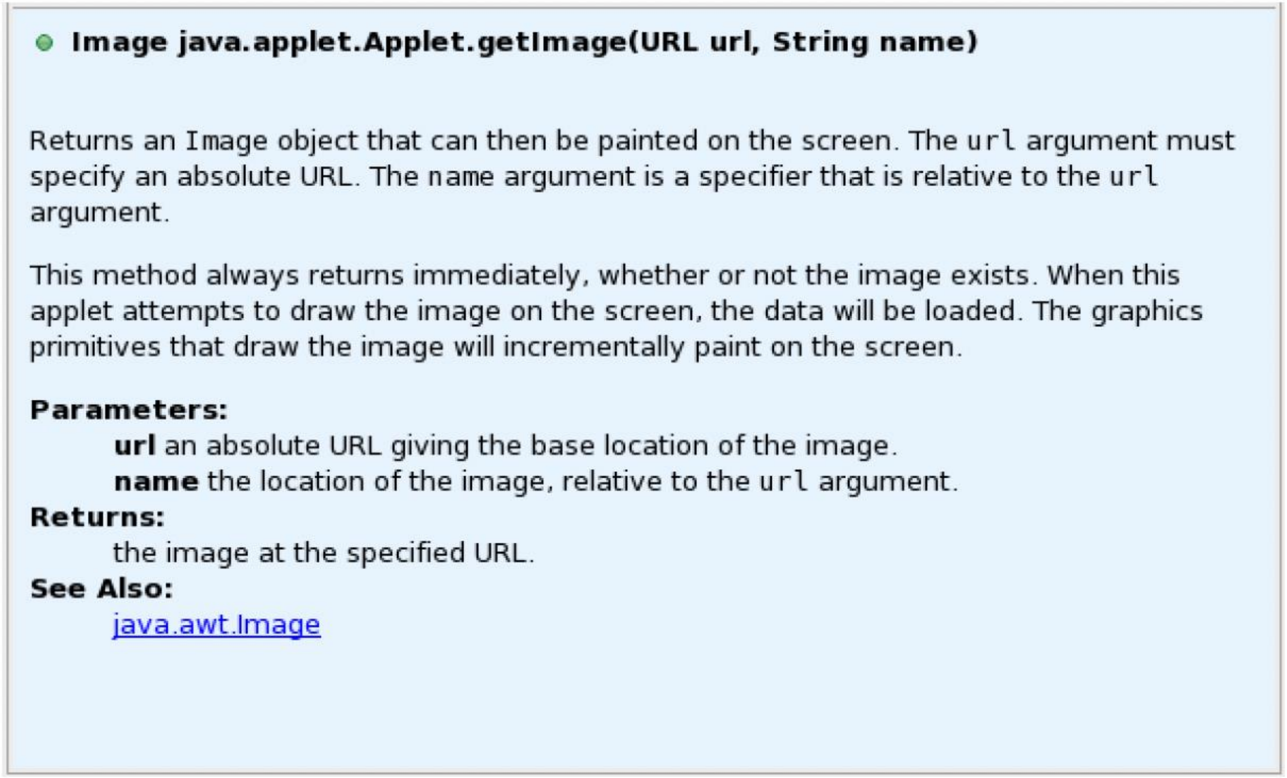
- Any system needs documentation aimed at users of its components (methods, classes, packages).
- Documentation held separately from code tends not to get updated.
- Javadoc (from Sun/Oracle) lets you write stylised comments that generate pretty, hyperlinked HTML docs.
- See the Java 8 API docs: docs.oracle.com/javase/8/docs/api/
- Doxygen (doxygen.org) is the equivalent tool for C++.

Javadoc example

```
/**
 * 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 relative to the url argument.
 *<p>
 * @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;
    }
}
```

Javadoc produces hyperlinked HTML docs. Doxygen is the C++ equivalent.

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

Rendered Javadoc (Eclipse / IntelliJ)

- Javadoc turns the structured comment tags into formatted HTML documentation.
- Sections rendered: method signature, description, Parameters, Returns, See Also.
- @param — documents each parameter with name and description
- @return — describes the return value
- @see — links to related classes or methods
- {@link URL} — inline hyperlink to another class
- IntelliJ and Eclipse both render Javadoc inline as you hover over methods.

Use of object-oriented features

- Construction is intimately connected to design — OO design motivates OO code.

Use of object-oriented features

- Construction is intimately connected to design — OO design motivates OO code.
- Key need: control complexity and abstract away from detail (essential for large systems).

Use of object-oriented features

- Construction is intimately connected to design — OO design motivates OO code.
- Key need: control complexity and abstract away from detail (essential for large systems).
- Make appropriate use of:
    - Classes — grouping of behaviour, conceptual abstraction, hiding implementation

Use of object-oriented features

- Construction is intimately connected to design — OO design motivates OO code.
- Key need: control complexity and abstract away from detail (essential for large systems).
- Make appropriate use of:
  - Classes — grouping of behaviour, conceptual abstraction, hiding implementation
  - Inheritance — incremental extension without knowing implementation details

Use of object-oriented features

- Construction is intimately connected to design — OO design motivates OO code.
- Key need: control complexity and abstract away from detail (essential for large systems).
- Make appropriate use of:
  - Classes — grouping of behaviour, conceptual abstraction, hiding implementation
  - Inheritance — incremental extension without knowing implementation details
  - Interfaces — decouple users from implementations; program to the interface

Use of object-oriented features

- Construction is intimately connected to design — OO design motivates OO code.
- Key need: control complexity and abstract away from detail (essential for large systems).
- Make appropriate use of:
  - Classes — grouping of behaviour, conceptual abstraction, hiding implementation
  - Inheritance — incremental extension without knowing implementation details
  - Interfaces — decouple users from implementations; program to the interface
- Revise classes, interfaces and inheritance in Java before proceeding.

