CS 130 SOFTWARE ENGINEERING

TESTING

Professor Miryung Kim
UCLA Computer Science
Based on Materials from Miryung Kim

ANNOUNCEMENT

- Symbolic Execution article is difficult to understand – therefore it's optional to read and not required
- Please also refer to Handout-TestingClassActivity.docx in CCLE.
- The will be also the scanned handwritten notes.

AGENDA

- Part 1: Testing adequacy criteria and Junit
- Part 2: The number of paths and Infeasible paths
- Part 3: Symbolic execution Test Generation
- Part 4: Regression test selection

MIDTERM DISTRIBUTION

- High: 99.5
- Average: 82.47
- Median: 84.75
- Low: 42.5

- Please see Midterm-Fall2016-Rubric-TAAnnotation.pdf in Week 6 of CCLE.

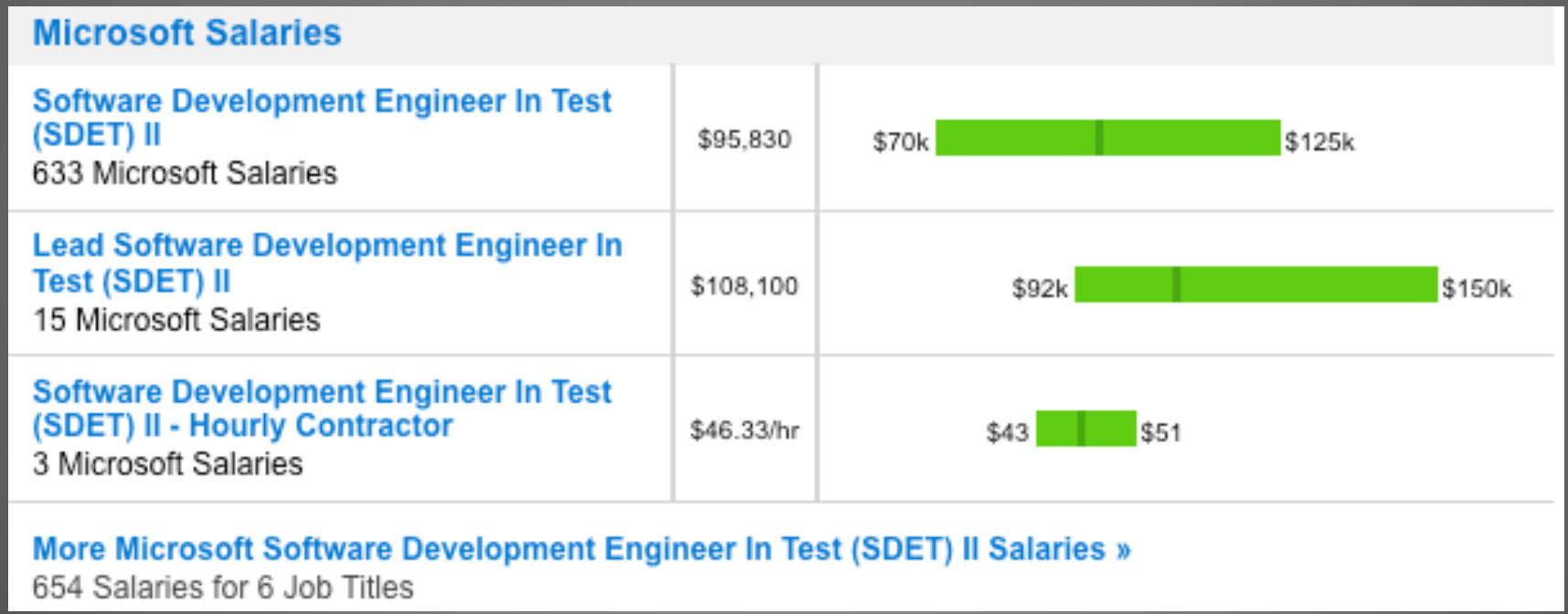
MID-QUARTER COURSE SURVEY

- This is a two page survey.
- It is "anonymous."
- Be specific about how we can improve your learning experience.
  - *"What in this course has helped you the most or do you like the best?"*
  - *"What has interfered with your learning in the course?"*
  - *"What changes would you make if you were the instructor to avoid the problems you've mentioned above"*

TESTING OVERVIEW

TESTING

- Testing is the most popular quality-assurance activity.
- Software development engineer in test (SDET)



TESTING

- A practice supported by a wealth of industrial and academic research and by commercial experience.
- Testing and code review/inspection are the most common quality assurance methods.

