Lecture 9: Writing and documenting code

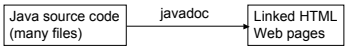
Andrew Myers
CS 211 Spring 2006

Divide-and-conquer programming

- Break program into manageable parts that can be implemented, tested in isolation
- Define interfaces for parts to talk to each other
- Make sure contracts are obeyed
  - Clients use interfaces correctly
  - Implementers implement interfaces correctly (test!)
- Key: good interface documentation
  - Java problem: class interface is mixed in with rest of class definition. Want a separate presentation.

Javadoc

- Extracts documentation from classes, interfaces
  - Requires properly formatted comments
- Produces browsable, hyperlinked HTML web pages



- An important Java tool for presenting code interfaces!
- Some languages (e.g. C++) have separate interface files ("header files" aka ".h files")
  - Provides a separate check that interface is correct
  - Javadoc: convenient, a little dangerous…

Java API

- Javadoc documentation on standard Java libraries available at
  http://java.sun.com/j2se/1.5.0/docs/api/

(demo)

Developing and documenting an ADT

- Abstraction: a closed interval [a,b] on the real number line. [a,b] = { x | a ≤ x ≤ y }
- How to define a Java interface for this ADT?

1. Write an *overview* for the ADT.

Javadoc comment

```
/** An Interval represents a
 * closed interval [a,b]
 * on the real number line.
 */
```

Abstract description of values of ADT.

2. Identify operations

Decide on the right set of operations for the ADT to support

- Should be enough operations for needed tasks
- Avoid unnecessary operations that client could implement efficiently without access to internals of class

3. Write method specs

Write specifications for each operation (method).

- Signature: types of method arguments, return.
- Description of what the method does (abstractly).

- Good (definitional):

```
/** Add two intervals. The sum of two intervals is
 * a set of values containing all possible sums of
 * two values, one from each of the two intervals. */
 public Interval plus(Interval i);
```

- Bad: (operational):

```
/** Return a new Interval with lower bound a+i.a,
 *  upper bound b+i.b. */
```

  - Not abstract, might as well read the code…

Parts of a method spec

Method overview

```
/** Add two intervals. The sum of two intervals is
 * a set of values containing all possible sums of
 * two values, one from each of the two intervals.
 *
 * @param i the other interval
 * @return the sum of the two intervals
 */
```

Method description

Additional tagged clauses

- Attach before methods of class or interface.
  - No need in class if it impls interface with specs

Some useful Javadoc tags

- `@return` *description*
  - Use to describe the return value of the method, if any
  - E.g., `@return the sum of the two intervals`
- `@param` *parameter-name description*
  - Describes the parameters of the method
  - E.g., `@param i the other interval`
- `@author` *name*
- `@deprecated` *reason*
- `@see` *package.class#member*
- `{@code` *expression*`}`
  - Put expression in code font

More thoughts on programming

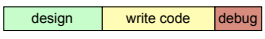
- Careful interface design makes program development more likely to succeed
- Many other ways to make program development easier (or harder)…
  - Applies to other engineering too…

Design is faster than debugging

- Timeline is Design ⇒ Code ⇒ Debug
- Timeline like this?

  - Common error: not enough time spent designing

- Extra time designing reduces coding and debugging time
  - Speed, not haste
  - An ounce of prevention…



Pair programming

- Work in pairs
- Pilot/copilot
  - Pilot codes, copilot directs
  - Pilot must convince copilot that code works
- Or: work more independently
  - frequent design review: both programmers must convince the other
- Reduces debugging time

Simplicity

The present letter is a very long one, simply because I had no time to make it shorter. –Blaise Pascal

Be brief. –Strunk & White

- Applies to programming… simple code is:
  - Easier and quicker to understand
  - More likely to work correctly
- Good code is simple, short, and clear
  - Save complex algorithms, data structures for where they are needed.
  - Always reread code (and writing) to see if it can be made shorter, simpler, clearer.

Know your audience

- Code and specs have a target audience: the programmers who will maintain, use it
- Should be written with
  - Enough documented detail so they can understand it
  - while avoiding spelling out the obvious

Consistency

A foolish consistency is the hobgoblin of little minds -- Emerson

- Pick a consistent coding style, stick with it
  - Make your code understandable by "little minds"
- Teams should set common style
- Match style when *editing* **someone** else's *code*

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

Premature optimization

- What people do for speed:
  - Copy code to avoid overhead of abstraction mechanisms
  - Write more complex, longer code
  - Violate abstraction barriers
- Result: not simple or clear
- Performance gains often negligible
  - Avoid trying to accelerate performance until you
    - Have the program designed and working
    - Know that simplicity needs to be sacrificed
    - Know where simplicity needs to be sacrificed

Design vs. programming by example

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