Java Packages

- Packages allow related pieces of code to be grouped together.

Java Packages

- Packages allow related pieces of code to be grouped together.
- They are units of encapsulation: by default, fields and methods are visible within the same package.

Java Packages

- Packages allow related pieces of code to be grouped together.
- They are units of encapsulation: by default, fields and methods are visible within the same package.
- They organise the namespace — avoiding name clashes between large components.

Java Packages

- Packages allow related pieces of code to be grouped together.
- They are units of encapsulation: by default, fields and methods are visible within the same package.
- They organise the namespace — avoiding name clashes between large components.
- **Caution**: the package 'hierarchy' is NOT a true hierarchy for access-restriction purposes.
- A package and its sub-packages have no special access relationship — they are like any other packages.

How to write good code — summary

- Style & Naming:
  - Follow coding standards and conventions
  - Consistent brackets, indentation, whitespace
  - Meaningful names (8–20 chars)

How to write good code — summary

- Style & Naming:
  - Follow coding standards and conventions
  - Consistent brackets, indentation, whitespace
  - Meaningful names (8–20 chars)
- Comments & Documentation:
  - Avoid obvious comments
  - Use Javadoc for public APIs
  - Make code self-documenting

How to write good code — summary

- Style & Naming:
  - Follow coding standards and conventions
  - Consistent brackets, indentation, whitespace
  - Meaningful names (8–20 chars)
- Comments & Documentation:
  - Avoid obvious comments
  - Use Javadoc for public APIs
  - Make code self-documenting
- OO & Structure:
  - Use classes, interfaces, inheritance appropriately
  - Use packages for encapsulation

How to write good code II

- Restrict local variable scope as much as possible.

How to write good code II

- Restrict local variable scope as much as possible.
- Conditionals:
  - With if-else: put the normal/frequent case first.
  - Use while, do-while, for, and for-each loops appropriately.
  - Avoid deep nesting — it hurts readability and testability.

How to write good code II

- Restrict local variable scope as much as possible.
- Conditionals:
  - With if-else: put the normal/frequent case first.
  - Use while, do-while, for, and for-each loops appropriately.
  - Avoid deep nesting — it hurts readability and testability.
- Keep methods short; > 200 lines is a warning sign.

How to write good code II

- Restrict local variable scope as much as possible.
- Conditionals:
  - With if-else: put the normal/frequent case first.
  - Use while, do-while, for, and for-each loops appropriately.
  - Avoid deep nesting — it hurts readability and testability.
- Keep methods short; > 200 lines is a warning sign.
- Use assertions and handle errors with defensive programming.

How to write good code II

- Restrict local variable scope as much as possible.
- Conditionals:
  - With if-else: put the normal/frequent case first.
  - Use while, do-while, for, and for-each loops appropriately.
  - Avoid deep nesting — it hurts readability and testability.
- Keep methods short; > 200 lines is a warning sign.
- Use assertions and handle errors with defensive programming.
- Apply OO design practices: principles and patterns.

How to write good code III

- Balance structural complexity vs. code duplication — don't repeat 5 lines when easy refactoring avoids it, but don't over-engineer to avoid writing something twice.

How to write good code III

- Balance structural complexity vs. code duplication — don't repeat 5 lines when easy refactoring avoids it, but don't over-engineer to avoid writing something twice.
- Remove dead code, unused imports, commented-out blocks.

How to write good code III

- Balance structural complexity vs. code duplication — don't repeat 5 lines when easy refactoring avoids it, but don't over-engineer to avoid writing something twice.
- Remove dead code, unused imports, commented-out blocks.
- Be clever, but not too clever — the next person may be less clever than you.

How to write good code III

- Balance structural complexity vs. code duplication — don't repeat 5 lines when easy refactoring avoids it, but don't over-engineer to avoid writing something twice.
- Remove dead code, unused imports, commented-out blocks.
- Be clever, but not too clever — the next person may be less clever than you.
- Don't use deprecated, obscure, or unstable language features unless absolutely necessary.

How to write good code III

- Balance structural complexity vs. code duplication — don't repeat 5 lines when easy refactoring avoids it, but don't over-engineer to avoid writing something twice.
- Remove dead code, unused imports, commented-out blocks.
- Be clever, but not too clever — the next person may be less clever than you.
- Don't use deprecated, obscure, or unstable language features unless absolutely necessary.
- Excellent resource: Code Complete 2nd Ed. — Steve McConnell.

Reading & Resources

- Essential: Online introductory Javadoc tutorials
- Reference: Oracle's official JavaDoc documentation
- IntelliJ: Using JavaDoc in IntelliJ (see course materials)
- Recommended: Code Complete 2nd Ed. — Steve McConnell
- Recommended: Java packages documentation (Oracle)
- Revision: Java/programming resources on course page