

## CMSC 433 – Programming Language Technologies and Paradigms Spring 2003

Specifications  
February 20, 2003

## Administrivia

- Commentary for project 1 due Friday in Jeff's section, Monday in Bill's section
  - New version of Submit.jar available
- Project 2 due on 28th (week from Friday)
  - Junit installed on linuxlab, in default class path

## Project 2

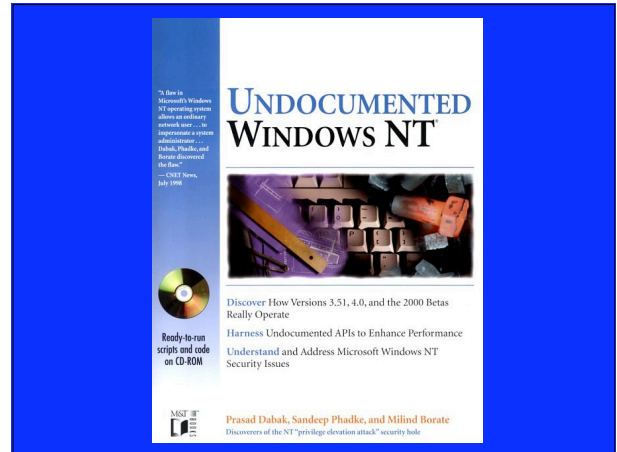
- We write the code, you write the tests
  - Specification
    - Graph.java, Vertex.java
    - GraphFactory.java, DotFormatException.java
  - *Sample* implementation
    - MyGraph.java
    - (GraphFactory.java, DotFormatException.java)

## Important Things to Notice

- We have given you one implementation
  - Incomplete
  - Not necessarily correct
  - Your tests should check the behavior of any valid implementation
- Test cases should follow TestRunner pattern
  - Test methods begin with test, public, no args, void ret
  - Write as many classes as you like/need
  - We will test all classes that extend of TestCase

## Software Specifications

- A software specification defines the behavior of an abstraction
  - not the implementation
- Any user of your code should depend on your specification
- When revising the code, you are free to change the implementation, so long as the code still meets the specification
  - users that depend on your implementation rather than your specification could be in trouble



## Specifications are important

- Good specifications are important
- Key to allowing software composition
  - Write code that uses version 1.3.1 of a library
    - Still works with version 1.4.0 of that library
  - Write version 1.3.1 of a library
    - introduce improvements in version 1.4.0 that don't break everyone's code

## Good specifications are hard and rare

- Very difficult to get people to write specifications
  - even harder to keep them up to date
- Having specifications in a separate document from code almost guarantees failure
  - rationale for Javadoc
- Hard to accurately and formally capture all properties of interest
  - always finding important details not specified

### Specifications help you write code

- For many subtle and interesting algorithms and data structures, having internal specifications/invariants about the algorithm and data structure are vital to getting the code right
  - e.g., in a binary search tree, all nodes reachable from the left branch have a smaller key than the current node, and all nodes reachable from the right branch have a larger key than the current node

### Specifications help you maintain code

- In the real world, much coding effort goes into modifying previously written code
  - often originally written by somebody else
  - perhaps six different people have modified this code
- Documenting and respecting key internal specifications are the way to avoid a bloody mess
- Documenting and respecting key external specifications are the way to avoid having your customers storm the office with torches and pitchforks

### Public vs. Private Specifications

- No fundamental difference
- Just who the target audience is
  - Who reads it
  - Who's permission you have to get if you want to change it

### Formal vs. Informal Specifications

- Formal specifications
  - for all  $i$ ,  $0 < i < d.length$ ,  $d[i-1] < d[i]$   
and there exists  $j$ ,  $0 \leq j < d.length$ , such that  $d[j] = x$
- Informal specifications
  - the array  $d$  is sorted, and some element of the array  $d$  is equal to  $x$

## Advantages and Disadvantages

- Formal specifications
  - Forces you to be very clear
  - automated tools can check some specifications
    - either at compile-time (static checking) or run-time (dynamic checking)
- Informal specifications
  - Some important properties are hard to express formally
    - Sometimes just difficult
    - Sometimes we don't have the necessary formal notation
  - Some people are intimidated by formal specs

