CSED332: Software Design Methods

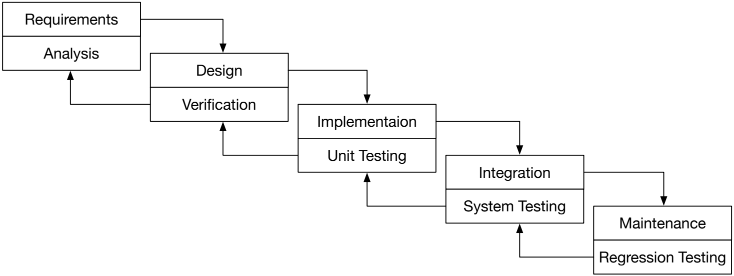
Lecture 2: Testing

Kyungmin Bae

Department of Computer Science and Engineering
POSTECH

Last Lecture

- Software process (model)

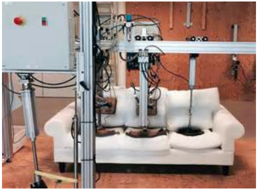


- Software configuration management
  - version control, building, change management, releasing

This Lecture: Testing

- Why test and what is a test?
- How to use JUnit to write unit tests?
- How do we write high-quality tests?

Testing?



NOKIA

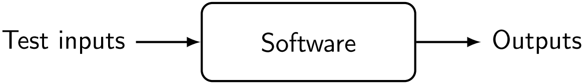
2011 NISSAN LEAF
INSURANCE INSTITUTE FOR HIGHWAY SAFETY
CEF1104

Why Test?

- Improve quality: find faults
- Measure quality
  - Prove there are no faults? (Is it possible?)
  - Determine if software is ready to be released
  - Determine what to work on
  - See if you made a mistake
- Learn the software

What is a Test?

- Run code for some inputs, and check outputs



- Some terminology
  - Test case: test inputs, values, data
  - Test suite: a set of test cases
  - Test oracle: mechanism for deciding if test has passed/failed.
- Tests can document faults and code

Example: TriTyp

- Input: three integers for the lengths of the sides of a triangle
- Output: the type of the triangle
  1. if triangle is scalene (no sides are equal)
  2. if triangle is isosceles (two sides are equal)
  3. if triangle is equilateral (all sides are equal)
  4. if not a triangle

- Test cases?
  - $(2, 2, 2)$,
  - $(2, 2, 3)$,
  - $(2, 2, 4)$,
  - $(2, 3, 4)$,
  - ...

- Test oracle:
  - $(2, 2, 2) : 3$,
  - $(2, 2, 3) : 2$,
  - $(2, 2, 4) : 4$,
  - $(2, 3, 4) : 1$,
  - ...

Mistake, Fault, Error, Failure

1. Programmer makes a mistake
2. Fault (defect, bug) appears in the program.
3. Fault remains undetected during testing (running test inputs).
4. The program fails (based on test oracles) during execution.

- Error (ISO/IEC/IEEE 24765)

  difference between computed, observed, or measured value/condition and true, specified, or theoretically correct value/condition

- What does Bug mean in "Bug Report"?

Where is Fault, Error, or Failure?

- Doubling the balance and then plus 10

```
int calAmount () {
    int ret = balance * 3;
    ret = ret + 10;
    return ret;
}
```

- Test input?   Test oracle?

```
void testCalAmount() {
   Account a = new Account();
   Account.setBalance(1);
   int amount = Account.calAmount();
   assertTrue(amount == 12);
}
```

Where is Fault, Error, or Failure?

- Not allow withdrawal when there is a balance of 100 or less

```
boolean doWithdraw(int amount) {
     if (Balance<100)
       return false;
     else
       return withDraw(amount);
}
```

- Test input?   Test oracle?

```
void testWithDraw() {
   Account a = new Account();
   Account.setBalance(100);
   boolean success = Account.doWithdraw(10);
   assertTrue(!success);
}
```

Who Should Test?

- Developer?
- Separate “quality assurance” group?
- Programmer?
- User?
- ~~Someone with a degree in “testing”?~~

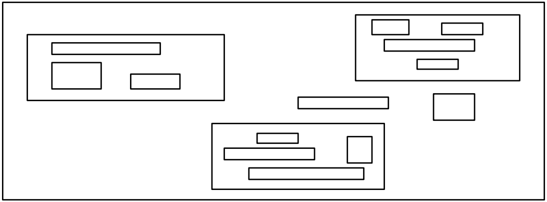
What Kind of Tests?

- Programmer tests / non-programmer tests
- Developer / Tester
- Unit tests / Integration tests / System tests / acceptance tests
- Automated tests / Manual tests
- Regression tests / Exploratory testing
- . . .

Types of Testing: Code or Functionality?

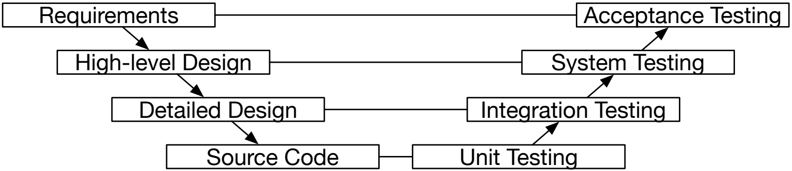
- Black Box Testing (you know the functionality)
  - derive tests from external descriptions of the software
  - test functionality and interface by observing its external behavior
  - requirements, specifications, design, . . .
- White Box Testing (you know the code)
  - derive tests from the source code of the software
  - examination of procedural level in detail
  - crashes, out of bounds, file not closed, uncaught exceptions, . . .