TESTING AND DEBUGGING

- Testing is a means of detecting/ revealing errors.
- Debugging is a means of diagnosing and correcting the root causes of errors that have already been detected.

KINDS OF TESTING

- **Unit testing** is the execution of a complete class, routine, or small program or team of programmers.
- **Component testing** is the execution of a class, package, small program, or other program element
- **Integration testing** is the combined execution of two or more classes, packages, components, or subsystems.

- **System testing** is the execution of the software in its final configuration, including integration with other software and hardware systems.
- **Regression testing** is the repetition of previously executed test cases for the purpose of finding defects.

ANOTHER CLASSIFICATION OF TESTING

- **Black-box** testing refers to tests in which the test cannot see the inner workings of the item being tested.
- **White-box** testing refers to tests in which the tester is aware of the inner workings of the item being tested.

DEVELOPER TESTING IN SOFTWARE QUALITY

- Testing's goal runs counter to the goals of other development activities.
- Testing can never completely prove the absence of errors.
- Testing by itself does not improve software quality.
- Testing requires you to assume that you will find errors in your code.

LIMITATIONS OF DEVELOPER TESTING

- Developer tends to have an optimistic view of test coverage.
- Developer tends to skip more sophisticated kinds of test coverage.

TEST ADEQUACY CRITERIA

WHY TEST ADEQUACY CRITERIA?

- Problem 1. Sometimes developers write not enough tests.
- Problem 2. Sometimes they write too much redundant tests, causing quality assurance overhead.
- Problem 3. During software evolution, we don't have a time to retest all tests again. Identifying relevant tests (relevant to code change) is hard.

```
 * Copyright (c) 2004-2006 Codign Software, LLC.
 *
 * All rights reserved. This program and the accompanying materials are made
 * available under the terms of the Eclipse Public License v1.0 which
 * accompanies this distribution, and is available at
 * http://www.eclipse.org/legal/epl-v10.html
 *
 ******************************************************************************/


package com.codign.sample.pathexample;

public class PathExample {

    public int returnInput(int x, boolean condition1,
                                  boolean condition2,
                                  boolean condition3) {
        if (condition1) {
            x++;
        }
        if (condition2) {
            x--;
        }
        if (condition3) {
            x=x;
        }
        return x;
    }
}
```

TEST COVERAGE

- Statement coverage
  - How many statements are exercised by tests?
- Branch coverage
  - How many of possible branch evaluations are exercised by tests?
- Path coverage
  - How many of possible paths are exercised by tests?

COVERAGE CRITERIA

- Statement coverage: Has each statement been executed?
- Branch coverage: Has each control structure evaluated both true and false?
- Path coverage: Has every possible route been executed?

BRANCH AND PATH COVERAGE

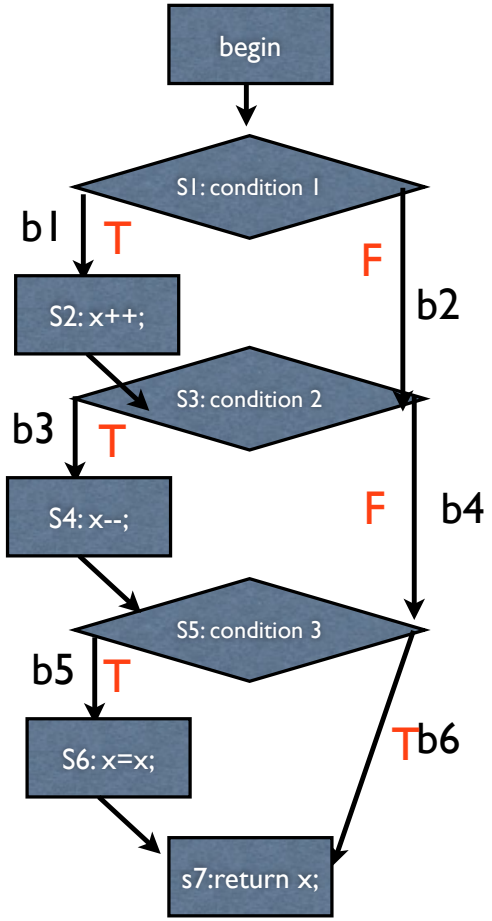
```
 * Copyright (c) 2004-2006 Codign Software, LLC.
 *
 * All rights reserved. This program and the accompanying materials are made
 * available under the terms of the Eclipse Public License v1.0 which
 * accompanies this distribution, and is available at
 * http://www.eclipse.org/legal/epl-v10.html
 *
 ******************************************************************************/


package com.codign.sample.pathexample;

public class PathExample {

    public int returnInput(int x, boolean condition1,
                                  boolean condition2,
                                  boolean condition3) {
        if (condition1) {
            x++;
        }
        if (condition2) {
            x--;
        }
        if (condition3) {
            x=x;
        }
        return x;
    }
}
```

