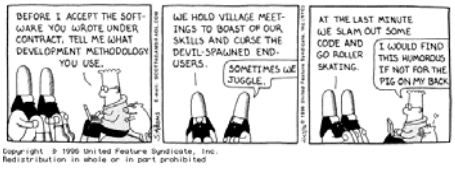
Designing, Coding, and Documenting



Lecture 10
CS211 – Spring 2007

Announcements

- A2 due Sunday
  - better start now
- Prelim March 8
  - If you have a conflict, notify Kelly Patwell ASAP
  - see website for contact info
- A3 due March 14
  - more difficult than the previous assignments – more code
  - but we will give you lots of direction

Quiz 2 – What do you find most difficult?

grammars, parsing	37
induction	30
recursion	13
time management, assigment takes too long	12
inheritance	5
understanding assignment directions	5
nothing	4
linked lists	3
quizzes	2
using the Java API	2
applying lecture to assignments	2
staying awake, too dark in classroom	2
error handling	2
assignment 2	2
reading the textbook	2
debugging	1
getting up in time for class	1

Quiz 2 – What value is printed?

```
class Foo {
  String s;

  Foo(String t) {
    s = "Happy " + t;
  }

  public String toString() {
    return s;
  }
}

class Bar extends Foo {
  Bar(String r) {
    super("New " + r);
  }
}

System.out.println(new Bar("Year!"));
```

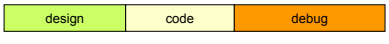
t: "New Year!"
s: "Happy New Year!"
r: "Year!"
super("New Year!")

Designing and Writing a Program

- Don't sit down at the terminal immediately and start hacking
- Design stage – *THINK* first
  - about the data you are working with
  - about the operations you will perform on it
  - about data structures you will use to represent it
  - about how to structure all the parts of your program so as to achieve abstraction and encapsulation
- Coding stage – code in small bits
  - test as you go
  - understand preconditions and postconditions
  - insert sanity checks (assert statements in Java are good)
  - worry about corner cases
- Use Java API to advantage

The Design-Code-Debug Cycle

- Design is faster than debugging (and more fun)
  - extra time spent designing reduces coding and debugging
- Which is better?



- Actually, should be more like this:

Divide and Conquer!

- Break program into manageable parts that can be implemented, tested in isolation
- Define interfaces for parts to talk to each other – develop *contracts* (preconditions, postconditions)
- Make sure contracts are obeyed
  - Clients use interfaces correctly
  - Implementers implement interfaces correctly (test!)
- Key: good interface documentation

Pair Programming

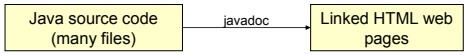
- Work in pairs
- Pilot/copilot
  - pilot codes, copilot watches and makes suggestions
  - pilot must convince copilot that code works
  - take turns
- Or: work independently on different parts after deciding on an interface
  - frequent design review
  - each programmer must convince the other
  - reduces debugging time
- Test everything

Documentation is Code

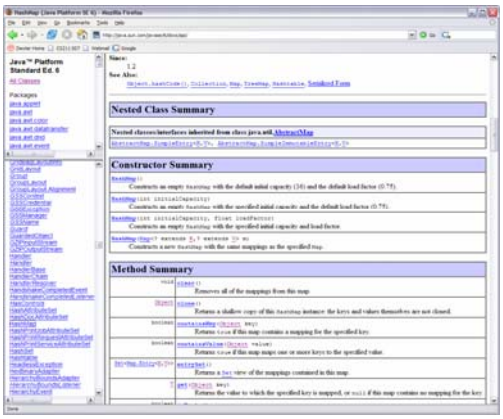
- Comments (esp. specifications) are as important as the code itself
  - determine successful use of code
  - determine whether code can be maintained
  - creation/maintenance = 1/10
- Documentation belongs in code or as close as possible
  - Code evolves, documentation drifts away
  - Put specs in comments next to code when possible
  - Separate documentation? Code should link to it.
- Avoid useless comments
  - `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.

