Software Specifications

- A specification defines the behavior of an abstraction
- This is the contract between user and provider
  - Provider’s code must implement the specification
  - Providers are free to change the implementation
    - So long as the new code meets the specification
  - Users who depend only on specification won’t have trouble
    - Don’t rely on implementation
- Black box testing essentially checks compliance of an implementation with its specification

Good Specifications are Hard to Write

- 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: extract a standalone specification from the code and embedded comments
- Hard to accurately and formally capture all properties of interest
  - Always finding important details not specified

Specifications Help You Write Code

- Lots of subtle algorithms and data structures
  - Internal specs/invariants vital to correct implementation
- Example: Binary Search Tree
  - All nodes reachable from left child have smaller key than current node
  - All nodes reachable from right child have larger key than 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 many different people have modified this code
- Documenting and respecting key internal specifications are the way to avoid a mess
- Documenting and respecting key external specifications are the way to avoid having your customers storm the office with torches and pitchforks

Formal vs. Informal Specifications

\[\text{static int find(int} \ d, \text{int} \ x)\]

- An informal specification
  - If the array \(d\) is sorted, and some element of the array \(d\) is equal to \(x\), then \(\text{find()}\) returns the index of \(x\) ……

- A formal specification
  - \((\text{for all} \ i, 0 < i < d.\text{length}, d[i-1] < d[i] \text{ and there exists } j, 0 <= j < d.\text{length}, \text{such that } d[j] == x) \text{ implies find}(d,x) = j \) ……

- Note: These specs assume array has no duplicates

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 don’t have the necessary formal notation
  - Some people are intimidated by formal specs

Types of External Specifications

- Specifications on methods
  - Pre-conditions/requires: What must be true before call
  - Post-conditions/effects: What is must be true after call
    - Often relates final values to initial values

\[
\begin{align*}
\text{// precondition: the array } d \text{ is sorted} \\
\text{// postcondition:} \\
\text{// return} & \text{Value } >= 0 & \& \& d[\text{returnValue}] == x \\
\text{or } & \text{return} \text{Value } == -1 & \& \& x \text{ does not occur in } d \\
\text{static int find(int} & \ d[], \text{int} \ x); \\
\end{align*}
\]
Types of Internal Specifications

- Specifications appearing within code itself
  - i.e., comments
- 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
  - $\text{elementCount} \leq \text{elementData.length}$

Specifications and Subtyping

- Liskov substitution principle (original? formal stmt)
  - If for each object $o_1$ of type $S$ there is an object $o_2$ of type $T$ such that for all programs $P$ defined in terms of $T$, the behavior of $P$ is unchanged when $o_1$ is substituted for $o_2$ then $S$ is a subtype of $T$.
  - i.e., if anyone expecting a $T$ can be given an $S$, then $S$ is a subtype of $T$.
- If we override a method, how do the specifications of the original and new method relate?

Specifications and Subtyping (cont’d)

```java
// precondition: the array $d$ is sorted
// postcondition:
//    returnValue >= 0 && d[returnValue] == x
// or (returnValue == -1 && x does not occur in $d$)
static int find(int $d[]$, int $x$);
```

- If we override this method, can the new method
  - Have true as a precondition?
  - Have precond “$d$ is sorted and exists $i$ s.t. $d[i] == x$”?
  - Have postcond “$\text{returnValue} == -1$ or $\text{returnValue}$ is first $d[\text{returnValue}] == x$”?
  - Throw $\text{NoSuchElementException}$ rather than returning -1 when $x$ does not occur in $d$?

What Makes a Good Specification?

- Sufficiently restrictive
  - Forbids unacceptable implementations
- Sufficiently general
  - Allows all acceptable implementations
- Clear
  - Easy to understand
  - A little redundancy may help (some people disagree)
Javadoc

- Integrates documentation into source code as comments
- Will generate an external specification

```java
/** 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 example

```java
/** Given a sorted array, returns the index into the array of the given element, otherwise returning -1. 
   @param d array to search in, assumed sorted 
   @param x the element to search for 
   @returns i >= 0 when d[i] == x, and -1 when x does not occur in d 
*/
public static int find(int d[], int x) {
    ...
}
```

Javadoc example: HTML

<table>
<thead>
<tr>
<th>Method Detail</th>
</tr>
</thead>
<tbody>
<tr>
<td>find</td>
</tr>
</tbody>
</table>
| public static int find(int d[], int x) { 
  
  Given a sorted array returns the index into the array of the given element, otherwise returning -1.
  
  Parameters:
  0 - array to search in, assumed sorted 
  x - the element to search for 
  Returns:
  i >= 0 when d[i] == x, and -1 when x does not occur in d |

A Few Javadoc Tags

- Special tags for classes
  - @author
  - @version
- Special tags for methods
  - @param
  - @return
  - @exception
- Reference to another element
  - @see
Object Modeling Technique (OMT)

- Graphical representation of OO relationships
  - **Class diagrams** show the static relationship between classes
  - **Object diagrams** represent the state of a program as series of related objects
  - **Interaction diagrams** illustrate execution of the program as an interaction among related objects

Classes

```
ClassName
Operation1()
Type Operation2()
...
instanceVariable1
Type instanceVariable2
...
```

Object instantiation

```
Instaniator                      Instantiatee
```

Subclassing and Abstract Classes

```
ParentClass                    AbstractClass
Operation()                    Operation()

ConcreteSubclass              Implementation
Operation()                   pseudocode
```
Pseudo-code and Containment

```
Window
  Area()
  rectangle
  return rectangle->Area()

Rectangle
  Area()
  width
  height
  return width * height
```

Object diagrams

```
Object diagrams

aDrawing
  shape[0]
  shape[1]
  aLineShape
  aCircleShape
```

Interaction diagrams

```
Interaction diagrams

aCreationTool
  newLineShape
  Add(aLineShape)
  Refresh()
  Draw()
```