cs2220: Engineering Software

Class 6: Defensive Programming

Fall 2010
University of Virginia
David Evans

Menu

Recap Validation

Hopelessness of both testing and analysis!

Defensive Programming

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?

{ null, [10], [1,2,3,4] }

```
public static int [] histogram (int [] a)
// unspecified
{
  int maxval = 0;
  for (int i = 0; i < a.length; i++) {
    if (a[i] > maxval) {
      maxval = a[i];
    }
  }
  int histo [] = new int [maxval + 1];
  for (int i = 0; i < a.length; i++) {
    histo[a[i]]++;
  }
  return histo;
}
```

NullPointerException

How many paths?

Arrays are bounded in java: maximum size is 2^{31}-1

First loop:

$$1 + 2 + 2^2 + \ldots + 2^{2^{31}-1}$$

Second loop: path completely determined by first loop

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:

$$\frac{\text{number of statements executed on at least one test}}{\text{number of statements in program}}$$

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 — Modularity, Decoupling

Design for Analyzability — Modularity, Narrow Interfaces

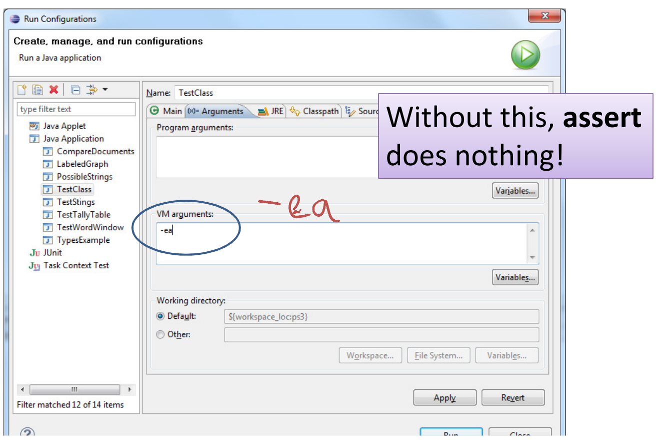
Programming Defensively

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 **AssertionException** thrown. If there is an optional stringExpression, it is evaluated (and converted to a String) and included in the AssertionException.

Enabling Assertions



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)
```

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 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));
}
```

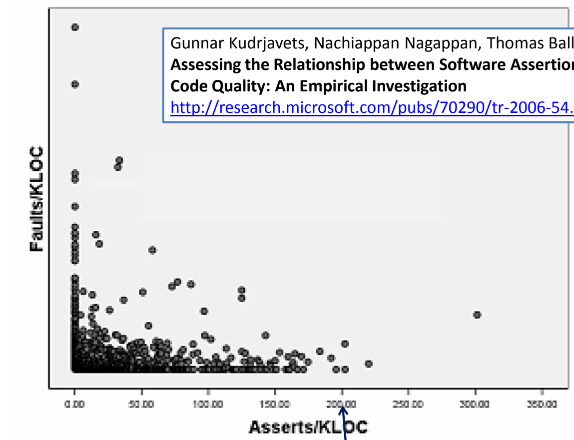
```
0.75
Exception in thread "main" java.lang.StackOverflowError
        at ps3.TestClass.divide(TestClass.java:6)
        at ps3.TestClass.divide(TestClass.java:6)
        at ps3.TestClass.divide(TestClass.java:6)
        at ps3.TestClass.divide(TestClass.java:6)
        at ps3.TestClass.divide(TestClass.java:6)
        ...
```

Where should we use **assert**?

```
public static int [] histogram (int [] a)
{   assert a != null;
  int maxval = 0;
  for (int i = 0; i < a.length; i++) {
    if (a[i] > maxval) {
      maxval = a[i];
    }
  }
  int histo [] = new int [maxval + 1];
  for (int i = 0; i < a.length; i++) {
    histo[a[i]]++;
  }
  return histo;   assert a[i] <= maxval
}   assert a[i] < histo.length
assert a[i] >= 0 : "Negative!";
```

1. To give useful debugging information **when a 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



Gunnar Kudrjavets, Nachiappan Nagappan, Thomas Ball.
Assessing the Relationship between Software Assertions and Code Quality: An Empirical Investigation
http://research.microsoft.com/pubs/70290/tr-2006-54.pdf

200 assertions per 1000 lines of code

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

Violating Requires

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

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

Use **Exceptions** to **Remove Preconditions**

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

⬇

```
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

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

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

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

Implementation

```
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.ArrayIndexOutOfBoundsE
at ps3.TestClass.biggest(T
at ps3.TestClass.main(Tes

```
public static int biggest (int [] a) {
  assert a != null && a.length > 0;
  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.AssertionError**
at ps3.TestClass.biggest(TestClass.java:9)
at ps3.TestClass.main(TestClass.java:46)

Using Biggest with Exception

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

TestClass.java:line 41 Unhandled exception type NoElementException

This is a **compile**-time error: you cannot even run this code.

Catching Exceptions

```
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.
```

Statement ::= *CatchStatement*
CatchStatement ::= **try** *Block Handler* OptFinally*
Handler ::= **catch** **(***ExceptionType Var***)** *Block*
OptFinally ::= **finally** *Block* | ε
Block ::= **{** *Statement** **}**

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

Throwing Exceptions

```
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;
}
```

What is **NoElementException**?

Exceptions are Objects

class NoElementException extends Exception { }

java.lang.Object
↑
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.

⇓

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

```
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 **Assert**ive!

Use **assert**ions 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