Javadoc

- An important Java documentation tool



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



How Javadoc is Produced

indicates Javadoc comment

Javadoc keywords

can include HTML

```
/**
 * Constructs an empty <tt>HashMap</tt> with the specified initial
 * capacity and the default load factor (0.75).
 *
 * @param  initialCapacity the initial capacity.
 * @throws IllegalArgumentException if the initial capacity is negative.
 */
public HashMap(int initialCapacity) {
    this(initialCapacity, DEFAULT_LOAD_FACTOR);
}

/**
 * Constructs an empty <tt>HashMap</tt> with the default initial capacity
 * (16) and the default load factor (0.75).
 */
public HashMap() {
    this.loadFactor = DEFAULT_LOAD_FACTOR;
    threshold = (int)(DEFAULT_INITIAL_CAPACITY * DEFAULT_LOAD_FACTOR);
    table = new Entry[DEFAULT_INITIAL_CAPACITY];
    init();
}
```

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***`}`**
- Puts expression in code font

Developing and Documenting an ADT

1. Write an overview – purpose of the ADT
2. Decide on a set of supported operations
3. Write a specification for each operation

1. Writing an ADT Overview

- Example abstraction: a closed interval [a,b] on the real number line
  - [a,b] = { x | a ≤ x ≤ y }
- Example overview:

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

Javadoc comment

Abstract description of the ADT's values

2. Identify the Operations

- Enough operations for needed tasks
- Avoid unnecessary operations – keep it simple!
  - Don't include operations that client (without access to internals of class) can implement

3. Writing Method Specifications

- Include
  - Signature: types of method arguments, return type
  - Description of what the method does (abstractly)
- Good description (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 description (operational)

```
/** Return a new Interval with lower bound a+i.a,
 * upper bound b+i.b.
 */
 public Interval plus(Interval i);
```

Not abstract, might as well read the code…

3. Writing Specifications (cont'd)

- Attach before methods of class or interface

```
/** 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 interv
 */
```

Method overview
Method description
Additional tagged clauses

Know Your Audience

- Code and specs have a target audience
  - the programmers who will maintain and use it
- Code and specs should be written
  - With enough documented detail so they can understand it
  - While avoiding spelling out the obvious
- Try it out on the audience when possible
  - design reviews before coding
  - code reviews

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*
  - Not just syntax, also design style

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 be correct
- 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

Choosing Names

- Don't try to document with variable names
  - Longer is not necessarily better

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

int search(int[] a, int x);
```

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

Avoid Copy-and-Paste

- Biggest single source of program errors
  - Bug fixes never reach all the copies
  - Think twice 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

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

What Makes a Good Algorithm?

- Suppose you have two possible algorithms or data structures that basically do the same thing; which is *better*?
- Well…what do we mean by *better*?
  - Faster?
  - Less space?
  - Easier to code?
  - Easier to maintain?
  - Required for homework?
- How do we measure time and space for an algorithm?

Inheritance vs Encapsulation

- Inheritance is useful, but it is possible to overdo it
- Overused by many Java & OO programmers
- `class C extends D` means state of D, methods of D are accessible in C
  - Tempting, often useful, but also can be 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
- Prefer Java interfaces instead

Avoid Premature Optimization

- Temptations to avoid
  - Copying code to avoid overhead of abstraction mechanisms
  - Using more complex algorithms & data structures unnecessarily
  - Violating abstraction barriers
- Result:
  - Less simple and clear
  - Performance gains often negligible
- Avoid trying to accelerate performance until
  - You have the program designed and working
  - You know that simplicity needs to be sacrificed
  - You know where simplicity needs to be sacrificed

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

Maintain State in One Place

- Often state is duplicated for efficiency
- But difficult to maintain consistency
- *Atomicity* is the issue
  - if the system crashes while in the middle of an update, it may be left in an inconsistent state
  - difficult to recover

How to Make your Group Project Harder

1. Have one person do all the work, so that person burns out, and no one else 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.

How to Make your Group Project Harder

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 the slowest person.
   - For extra effectiveness, everyone simultaneously edits files in the same directory, preferably the same file. 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, miss some other classes as well.

How to Make your Group Project Harder

7. Don't ask the TAs or the instructors 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 all rules of thumb; but there is no panacea, and every rule has its exceptions
- You can only learn by doing – we can't do it for you
- Following software engineering rules only makes success more likely!