Types of Testing: Top or Bottom?



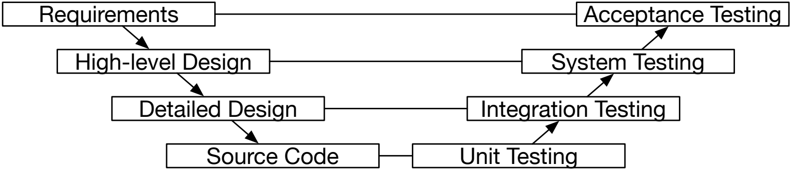
- Top-Down Testing
  - test the main procedure first, then
  - go down through procedures it calls, and so on
- Bottom-Up Testing
  - test procedures that make no calls first (the leaves in the tree), and
  - move up to the root.

Types of Testing: Units, Modules, or Systems? (1)



- Unit testing
  - test individual (or groups of) units
  - performed by programmers
  - white, bottom-up
- Integration testing
  - test the interaction between components or units
  - performed by programmers
  - white (black for modules), bottom-up (plus unit testing)

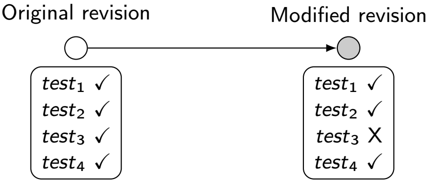
Types of Testing: Units, Modules, or Systems? (2)



- System testing
  - test a complete system with respect to its requirements
  - usually performed by external test group
  - black, top-down
- Acceptance testing
  - determine if system satisfies acceptance criteria by customer
  - performed by customer representative (alpha/beta testing)
  - black, top-down

Types of Testing: New Bugs or Old Bugs?

- Regression testing
  - make sure that everything that worked in the past still works



Original revision

test$_1$ ✓
test$_2$ ✓
test$_3$ ✓
test$_4$ ✓

Modified revision

test$_1$ ✓
test$_2$ ✓
test$_3$ X
test$_4$ ✓

  - rerun whenever changed
  - add to regression tests if new bug found
- Exploratory testing
  - look for new bugs (often contrast to scripted testing)

Types of Testing: Manual or Automated?

- Manual testing
  - perform lists of "instructions"
  - good for exploratory and testing GUI
- Automated testing
  - test = program (code or script)
  - created by tool that "records" manual tests

Automated Testing: Trade-Off

- Benefits of automated testing
  - reduces human errors, and variance in test quality
  - significantly reduces the cost of regression testing
  - real projects often have more test code than production code
- Limitation of automated testing
  - some tests cannot be automated: usability testing
  - can break if environment changes: maintaining tests is hard
  - when too expensive to automate and maintain automated

More Types of Testing

- Fault-based testing
  - look for common faults
- Scenario-based testing
  - derive tests based on user stories
- Model-based testing
  - derive tests from model of the software
- Random testing
  - see if random input crashes the program or agrees with the oracle
- . . .

Test Automation

Test Automation Framework

- Assumptions, concepts and tools that support test automation
  - test = program (code or script)
- xUnit framework
  - Programmer's testing tools
  - Unit testing, but also integration testing and functional testing
  - Regression testing
  - Test-first design
  - Each code unit requires several tests

What is JUnit

- Open source Java testing framework
  - used to write and run repeatable automated tests
  - structure for writing test drivers
  - widely used in industry
- JUnit features include
  - assertions for testing expected results
  - test features for sharing common test data
  - test suites for easily organizing and running tests
  - graphical and textual test runners
- JUnit can be used
  - stand alone Java programs (from the command line)
  - within an IDE such as Eclipse and IntelliJ

JUnit Tests

- JUnit can be used to test
  - entire or part of an object
  - interaction between several objects
- Primarily for unit and integration testing, not system testing
- Each test is embedded into one test method
- Test class
  - contains one or more test methods
  - methods to set up/update the state before/after each/all test(s)
- Get started at `https://junit.org/junit5`

Writing Tests for JUnit

- Need to use the methods of org.junit.jupiter.api.Assertions class
  - javadoc gives a complete description of its capabilities
  - all of the methods return void
- Test method
  - checks a condition (assertion)
  - reports to the test runner whether the test failed or succeeded
- Test runner
  - use the result to report to the user (in command line mode)
  - update the display (in an IDE)
- A few representative methods of org.junit.jupiter.api.Assertions
  - assertTrue (boolean), assertTrue (boolean, String)
  - assertEquals (Object, Object)

Example JUnit Test Case (1)

```java
public class Calc {
   static public int add (int a, int b) {
      return a + b;
   }
}
```