Control Flow Graph



Fill out the following code coverage table by running the program with the following inputs

input	exercised statements	exercised branches	exercised paths
(cond1=true, cond2=true, cond3=true)	s1, s2, s3, s4, s5, s6, s7	b1, b3, b5	[b1, b3, b5]
Coverage	100%	50%	12.5%
(cond1=false, cond2=false, cond3=false)	s1,s3,s5,s7	B2, b4, `b6	[b2, b4, b6]
Coverage	100%	100%	25%
(cond1=false, cond2=true, cond3=true)	S1, s3, s4, s5, s6, s7	B2, b3, b5	[B2, b3, b5]
Coverage	100%	100%	37.5%

- Why we did not care about X?
- What is the goal of writing tests in this case?
  - There are no fixed goals.
  - One way of ensuring test adequacy is to increase code coverage.
  - Among many code coverage notations, we are going to focus on "branch / path" coverage which are based on "Control Flow Graph"
  - The reason that we did not care about int X, is that X was never used as an argument to the control predicates.

THINK PAIR SHARE 1

- After execution of each statement in the main program, what is a cumulative statement, branch and path coverage respectively?

```
   public static int
returnInput(int x, boolean
condition1, boolean
condition2,
         boolean condition3)
{
      if (condition1) {
         x++;
      }
      if (condition2) {
         x--;
      }
      if (condition3) {
         x = x;
      }
      return x;
```

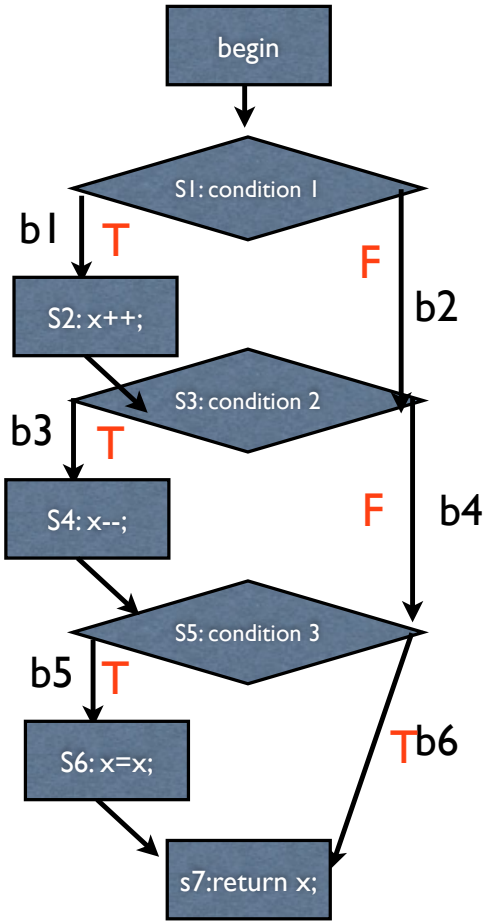
```
   public static void main
(String args[]){
      returnInput(3,true, true,
true);

      returnInput(5,false,
false, false);

      returnInput(2,false,
true, true);


}
```

Control Flow Graph



Fill out the following code coverage table by running the program with the following inputs

input	exercised statements	exercised branches	exercised paths
(cond1=true, cond2=true, cond3=true)	s1, s2, s3, s4, s5, s6, s7	b1, b3, b5	[b1, b3, b5]
Coverage			
(cond1=false, cond2=false, cond3=false)			
Coverage			
(cond1=false, cond2=true, cond3=true)			
Coverage			

BRANCH COVERAGE RECAP

- Branch coverage is measured w.r.t whether the branch takes true and false side.
- Suppose that we have a simple program
- if (x>1) x++ else x--;
- T1: x=2 makes the branch b to evaluate to T
- T2: x=0 makes the branch b to evaluate to F.
- So when we have T1 only, it's 50% in terms of branch coverage, while when we have T1 and T2 we have 100%.
- Select x=3, is this necessary?

