Testing

Fishing for Bugs
Each test examines one path through the program

Exhaustive
All possible inputs: infeasible for all non-trivial programs

Path-Complete
All possible paths through the program

Path-Complete Testing

Insufficient
One execution of a path doesn’t cover all behaviors
Often bugs are missing paths

Impossible
Most programs have an “infinite” number of paths
Branching
Can test all paths
Loops and recursion
Test with zero, one and several iterations

Coverage Testing

Statement Coverage:
number of statements executed on at least one test
number of statements in program

Can we achieve 100% statement coverage?
Testing Recap

- Testing can find problems, but cannot prove your program works
  - Since exhaustive testing is impossible, select test cases with maximum likelihood of finding bugs
  - A successful test case is one that reveals a bug in your program!
- Typically at least 40% of cost of software project is testing, often >80% of cost for safety-critical software

Is it really hopeless?

Since we can’t test all possible paths through a program, how can we increase our confidence that it works?

Analysis

- Make claims about all possible paths by examining the program code directly
  - Testing (dynamic analysis): checks exactly one program path
  - Static analysis: reasons about all possible program paths
- Use formal semantics of programming language to know what things mean
- Use formal specifications of procedures to know that they do

Hopelessness of Analysis

It is impossible to correctly determine if any interesting property is true for an arbitrary program!

The Halting Problem: it is impossible to write a program that determines if an arbitrary program halts.

Compromises

- Use imperfect automated tools:
  - Accept unsoundness and incompleteness
  - False positives: sometimes an analysis tool will report warnings for a program, when the program is actually okay (unsoundness)
  - False negatives: sometimes an analysis tool will report no warnings for a program, even when the program violates properties it checks (incompleteness)
    - Java compiler warnings attempt to do this
- Use informal reasoning

Dealing with Hopelessness

Since both testing and analysis are hopeless in general what can we do?

Design for Testability  Design for Analyzability

Modularity  Decoupling  Narrow Interfaces
Programming Defensively

Assertions

Statement ::= assert booleanExpression optStringExpression;
booleanExpression ::= [any Java expression that evaluates to a boolean value]
optStringExpression ::= ε | : stringExpression
stringExpression ::= [any Java expression that can be converted to a String value]

Semantics: To evaluate an assert statement, evaluate the booleanExpression. If the booleanExpression evaluates to true, do nothing. If it is false, the assertion fails and an AssertionError thrown. If there is an optional stringExpression, it is evaluated (and converted to a String) and included in the AssertionError.

Examples

public class TestClass{
    public static double divide(int a, int b) {
        assert b != 0;
        return (double) a / b;
    }
    public static void main(String[] args) {
        System.out.println(divide(3, 4));
        System.out.println(divide(3, 0));
    }
}

0.75
Exception in thread “main” java.lang.AssertionError
at ps3.TestClass.divide(TestClass.java:6)
at ps3.TestClass.main(TestClass.java:16)

Enabling Assertions

Without this, assert does nothing!

Examples

public class TestClass{
    public static double divide(int a, int b) {
        assert b != 0 : "Division by zero";
        return (double) a / b;
    }
    public static void main(String[] args) {
        System.out.println(divide(3, 4));
        System.out.println(divide(3, 0));
    }
}

0.75
Exception in thread “main” java.lang.AssertionError: Division by zero
at ps3.TestClass.divide(TestClass.java:6)
at ps3.TestClass.main(TestClass.java:16)

Tricky Example

public class TestClass{
    public static double divide(int a, int b) {
        assert b != 0 : divide(a, b);
        return (double) a / b;
    }
    public static void main(String[] args) {
        System.out.println(divide(3, 4));
        System.out.println(divide(3, 0));
    }
}

0.75
Exception in thread “main” java.lang.StackOverflowError
at ps3.TestClass.divide(TestClass.java:6)
at ps3.TestClass.divide(TestClass.java:6)
at ps3.TestClass.divide(TestClass.java:6)
at ps3.TestClass.divide(TestClass.java:6)
at ps3.TestClass.divide(TestClass.java:6)
...
Where should we use `assert`?