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class CalcTest {
    @Test
    public void testAdd() {
        assertEquals(
            5,              // extected result
            Calc.add (2, 3)     // the test
        );
    }
}
```

- We use JUnit 5 (released on September 10, 2017)

Sample Assertions

- `static void assertEquals(Object expected, Object actual)`
  asserts that two objects are equal

- `static void assertEquals(long expected, long actual)`
  asserts that two longs are equal

- `static void assertEquals(double e, double a, double delta)`
  asserts that two doubles are equal to within a positive delta

- For a complete list, see
  `https://junit.org/junit5/docs/current/api/org/junit/jupiter/api/Assertions.html`

JUnit Test Fixtures

- A test fixture is the state of the test
  - Objects and variables that are used by more than one test
  - Initializations (prefix values)
  - Reset values (postfix values)
- Different tests can use the objects without sharing the state
- Objects used in test fixtures declared as instance variables
- They should be initialized in a @BeforeEach method
  - JUnit runs them before every @Test method
- Can be deallocated or reset in an @AfterEach method
  - JUnit runs them after every @Test method

Example JUnit Test Case (2-1)

```java
import java.util.Iterator;
import java.util.List;

public class Min {
    /**
     * Returns the mininum element in a list
     * @param list Comparable list of elements to search
     * @return the minimum element in the list
     * @throws NullPointerException if list is null or any elements are null
     * @throws ClassCastException if list elements are not mutually comparable
     * @throws IllegalArgumentException if list is empty
     */
    public static <T extends Comparable<? super T>> T
    min (List<? extends T> list) {
        if (list.size() == 0) {
            throw new IllegalArgumentException ("Min.min");
        }
        Iterator<? extends T> itr = list.iterator();
        T result = itr.next();
        if (result == null)
            throw new NullPointerException ("Min.min");
        while (itr.hasNext()) {
            T comp = itr.next();
            if (comp.compareTo (result) < 0)
                result = comp;
        }
       return result;
    }
}
```

Example JUnit Test Case (2-2)

- Standard imports for all JUnit classes:

```
import java.util.*;
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;
```

- Test fixture and pre-test setup method (prefix):

```
private List<String> list;   // Test fixture

@BeforeEach
public void setUp() {    // called before each test method
    list = new ArrayList<String>();
}
```

- Post test teardown method (postfix)

```
@AfterEach
public void tearDown() {  // called after each test method
    list = null;
}
```

Example JUnit Test Case (2-3)

- JUnit test cases using the fixture

```
@Test
public void testSingleElement() {
    list.add("cat");
    Object obj = Min.min(list);
    assertTrue(obj.equals("cat"), "Single Element");
}
```

```
@Test
public void testDoubleElement() {
    list.add("dog");
    list.add("cat");
    Object obj = Min.min(list);
    assertTrue(obj.equals("cat"), "Double Element");
}
```

Example JUnit Test Case (2-4)

- NullPointerException test:

```
@Test
public void testForNullElement() {
    assertThrows(NullPointerException.class, () -> {
        list.add(null);
        Min.min(list);
    });
}
```

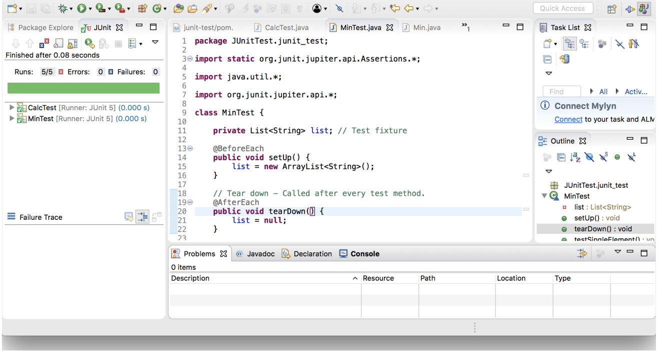
  - notice “lambdas” of the form *args* -> *code*

- Java generics don’t prevent clients from using raw types:

```
@Test
public void testMutuallyIncomparable() {
    assertThrows(ClassCastException.class, () -> {
        List list = new ArrayList();
        list.add("cat");
        list.add(1);
        Min.min(list);
    });
}
```

How to Run Tests

- JUnit provides test drivers
  - IDE (like Eclipse) exploits them to run JUnit test cases



Finished after 0.08 seconds

Runs: 5/5 Errors: 0 Failures: 0

CalcTest [Runner: JUnit 5] (0.000 s)
MinTest [Runner: JUnit 5] (0.000 s)

```java
package JUnitTest.junit_test;

import static org.junit.jupiter.api.Assertions.*;

import java.util.*;

import org.junit.jupiter.api.*;

class MinTest {

    private List<String> list; // Test fixture

    @BeforeEach
    public void setUp() {
        list = new ArrayList<String>();
    }

    // Tear down - Called after every test method.
    @AfterEach
    public void tearDown() {
        list = null;
    }
```

0 items

Description	Resource	Path	Location	Type

- Maven
- Console

Build Scaffolding for Incomplete Code

- Code (temporarily) written in order to unit test a program
- Stub (or Mock): a module to simulate components
  - not written yet, or interactive input

```
public void movePlayer(Player player, int diceValue) {
    player.setPosition(1);
}
```

- Driver: a software module used to invoke a module under test
  - provide test inputs, control/monitor execution, report test results

```
public void test() {
    movePlayer(new Player(0), 3);
}
```

JUnit Resources

- JUnit download and documentation:
  `http://www.junit.org/`
- JUnit Cookbook (JUnit 4):
  `http://junit.sourceforge.net/doc/cookbook/cookbook.htm`

Parameterized Unit Tests

- Problem: testing function with similar values
  - How to avoid test code bloat?
- Simple example: adding two numbers
  - Adding one pair of numbers is just like adding any other pair
  - only want to write one test
- Parameterized unit tests
  - same tests are run on each set of data parameters
  - data values are provided by various sources

Examples JUnit Test Case (3-1)

- @ValueSource for single parameters