THINK PAIR SHARE 1

- Now consider a program with three if-statements in a row.

```
if (x>1) x++ else x-- // let's call the branch b1
if (y>2) y:=0 else y:=1 // let's call the branch b2
if (z>3) z:=0 else z:=2 //  let's call the branch b3
```

X	Y	Z
0 (F)	0 (F)	0 (F)
2 (T)	3 (T)	4 (T)

THINK PAIR SHARE 1

- Now consider a program with three if-statements in a row.

```
if (x>1) x++ else x-- // let's call the branch b1
if (y>2) y:=0 else y:=1 // let's call the branch b2
if (z>3) z:=0 else z:=2 //  let's call the branch b3
```

- Branch coverage means we want to make a set of inputs where b1 to T, b1 to F, b2 to T, b2 to F, b3 to T, and b3 to F happen at least once.
- T1 (2,3,4) makes b1-T, b2-T, b3-T, leading to 50% branch coverage, as it covers 3/6.
- T2 (0,1,2) makes b1-F, b2-F, b3-F, adding another 50% branch coverage as it covers the other 3 out of 6. Having T1 and T2 together makes the branch coverage 100%.

THINK PAIR SHARE 2

- After execution of each statement in the main program, what is a cumulative statement, branch and path coverage respectively?
- 1. Draw a control flow graph for the above complexfun program.
- 2. What is the maximum number of paths for the above complexfun program?

```
public static int complexfun
(int array[], int k) {
      int value = 0;
      for (int i=0; i<2; i++)
{
         int a = array[i];
         if (a > k) {
            value = value+a;
         }else {
            value = value-a;
         }
      }
      return value;
   }
```

```
public static void main (String
args[]){
      int a[] ={3,5,7};
      complexfun(a, 10);
      int b[] = {5, 6, 9, 11,
15};
      complexfun(b, 4);
      int c[] = {7, 2, 1, 2, 5,
6};
      complexfun(c, 4);
      }
```

THINK PAIR SHARE 3

- Which of the three coverages among path, branch, and statement is the **most restrictive notion** of coverage (harder to achieve)?
  - Path coverage

THINK PAIR SHARE 4

- What is the number of paths if there are n branch executions and there are no infeasible paths?
- 2^n

THINK PAIR SHARE 5

- If there are k branch executions in the loop that iterates n times, how many paths do you have?
- n*k
- 2^(nk) if there are no infeasible path, and **the later branch executions are independent from early executions.**

JUNIT

JUNIT

- automated unit testing framework
- provides the required environment for the component
- executes the individual services of the component
- compared the observed program state with the expected program state
- reports any deviation from the expectations
- does all of this automatically

VALIDATING PROGRAM STATES

- The main tool of component test is the comparison of the observed state with the expected state.
- assertions
- assert in JAVA (JDK 1.4 and later)

`assert(b)`

- If b is true, nothing happens---the assertion passes.
- If b is false, a runtime error occurs.
- C and C++, similar to executing abort()
- Java, raise AssertionError exception

EXAMPLE: COMPARISON OF RATIONAL NUMBERS

- Identity 1/3 = 1/3?
- Different representations 2/6 = 1/3?
- Integers 3/3 = 1?
- Nonequality 1/3 != 2/3?

EXAMPLE

```
class RationalAssert {
    public static void main(String args[]){
      assert new Rational(1,3).equals(new
      Rational(1,3));
      assert new Rational(2,6).equals(new
      Rational(1,3));
      assert new Rational(3,3).equals(new
      Rational(1,1));
      assert !new Rational(2,3).equals(new
      Rational(1,3));
    }
 }
```

EXECUTION

```
$ javac -source 1.4 RationalAssert.java

$ java -ea RationalAssert
Exception in thread “main”
java.lang.AssertionError at
RationalAssert.main(RationalAssert.java:
3)

$
```

