More programming advice

CS 211 Spring 2006

Andrew Myers

Documentation is code

- Comments (esp. specifications) are as important as "real code"
  - determine successful use of code
  - determine whether code can be maintained
  - creation/maintenance = 1/10
- Documentation belongs in code or as close as possible
  - Code evolves, documentation drifts away
  - Put specs in comments next to code when possible
  - Separate documentation? Code should link to it.
- Avoid breaking up code with documentation
  - `x = x + 1; // add one to x` -- Yuck!
  - Need to document algorithm? Write a paragraph at the top. Or break method into smaller, clearer pieces.

Choosing names

- Long names are not specs

```
int searchForElement(int[] array_of_elements,
                     int element_to_look_for);
```

vs.

```
/** clear spec. */
int search(int[] a, int x);
```

- Don't try to document an algorithm with variable names.

```
one_plus_length = the_length + 1; // one more than the length
```

vs.

```
m = n + 1;
```

- Names should be short but suggestive
- Local variable names should usually be short.

Know your audience

- Code and specs have a target audience beyond Java compiler: programmers who use and maintain it
- Goal:
  - Enough documented detail so they can understand it
  - Avoid belaboring the obvious
  - Be brief!
- Try it out on the audience when possible
  - design reviews before coding
  - code reviews

Consistency

- Pick a consistent coding style, stick with it
- Teams should agree on common style
- Match style when *editing* **someone else's** *code*
- Not just syntax, also design style

Exposing the rep

- In Java 1.1, a security-critical class (the class of all class objects, Class):

```
class Class {
  private Object[] signers;
  Object[] getSigners() {
    return signers;
  }
}
```

- Not too secure!
- Problem: exposes the representation to clients
- Options:
  - Copy state to new object
  - Add observers for observing state components, e.g.

```
int numSigners();
Object getSigner(int i);
```

Inheritance vs. encapsulation

- Inheritance is overused by most Java (& OO) programmers
- `class C extends D` means state of D, methods of D are accessible in C
  - Tempting and dangerous!
- C becomes a subtype of D
- Inherit only if a C should be used as a D
  - all methods of D should still make sense
  - A function expecting a D will work on a C
- Try to use Java interfaces instead
  - `D implements I, C implements I`
  - Avoids coupling C and D code

7

Copying code spreads bugs

- Biggest single source of program errors: copying code
  - Bug fixes never reach all the copies
  - Think thrice before using your editor's copy-and-paste function

- Abstract instead of copying!
  - Write many calls to a single function rather than copying the same block of code around

8

Avoid duplication

- Duplication in source code creates an implicit constraint to maintain, a quick path to failure
  - Duplicating code fragments (by copying)
  - Duplicating specs in classes and in interfaces
  - Duplicating specifications in code and in external documents
  - Duplicating same information on many web pages
- Solutions:
  - Named abstractions (e.g., declaring functions)
  - Indirection (linking pointers)
  - Generate duplicate information from source (e.g., Javadoc!)
- *If you must duplicate*:
  - Make duplicates link to each other so always can find all clones

9

Design vs. programming by example

- Programming by example:
  - Copy (!) code that does something like what you want, hack it until it works
- Problems:
  - don't understand code fully
  - usually inherit unwanted functionality and bugs
  - a bolted-together hodge-podge is hard to understand or maintain
- Alternative: design
  - Understand exactly why your code works
  - Reuse abstractions, not code templates

10

Avoid premature optimization

- What people do for speed:
  - Copy code to avoid overhead of abstraction mechanisms
  - Use more complex algorithms & data structures
  - Violate abstraction barriers
- Result: less simple and clear
- Result: performance gains often negligible
- Avoid trying to accelerate performance till:
  - The program is designed and working
  - You know that simplicity needs to be sacrificed
  - You know where simplicity needs to be sacrificed

11

How to make your group project harder

1. Have one person do all the work, so she burns out no one can finish the project
2. Decide that the other member(s) of your group are useless and don't communicate or meet with them.
3. 1+2: decide that all the other members of your group are useless and you are the lone master hacker. Charge off and code everything up without talking to anyone else. Unless you are very unlucky, you'll make some bad assumption that forces all your code to be thrown out anyway.

12

How to make your group project harder, part 2

4. Everyone implements pieces of the system with no discussion of how they will fit together until just before the assignment is due. You won't be able to glue it all together in time.
5. Work extremely closely all the time, spending all your time talking rather than doing actual implementation; the group will slow down to the speed of one person.
   - For extra effectiveness, everyone simultaneously edits different files in the same directory. Something is always broken, testing impossible.
6. Don't start until three days before the assignment is due. Pull three all-nighters in a row. With lack of sleep you will write broken code. With luck, you will get sick, blow some other classes too!

How to make your group project harder, part 3

7. Don't ask the TAs or the professor any questions when design problems come up; put off working on the project and hope the problems will magically solve themselves.
8. Don't use any of the techniques for software design that you learn in this class. This works best if you don't attend class at all -- avoid polluting your mind.

No silver bullets

- These are rules of thumb; every rule has exceptions
- Following software engineering rules only makes success more likely!