1. To give useful debugging information when a \textit{REQUIRES} precondition is violated.
2. To check assumptions on which our code relies.

Judicious use of asserts:
- saves debugging time
- provides useful documentation
- increases confidence in results

How many assertions?

About 5% of the statements in a good Java program should be asserts!


200 assertions per 1000 lines of code

Exceptions

- In C/C++: can lead to anything
  - Machine crash
  - Security compromise
  - Strange results
- In Java: \textit{often} leads to runtime exception

When an assert fails, it generates an Exception. Other failures also generate Exceptions.

Use Exceptions to Remove Preconditions

```java
public static int biggest (int [ ] a)
// REQUIRES: a has at least one element
// EFFECTS: Returns the value biggest
// element of a.
```

```java
public static int biggest (int [ ] a)
throws NoElementException
// REQUIRES: true
// EFFECTS: If a has at least one element, returns the // value biggest element of a. Otherwise, throws // NoElementException.
```

Using Biggest with Requires

```java
public static void main(String[ ] args) {
    int [ ] a = new int [0];
    System.out.println("Biggest: " + biggest(a));
    ...
    Exception in thread "main"
    java.lang.ArrayIndexOutOfBoundsException: 0
    at ps3.TestClass.biggest(TestClass.java:6)
    at ps3.TestClass.main(TestClass.java:37)
```
### Implementation

```java
public static int biggest (int[] a) {
    int res = a[0];
    for (int i = 1; i < a.length; i++) {
        if (a[i] > res) res = a[i];
    }
    return res;
}
```

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 0
at ps3.TestClass.biggest(TestClass.java:6)
at ps3.TestClass.main(TestClass.java:37)
```

### Using Biggest with Exception

```java
public static int biggest (int[] a) throws NoElementException
// REQUIRES: true
// EFFECTS: If a has at least one element, returns the value biggest element of a. Otherwise, throws NoElementException.
```

```
public static main(String[] args) {
    int[] x = new int[0];
    System.out.println("Biggest: " + biggest(x));
}
```

### Catching Exceptions

```java
public static int biggest (int[] a) throws NoElementException
// EFFECTS: If a has at least one element, returns the value biggest element of a. Otherwise, throws NoElementException.
```

```
try {
    System.out.println("Biggest: " + biggest(x));
} catch (NoElementException e) {
    System.err.println("No element exception: " + e);
}
```

### Throwing Exceptions

```java
public static int biggest (int[] a) throws NoElementException {
    if (a == null || a.length == 0) {
        throw new NoElementException();
    }
    int res = a[0];
    for (int i = 1; i < a.length; i++) {
        if (a[i] > res) res = a[i];
    }
    return res;
}
```

### Exceptions are Objects

```
java.lang.Object
class NoElementException extends Exception {}
```

```
java.lang.Throwable
java.lang.Exception
ps2.NoElementException
```

We will cover subtyping and inheritance soon.

### public Document(String fname, int window)

**REQUIRES** fname is the pathname for a readable file

**EFFECTS** Creates a new document from the file identified by fname using window size window.

```java
public Document(String fname, int window) throws FileNotFoundException
```

```
public Document(String fname, int window) throws FileNotFoundException
```

**EFFECTS** If fname is a readable file, creates a new document from that file using window size window. Otherwise, throws FileNotFoundException.

```
```
Using Document

```java
LabeledGraph g = new LabeledGraph();
Document d;
try {
    d = new Document(file, window);
g.addNode(file);
} catch (FileNotFoundException fnfe) {
    System.err.println("Error: cannot open file: " + file + " [" + fnfe + "]");
} catch (DuplicateNodeException e) {
    System.err.println("Error: duplicate file: " + file);
}
```

Mantra

**Be Assertive!**
Use assertions judiciously

**Exception Exceptionally**
Use exceptions to deal with exceptional circumstances
Handling exceptions is tricky: code can jump from anywhere inside to the catch handler!

Charge

**Next class:** designing and using exceptions exceptionally

**Reading:** finish Chapter 5 and Chapter 10

“Surprise” quiz possible on Tuesday

**Problem Set 3:** Designing and Implementing Data Abstractions
will be posted by tomorrow, due Sept 21