```
@ParameterizedTest
@ValueSource(ints = { 1, 2, 3 })
public void testDouble(int input) {
    assertEquals(input * 2, Calc.add(input, input));
}
```

- @CsvSource for comma-separated values

```
@ParameterizedTest(name = "{index}: {0} + {1} = {2}")
@CsvSource({ "1, 1, 2", "2, 3, 5", "2, 4, 6" })
public void testMany(int a, int b, int sum) {
    assertTrue(sum == Calc.add(a, b), "Addition Test");
}
```

Examples JUnit Test Case (3-2)

- @MethodSource for designating a method to generate data

```java
@ParameterizedTest(name = "{index}: {0} + {1} = {2}")
@MethodSource("dataProvider")
public void testDynamic(int a, int b, int sum) {
    assertTrue(sum == Calc.add(a, b), "Addition Test");
}
```

```java
static Stream<Arguments> dataProvider() {
    Random rand = new Random();
    return Stream.generate(() -> {
        int a = rand.nextInt();
        int b = rand.nextInt();
        return Arguments.of(a, b, a + b);
    }).limit(100);
}
```

  - notice the use of Java “streams” and lambdas

Parameterized Test Patterns (1)

- Contract model: assume, arrange, act, assert
  - assumptions (preconditions) limit values appropriately
  - arrangements set all necessary preconditions and inputs.
  - action performs activity under scrutiny
  - assertions (postconditions) check result
- 4A (assume, arrange, act, assert): most common pattern

```
@ParameterizedTest
@MethodSource
public void removeThenAdd(Set<String> set, String str) {
   assumeTrue(set.contains (str));      // Assume
   Set<String> copy =
       new HashSet<String>(set);        // Arrange
   copy.remove (str);                   // Act
   copy.add (str);
   assertEquals (set, copy);            // Assert
}
```

Parameterized Test Patterns (2)

- Roundtrip: $f^{-1}(f(x))$ for all x, given API f

```java
@ParameterizedTest @MethodSource
public void toStringParseRoundtrip(int value) {
    String s = Integer.toString(value);
    int parsed = Integer.parseInt(s);
    assertEquals(value, parsed);
}
```

- Commutative diagram: $f(I(x)) = I(g(x))$ for all x, given two implementations f and g (with IO transformation I)

```java
static String multiply(String x, String y) { ... }
static int multiply(int x, int y) { ... }

@ParameterizedTest @MethodSource
void CommutativeDiagram1(int x, int y) {
    String z = multiply(Integer.toString(x),
        Integer.toString(y));
    assertEquals(Integer.toString(multiply(x, y)), z);
}
```

Test Coverage

TriTyp: Which Test Suite is Better?

Test Suite 1

- $(1, 1, 1)$
- $(2, 2, 2)$
- $(3, 3, 3)$
- $(4, 4, 4)$
- $(5, 5, 5)$
- $(6, 6, 6)$

Test Suite 2

- $(2, 2, 2)$
- $(2, 2, 3)$
- $(2, 2, 4)$
- $(2, 3, 2)$
- $(2, 3, 3)$
- $(2, 3, 4)$

Test Suite 3

- $(1, 1, 1)$
- $(1, 1, 2)$
- $(1, 1, 3)$
- $(1, 1, 4)$
- $\ldots$
- $(1, 1, 100000)$

TriTyp: What Kind of Test is Effective?

- Code (from Ammann & Offutt)

