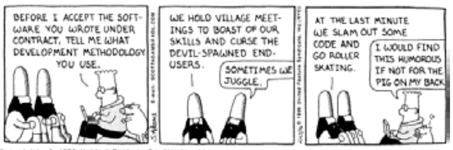
Designing, Writing, and Documenting



Lecture 11
CS211 – Fall 2006

Announcements

- Finding a partner
  - Check the newsgroup
    - Just post a message asking for a partner
  - Use signup sheet
    - We'll put you in contact with potential partners
- Upcoming Prelim I
  - Thu, Oct 12, 7:30-9:00pm
    - Same week as Fall Break
  - If you have a conflict, notify Course Administrator (Kelly Patwell; see website)
- A3 due date is now Oct 23 (after the prelim)
  - A3 is harder than the previous assignments
  - Start now!

Defining a "Natural Ordering"

- An object's natural ordering is determined by its compareTo method
  - For Java to know that an class can be compared, the class must implement the Comparable interface
- Java provides tools to work with objects of type Comparable
  - Examples: sort, binarySearch

```
public class Person
   implements Comparable<Person> {
   private String name;
   private int id, height, weight;

   public Person (String name, int id,
                  int height, int weight) {
      this.name = name; this.id = id;
      this.height = height;
      this.weight = weight;
   }

   public int compareTo (Person p) {
      return id - p.id;
   }
```

"Unnatural" Ordering

- The ordering given by compareTo is considered to be the *natural ordering* for a class
- Sometimes you need to sort based on a different ordering
  - Example: we may normally sort students by CUID, but we might want to produce a list alphabetized by name

```
interface Comparator<T> {
   int compare (T x, T y);
}
```

- Can use a Comparator (a class that implements the Comparator interface)

Arrays.sort(students,comparator)

- String, for example, has a predefined Comparator:

String.CASE_INSENSITIVE_ORDER

Comparators for the Person Class

```
class NameComparator implements Comparator<Person> {
   public int compare (Person p, Person q) {
      return p.getName().compareTo(q.getName());
   }
}

class HeightComparator implements Comparator<Person> {
   public int compare (Person p, Person q) {
      return p.getHeight() - q.getHeight();
   }
}

class WeightComparator implements Comparator<Person> {
   public int compare (Person p, Person q) {
      return p.getWeight() - q.getWeight();
   }
```

Sorting an Array of Persons

```
import java.util.Arrays;
Person[] p = …
```

- Sort by ID (this is the natural ordering) — `Arrays.sort(p);`
- Sort by name — `Arrays.sort(p, new NameComparator());`
- Sort by height — `Arrays.sort(p, new HeightComparator());`
- Sort by weight — `Arrays.sort(p, new WeightComparator());`

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 criticism: Class interface is mixed in with rest of class definition

Javadoc

- An important Java tool for presenting program interfaces!

Java source code (many files) → javadoc → Linked HTML web pages

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

- Some languages (e.g., C++) have separate interface files ("header files" aka ".h files")
  - Provides a separate check that interface is correct

Developing and Documenting an ADT

1. Write an overview for the ADT
2. Decide on the right set of supported operations for the ADT
3. Write specifications for each operation

1. Writing an ADT Overview

- Example abstraction: a closed interval [a,b] on the real number line
  - $[a,b] = \{ x \mid a \le x \le y \}$
- Example overview:

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

Abstract description of the ADT's values

Javadoc comment

2. Deciding on the Operations

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

3. Writing 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 intervals
*/
```

Method overview
Method description
Additional tagged clauses

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

Know Your Audience

- Code and specs have a target audience:
  the programmers who will maintain, use it
- Code and specs 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

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

Avoid Premature Optimization

- Temptations (to achieve speed)
  - Copy code to avoid overhead of abstraction mechanisms
  - Write more complex, longer code using fancier data structures
  - 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*
  - You *know* that simplicity needs to be sacrificed
  - You know *where* simplicity needs to be sacrificed
    - Can determine using *profiling* tools

Don't Copy-and-Paste Code

- Biggest single source of program errors: copying code
  - 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

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?

Sample Problem: Searching

- Determine if a *sorted* array of integers contains a given integer
- 1st solution: Linear Search (check each element)

```
static boolean find (int[ ] a, int item) {
  for (int i = 0; i < a.length; i++) {
        if (a[i] == item) return true;
        }
  return false;
  }
```

- 2nd solution: Binary Search

```
static boolean find (int[ ] a, int item) {
   int low = 0;
   int high = a.length - 1;
   while (low <=  high) {
            int mid = (low+high)/2;
            if (a[mid] < item)
                     low = mid+1;
            else if (item < a[mid])
                     high = mid - 1;
            else return true;
            }
   return false;
   }
```

Linear Search vs. Binary Search

- Which one is better?
  - Linear Search is easier to program
  - But Binary Search is faster… isn't it?
- How do we measure to show that one is faster than the other
  - Experiment?
  - Proof?
  - But which inputs do we use?
- Simplifying assumption #1: Use the *size* of the input rather than the input itself
  - For our sample search problem, the input size is n+1 where n is the array size
- Simplifying assumption #2: Count the number of "*basic steps*" rather than computing exact times

One Basic Step = One Time Unit

- Basic step:
  - input or output of a scalar value
  - accessing the value of a scalar variable, array element, or field of an object
  - assignment to a variable, array element, or field of an object
  - a single arithmetic or logical operation
  - method invocation (not counting argument evaluation and execution of the method body)
- For a conditional, we count number of basic steps on the branch that is executed
- For a loop, we count number of basic steps in loop body times the number of iterations
- For a method, we count number of basic steps in method body (including steps needed to prepare stack-frame)

Runtime vs. Number of Basic Steps

- But isn't this cheating?
  - The runtime is not the same as the number of basic steps
  - Time per basic step varies depending on computer, on compiler, on details of code…
- Well… yes, it is cheating in a way
  - But the number of basic steps is *proportional* to the actual runtime
- Which is better?
  - n or n^2 time?
  - 100 n or n^2 time?
  - 10,000 n or n^2 time?
- As n gets large, multiplicative constants become less important
- Simplifying assumption #3: Multiplicative constants aren't important