## Types of external specifications

- Pre-conditions/requires: What must be true before invoking a method
  - `int find(int d[], int x)`
  - precondition: the array `d` is sorted
- Post-conditions/effects: What is guaranteed to be true after invoking a method
  - postcondition:
    - `returnValue >= 0` and `d[returnValue] == x`
    - or `(returnValue == -1` and `x` does not occur in `d`)
  - often relates final values to initial values

## Types of internal specifications

- Loop invariants: condition that must hold at the beginning of each iteration of a loop
  - `d[0..i]` is sorted
- Data structure or field invariants
  - `elementCount <= elementData.length`

## Difficult issues

- Side effects
  - What effects does this operation have
- Performance
  - should you specify performance of operations
  - as hard as 451: what kind of bound (upper bound, amortized bound, expected bound, ...), order of bound, ...
  - But need at least informal specs
    - random access is fast, insertion/deletion can be slow

## What Makes a Good Specification?

- Sufficiently restrictive
  - Forbids unacceptable implementations
- Sufficiently general
  - Allows all acceptable implementation
- Clear
  - Easy to understand
  - A little redundancy may help

## Specifications/Interfaces as Contracts

- A specification acts as a contract
  - between the developer of a library
  - and the user of a library

## OO Programming

- OO Programming involves two very different concepts
  - inheritance - code reuse
    - in defining class B, I want to reuse method implementations defined for class A
  - subtyping - substitutivity
    - I want to be able to supply a B to someone who expects an A
    - Is B's implementation compatible with A's specification?

## Liskov substitution principle

- (Original?) Formal statement
  - If for each object  $o1$  of type  $S$  there is an object  $o2$  of type  $T$  such that for all programs  $P$  defined in terms of  $T$ , the behavior of  $P$  is unchanged when  $o1$  is substituted for  $o2$  then  $S$  is a subtype of  $T$ .
- Doesn't really handle interfaces or abstract classes
- Informal statement
  - If anyone expecting a  $T$  can be given an  $S$ , then  $S$  is a subtype of  $T$

## In an OO context, S subtype of T means

- The methods supported on instances of T are supported on instances of S
  - with compatible meanings
    - if the get method on T signals `notFound` by throwing an exception, S can't signal it by returning null

## Overriding/Implementing Functions

- Given

```
/** Search array for value */
/** @precondition: a is sorted */
/** @postcondition: returns index i s.t. a[i] == value,
                    or -1 if no such value exists */
int find( int [] d, int value);
```
- In an implementation of this function can we
  - a) Change the precondition to true?
  - b) Change the precondition to d is sorted and there exists an i s.t. d[i] = value?
  - c) Change the postcondition so that i == -1 or i is the first index s.t. d[i] == value?
  - d) Change the function so that it throws "NoSuchElementException" rather than returning -1 when value does not occur in d.

## Javadoc

- Java mechanism for integrating documentation into source code as comments
- `/** Javadoc comment */`

## JavaDoc example

```
/** Javadoc Comment for this class */
public class MyClass {
    /** Javadoc Comment for field text */
    String text;
    /** Javadoc Comment for method setText
     * @param t Javadoc comment for parameter t
     */
    public void setText(String t) {...}
}
```

## Javadoc tags

- Special tags for classes
  - [@author](#)
  - [@version](#)
- Special tags for methods
  - [@param](#)
  - [@return](#)
  - [@exception](#)
- Reference to another element
  - [@see](#)
- Can contain any HTML code

## A more detailed 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 a specifier that is relative to the url argument.
 * <p>
 * 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.
 *
 * @param url an absolute URL giving the base location of the image
 *         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;
    }
}
```

## Generated HTML

### getImage

```
public Image 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:

[Image](#)

## Javadoc tags

- [@author](#) (classes and interfaces only, required)
- [@version](#) (classes and interfaces only, required)
- [@param](#) (methods and constructors only)
- [@return](#) (methods only)
- [@exception](#) ([@throws](#) is a synonym added in Javadoc 1.2)
- [@see](#)
- [@since](#)
- [@serial](#) (or [@serialField](#) or [@serialData](#))
- [@deprecated](#) (see How and When To Deprecate APIs )