```
int Triang (int s1,
            int s2,
            int s3) {
   int c = 0;
   if (s1 <= 0 || s2 <= 0 ||
       s3 <= 0) {
      c = 4;
      return c;
   }
   if (s1 == s2)
      c = c + 1;
   if (s1 == s3)
      c = c + 2;
   if (s2 == s3)
      c = c + 3;
   if (c == 0) {
     if (s1+s2 <= s3 ||
         s2+s3 <= s1 ||
         s1+s3 <= s2)
        c = 4;
     else
        c = 1;
     return c;
   }
   if (c > 3)
      c = 3;
   else if (c == 1 && s1+s2 > s3)
      c = 2;
   else if (c == 2 && s1+s3 > s2)
      c = 2;
   else if (c == 3 && s2+s3 > s1)
      c = 2;
   else
      c = 4;
   return c;
}
```

- May want to cover all lines.

Test Coverage

- Test requirements
  - specific goals that must be satisfied during testing
  - e.g., each statement in source code
- Coverage
  - satisfaction of test requirements
  - measures the quality of tests
- Perfect coverage is impossible in general
  - too many (an infinite number of) inputs
  - infeasible test requirements: no test case to satisfy the requirement

Test Criteria (1)

- Test criteria
  - collection of rules that impose test requirements
  - e.g., to cover every statement
- Advantage
  - provide systemic ways to search the input space
  - automated tools: coverage analysis and test generation
- Different test criteria for different types of testing
  - too many test criteria?
  - what are the fundamental classes of test criteria?

Test Criteria (2)

- Define (abstract) model of the software
  - requirements (for acceptance testing)
  - designs (system and integration testing)
  - source code (integration and unit testing)
  - . . .
- Find test criteria for the model
  - requirement items (for acceptance testing)
  - elements in design (for system and integration testing)
  - modules (integration testing) or lines (unit testing) in code
  - . . .
- level of abstraction: black box or white box

Planning a Black Box Test Case

- Look at requirements or problem statements
  - JavaDoc (classes and methods), user interface, protocols, ...
- Test cases need to be
  - traceable to requirements
  - repeatable (very specific inputs and expected results)
- Make it simple
  - run the simplest test case first
  - plan the simplest test to test the exprected failure

Example: Planning a Black Box Test Case

Blue Marble Game

ID	Description	Expected Results	Actual Results	Use Case
3	Precondition: Game is in test mode, SimpleGameBoard is loaded, and game begins. ▶ Number of players: 2 ▶ Money for player 1: 1200 ▶ Money for player 2: 1200 ▶ Player 1 dice roll: 3	Player 1 is located at 'Hong Kong'		MOVE1
4	Precondition: Test case 3 has successfully completed ▶ Player 2 dice roll: 3	Player 1 is located at 'Hong Kong'. Player 2 is located at 'Hong Kong'.		MOVE2

Equivalence Partitioning

- Divide input domain into classes of equivalent values
  - inputs in the same class are expected to behave similarly
- Test requirements
  - at least one value chosen from each block
- Test criteria
  - how to choose effective subsets from partitions? (all or parts)
- Example: integer value
  - 0, 1, -1, biggest value, smallest value
  - criteria for ternary function: $5^3 = 125$ cases?, $5 * 3 = 15$ cases?

Example: TriTyp

- Partitioning 1 ($4 * 4 * 4 = 64$ partitions)

Characteristic	b_1	b_2	b_3	b_4
Relation of Side 1 to 0	$s_1 > 1$	$s_1 = 1$	$s_1 = 0$	$s_1 < 0$
Relation of Side 2 to 0	$s_2 > 1$	$s_2 = 1$	$s_2 = 0$	$s_2 < 0$
Relation of Side 3 to 0	$s_3 > 1$	$s_3 = 1$	$s_3 = 0$	$s_3 < 0$

- Partitioning 2

Characteristic	Blocks		
triangle = scalene	True	False	
triangle = isosceles	True	False	
triangle = equilateral	True	False	
triangle = invalid	True	False	
Relation between s_1 and s_2	$s_1 < s_2$	$s_1 = s_2$	$s_1 > s_2$
Relation between s_2 and s_3	$s_2 < s_3$	$s_2 = s_3$	$s_2 > s_3$
Relation between s_3 and s_1	$s_3 < s_1$	$s_3 = s_1$	$s_3 > s_1$

- constraints among characteristics:

  ($\text{equilateral} = T \rightarrow \text{isosceles} = T \wedge s_1 = s_2 \wedge s_2 = s_3 \wedge s_3 = s_1$), . . .

- how many "valid" partitions?

Equivalence Class Test Ideas

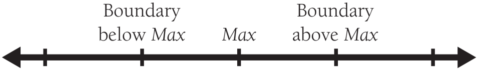
- Any object: the null pointer
- Strings: the empty string
- Collections
  - the empty collection, size 1, maximum size, . . .
- Sorting in ascending order
  - already sorted, has duplicates, has negative numbers
- Linked structures (trees, queues, . . .)
  - empty, minimal but non-empty, circular, . . .
- Equality comparison of objects
  - equal but not identical
  - different at lowest level but the same at upper

Dirty/Failure Test Cases

- Something cause division by zero
- Wrong input type
  - you are expecting an integer, they input a float
  - you are expecting a character, you get an integer
- Illogical path to access your functionality
- Program is aborted abruptly
- Input or output devices are unplugged

Boundary Analysis

- Errors tend to occur at the boundaries of domain
  - pick test values at the boundary of an equivalence class
  - test for error-checking by going beyond the boundary
- Three boundary cases



  - Range integer a to b: use a, b, $a-1$, $a+1$, $b-1$, $b+1$
  - Certain quantity q: use $q-1$, q, $q+1$

Combinatorial Testing (1)

- Choose effective subsets from partitions (and boundaries)
  - Q characteristics, each of which has B_i blocks
- Each choice
  - use one value from each block for each characteristic
  - $\max\{B_i\}_{i=1}^{Q}$ test requirements
  - e.g., 4 for TriTyp: $(2, 1, 0)$, $(1, 0, -1)$,$(0, -1, 2)$, $(-1, 2, 1)$
- All combinations
  - use all combinations of blocks from all characteristics
  - $\Pi_{i=1}^{Q} B_i$ test requirements
  - e.g., 64 for TriTyp in Partitioning 1

Combinatorial Testing (2)

- Pair-wise
  - use all pairs of blocks from all characteristics
  - $\leq (\max\{B_i\}_{i=1}^{Q})^2$ test requirements
  - Example: 16 for TriTyp in Partitioning 1

$$\begin{array}{rrrr}
(2,2,2), & (2,1,0), & (2,0,-1), & (2,-1,1) \\
(1,2,-1), & (1,1,1), & (1,0,2), & (1,-1,0) \\
(0,2,1), & (0,1,2), & (0,0,0), & (0,-1,-1) \\
(-1,2,0), & (-1,1,-1), & (-1,0,1), & (-1,-1,2)
\end{array}$$

- T-wise
  - extension of pair-wise from 2 to T
  - $\leq (\max\{B_i\}_{i=1}^{Q})^T$ test requirements
  - each choice if $T = 1$, pair-wise if $T = 2$, all comb. if $T = Q$

Combinatorial Testing Tool

- Reference
  - http://www.pairwise.org
- Example: PICT (https://github.com/microsoft/pict)
  - Equivalence classes (in trityp.txt)