THINK-PAIR-SHARE

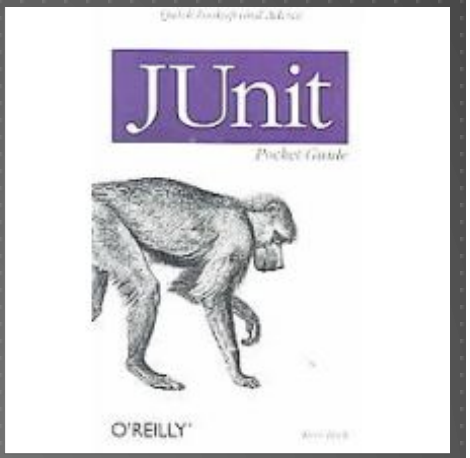
- What are the disadvantages of using asserts directly for testing?

EXECUTING COMPONENT TESTS

- If a test fails, the subsequent test cases are no longer executed
- One should be able to run tests individually, independent of other test cases
- One should be able to group tests into test suites
- One should be able to grasp immediately whether tests have failed and if so, which ones.

JUNIT

- JAVA unit test
- developed by Kent Beck and Erich Gamma



JUNIT TEST CASES

- Each test case is realized by its own class derived from JUNIT `TestCase` class
- Each test of the test case is realized by its own method whose name starts with test...
- `assertTrue()` method inherited from TestCase has the same meaning as `assert`.

JUNIT TESTCASE EXAMPLE

```
import junit.framework.*;
public class RationalTest extends TestCase {
// Create new test
     public RationalTest(String name) {
        super(name);
     }
     public void testEquality() {
        assertEquals(new Rational(1,3), new Rational(1,3));
        assertEquals(new Rational(2,6), new Rational(1,3));
        assertEquals(new Rational(3,3), new Rational(1,1));
        assertFalse(new Rational(2,3).equals(new Rational(1,3)));
     }
 }
 // Invoke GUI
 public static void main(String args[]) {
     String[] testCaseName = {RationalTest.class.getName()};
     junit.swingui.TestRunner.main(testcaseName);
 }
```

RUNNING JUNIT TESTCASE

- `javac -classpath .:wherever/junit.jar RationalTest.java`
- `java -classpath .:wherever/junit.jar RationalTest`

SETTING UP FIXTURE

- Test frequently need some fixture to execute
- Configuration files that must be read and processed
- External resources that must be requested and set up
- Services of other components that must be initialized

SETTING UP FIXTURE

- **Setting up:** the method `setUp()` is called before each test of the class
- **Tearing down:** the method `tearDown()` is called after each test. It is used for releasing fixture.

TEST FIXTURE

```
public class RationalTest extends TestCase {
     private Rational a_third;

     // Set up fixture
     // Called before each testXXX() method
     protected void setUp() {
        a_third = new Rational(1,3);
     }
     // Tear down fixture
     protected void tearDown() {
        a_third = null;
     }
     ...
}
```

ORGANIZING TEST CASES

- If multiple test cases are to be executed, these multiple test cases can be grouped into a test suite.
- A test suite is a container for multiple test cases.

TEST SUITE

```
TestSuite suite= new TestSuite();
suite.addTest(new RationalTest("testEquality"));
suite.addTest(new RationalTest("testNonEquality");;
TestResult result = suite.run();


TestSuite suite= new TestSuite(RationalTest.class);
TestResult result = suite.run();


public class RationalTest extends TestCase {

      public static Test suite() {
         TestSuite suite = new TestSuite(RationalTest.class);
         return suite;
      }
 }
```

RECAP 1.

- Testing is the most popular quality-assurance method in software engineering.
- Test adequacy criteria helps developers to assess how adequate your tests are.
- Path coverage is a much stricter test adequacy criterion than branch and statement coverage criteria.
- For an arbitrary program, it is very difficult to achieve 100% path coverage (in fact, the problem is undecidable).

RECAP 2.

- To write tests, the most important language tools are assertions.
- In JUnit, methods represent tests, and classes represent test cases; test suite groups multiple tests.

PREVIEW 1

- Please go through Junit Tutorial on line
- Symbolic execution paper is an optional reading for those who are super interested and eager *(but I completely understand that the article is hard to read so no worries.)*

PREVIEW 2

- We will discuss regression test selection, prioritization, and augmentation methods.
- We will have a class activity on test case generation.

QUESTIONS?