```
s1: >1, =1, =0, <0
s2: >1, =1, =0, <0
s3: >1, =1, =0, <0
```

- Result

```
$ ./pict trityp.txt
s1      s2      s3
<0      <0      =1
=1      <0      =0
>1      =1      =1
=0      >1      >1
=0      =1      =0
<0      >1      =0
>1      >1      <0
<0      =1      <0
>1      =0      >1
=1      =0      <0
=1      >1      =1
=0      <0      <0
=1      =1      >1
=0      =0      =1
<0      =0      >1
>1      =0      =0
>1      <0      >1
```

Code Coverage for White Box Testing

- Make sure tests cover each part of program
  - statements, branches, paths, conditions, loops, . . .
- Many test requirements (code to be covered)
  - take a long time to write
- Measures the quality of tests
  - how much of the program do the tests cover?
- Code coverage tools for Java
  - Cobetura, JaCoCo, EclEmma, Quilt, NoUnit, InsECT, . . .

Devising a Prudent Set of Test Cases

- Equivalence Partitioning or Boundary Analysis
  - still applies
- A metric for assessing how good your test suite
  - structural code coverage
  - data flow coverage

Structural Coverage Criteria: Method Coverage

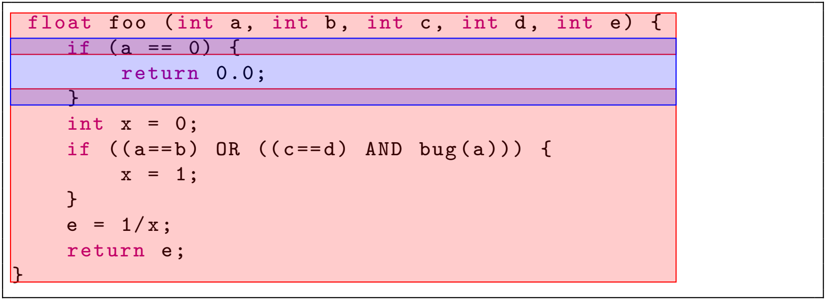
- All methods in all classes have been called

```
float foo (int a, int b, int c, int d, int e) {
    if (a == 0) {
        return 0.0;
    }
    int x = 0;
    if ((a==b) OR ((c==d) AND bug(a))) {
        x = 1;
    }
    e = 1/x;
    return e;
}
```

- Test cases
  - `foo(0, 0, 0, 0, 0) = 0.0`

Structural Coverage Criteria: Statement Coverage

- All lines in a method have been executed



```
float foo (int a, int b, int c, int d, int e) {
    if (a == 0) {
        return 0.0;
    }
    int x = 0;
    if ((a==b) OR ((c==d) AND bug(a))) {
        x = 1;
    }
    e = 1/x;
    return e;
}
```

- Test cases
  - `foo(1, 1, 1, 1, 1) = 0.0`
  - `foo(0, 0, 0, 0, 0) = 0.0`

Structural Coverage Criteria: Branch Coverage

- All predicates have been true and false (a.k.a. Decision Coverage)

```
1   float foo (int a, int b, int c, int d, int e) {
2       if (a == 0) {
3           return 0.0;
4        }
5       int x = 0;
6       if ((a==b) OR ((c==d) AND bug(a))) {
7           x = 1;
8       }
9       e = 1/x;
10      return e;
11  }
```

- Test cases

Line	Predicate	True	False
2	(a == 0)	foo(0,0,0,0,0)	foo(1,1,1,1,1)
6	(a==b) OR ((c==d) AND bug(a))	foo(1,1,1,1,1)	foo(1,2,1,2,1)

- Predicates have to be reachable.

Structural Coverage Criteria: Condition Coverage

- All sub-predicates have been true & false (a.k.a. Clause Coverage)

```
 1   float foo (int a, int b, int c, int d, int e) {
 2      if (a == 0) {
 3          return 0.0;
 4       }
 5      int x = 0;
 6      if ((a==b) OR ((c==d) AND bug(a))) {
 7          x = 1;
 8      }
 9      e = 1/x;
10      return e;
11  }
```

- Test cases

Line	Clause	True	False
2	(a == 0)	foo(0,0,0,0,0)	foo(1,2,1,2,1)
6	(a == b)	foo(1,1,1,1,1)	foo(1,2,1,2,1)
6	(c == d)	foo(1,1,1,1,1)	foo(1,2,1,2,1)
6	bug(a)	foo(x,2,1,1,1)	

- Sub-predicates have to be reachable.

Loops

- Write a test case such that you
  - do not go through the loop at all
  - go through the loop once
  - go through the loop twice
  - go through the loop many times
- Not possible to go through all paths with loops

Infeasible Coverage

- There are parts we cannot cover in practice.
- Example

```
int i;
for (i=0; i<100; i++) {
    printf("%d\n", i);
}
if (i == 200)
    panic("program internal error: 3");    // cannot reach
```

Data Flow Coverage

- Goal
  - finding "important" paths using interactions through data flow
  - ensuring that values are computed and used correctly
- Status of data
  - Def: value for variable is assigned
  - Use: variable's value is accessed
- Criterion: values by defs reach at least one, some, or all uses
  - bad value computation revealed only when it is used

Data Flow Coverage: All-Def Coverage

- Every def reaches some use

```
 1  int bar (bool c1, bool c2) {
 2      int x = 0;          // def
 3      if ( c1 )
 4          x = x - 1;      // def, use
 5      else
 6          x = 2;          // def
 7      if ( c2 )
 8          return x + 1;      // use
 9      else
10          return x - 1;      // use
11  }
```

- Test cases

Line	Def	Input
2	(x = 0)	bar(true,true)
4	(x = x - 1)	bar(true,true)
6	(x = 2)	bar(false,false)

Data Flow Coverage: All-Use Coverage

- Every def reaches all possible uses

```
 1  int bar (bool c1, bool c2) {
 2      int x = 0;          // def
 3      if ( c1 )
 4          x = x - 1;      // def, use
 5      else
 6          x = 2;          // def
 7      if ( c2 )
 8          return x + 1;     // use
 9      else
10          return x - 1;     // use
11  }
```

- Test cases

Def-Use	Input	Def-Use	Input
2-4	`bar(true,true)`	4-10	`bar(true,false)`
2-8	-	6-8	`bar(false,true)`
2-10	-	6-10	`bar(false,false)`
4-8	`bar(true,true)`		

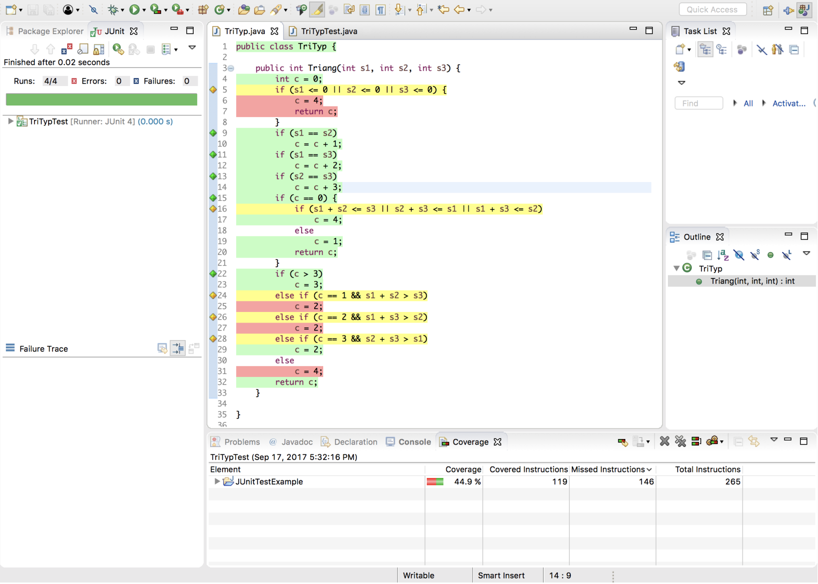
Relationship Among Coverage Criteria

- A criterion C_1 subsumes a criterion C_2
  - every test suite that satisfies criterion C_1 also satisfies C_2.
- Assumption
  - no exceptions/crashes occur during executions
  - all executions terminate
- Example
  - statement coverage subsumes method coverage
  - branch coverage subsumes statement coverage
  - all-use coverage subsumes all-def coverage
- Nonexample
  - branch coverage does not subsume condition coverage
  - condition coverage does not subsume branch coverage (why?)

Example: Code Coverage in Eclipse (1)

```java
public class TriTypTest {
    @Test public void testScalene() {
        assertEquals((new TriTyp()).Triang(2, 3, 4), 1);
    }
    @Test public void testIsosceles() {
        assertEquals((new TriTyp()).Triang(2, 3, 3), 2);
    }
    @Test public void testEquilateral() {
        assertEquals((new TriTyp()).Triang(2, 2, 2), 3);
    }
    @Test public void testNotTriangle() {
        assertEquals((new TriTyp()).Triang(1, 2, 3), 4);
    }
}
```

Example: Code Coverage in Eclipse (2)



Finished after 0.02 seconds

Runs: 4/4 Errors: 0 Failures: 0

TriTypTest [Runner: JUnit 4] (0.000 s)

Failure Trace

```java
public class TriTyp {

    public int Triang(int s1, int s2, int s3) {
        int c = 0;
        if (s1 <= 0 || s2 <= 0 || s3 <= 0) {
            c = 4;
            return c;
        }
        if (s1 == s2)
            c = c + 1;
        if (s1 == s3)
            c = c + 2;
        if (s2 == s3)
            c = c + 3;
        if (c == 0) {
            if (s1 + s2 <= s3 || s2 + s3 <= s1 || s1 + s3 <= s2)
                c = 4;
            else
                c = 1;
            return c;
        }
        if (c > 3)
            c = 3;
        else if (c == 1 && s1 + s2 > s3)
            c = 2;
        else if (c == 2 && s1 + s3 > s2)
            c = 2;
        else if (c == 3 && s2 + s3 > s1)
            c = 2;
        else
            c = 4;
        return c;
    }

}
```

TriTypTest (Sep 17, 2017 5:32:16 PM)

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
JUnitTestExample	44.9 %	119	146	265

Example: Code Coverage in Eclipse (3)

Additional Test Cases

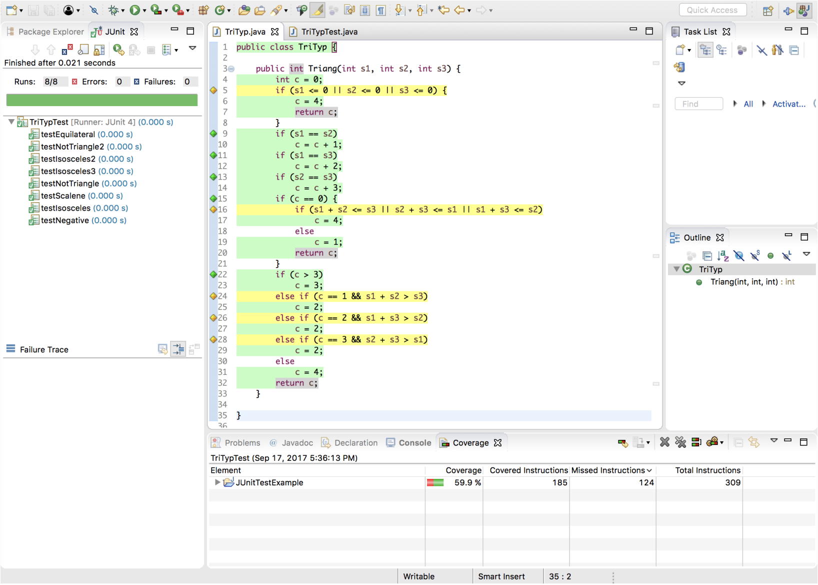
```java
@Test public void testNegative() {
    assertEquals((new TriTyp()).Triang(-1, -1, -1), 4);
}

@Test public void testIsosceles2() {
    assertEquals((new TriTyp()).Triang(2, 2, 3), 2);
}

@Test public void testIsosceles3() {
    assertEquals((new TriTyp()).Triang(2, 3, 2), );
}

@Test public void testNotTriangle2() {
    assertEquals((new TriTyp()).Triang(2, 1, 1), 4);
}
```

Example: Code Coverage in Eclipse (4)



Finished after 0.021 seconds

Runs: 8/8 ☒ Errors: 0 ☒ Failures: 0

- TriTypTest [Runner: JUnit 4] (0.000 s)
  - testEquilateral (0.000 s)
  - testNotTriangle2 (0.000 s)
  - testIsosceles2 (0.000 s)
  - testIsosceles3 (0.000 s)
  - testNotTriangle (0.000 s)
  - testScalene (0.000 s)
  - testIsosceles (0.000 s)
  - testNegative (0.000 s)

```java
public class TriTyp {

    public int Triang(int s1, int s2, int s3) {
        int c = 0;
        if (s1 <= 0 || s2 <= 0 || s3 <= 0) {
            c = 4;
            return c;
        }
        if (s1 == s2)
            c = c + 1;
        if (s1 == s3)
            c = c + 2;
        if (s2 == s3)
            c = c + 3;
        if (c == 0) {
            if (s1 + s2 <= s3 || s2 + s3 <= s1 || s1 + s3 <= s2)
                c = 4;
            else
                c = 1;
            return c;
        }
        if (c > 3)
            c = 3;
        else if (c == 1 && s1 + s2 > s3)
            c = 2;
        else if (c == 2 && s1 + s3 > s2)
            c = 2;
        else if (c == 3 && s2 + s3 > s1)
            c = 2;
        else
            c = 4;
        return c;
    }

}
```

TriTypTest (Sep 17, 2017 5:36:13 PM)

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
JUnitTestExample	59.9 %	185	124	309

Writable | Smart Insert | 35 : 2

Test Activities

- Test design
  - design test cases to satisfy coverage criteria
  - criteria-based (by tools) or human-based (by domain experts)
- Test automation
  - embed test values into executable scripts
  - JUnit (Java), NUnit (.NET), google test (C++), ...
- Test execution
  - run tests on the software and record the results
  - manual (e.g., usability testing) vs. automated
- Test evaluation
  - evaluate results of testing, report to developers
  - requires domain knowledge

When to Test

- During or after any development phases
  - requirement analysis, design, coding, . . .
- After any change made
  - (automated) regression testing
- Before coding: eXtreme Programming (XP)
  - base tests from user stories
  - more tests to improve coverage during development
- Continuous integration (CI)
  - automated unit tests and version control system
  - Jenkins, Buildbot, Travis CI, . . .

Summary

- Software testing
  - test cases, test suites, test oracles, . . .
  - JUnit: Test automation framework for Java
  - Test criteria: equivalence partitioning, structural code coverage
- Homework 2 (due 9/20)
  - maven, Java, and JUnit
- Reading
  - `http://www.exampler.com/testing-com/writings/catalog.pdf`
  - `http://www.exampler.com/testing-com/writings/short-catalog.pdf`

Questions?