LECTURE 18

FURTHER TESTING

Ian Stavness, CMPT 270, 2017-18 Term 1

Testing Review

- In almost all cases, it is impossible to test all possible inputs.
  - need to select good **test cases**
  - for each test case
    - need to find input data that satisfies the test case and the precondition
    - need to execute the program on the test data
    - need to verify that the output is correct, i.e., satisfies the postcondition
    - need to save the test case so that it can be easily used for regression testing

      e.g., JUnit testing

Testing Review

- Recall types of testing
  - Human testing: people reading the documents/code
  - Black-box testing: **test cases** based on the specification only, ignoring the actual code
    - Boundary-value testing
      - minimum, maximum, nominal values from a range; robust testing
      - special values
    - Each input and output case of the specification
      - include special cases
    - Equivalence-class testing
  - White-box testing: **test cases** based on the code used to implement the system
  - Object-oriented testing: **test cases** particularly suited to systems implemented by an object-oriented language

JUnit testing

- Until now, testing has been associated with a class
  e.g., include testing for a class in its main method
- JUnit, a tool to facilitate unit/module testing that is integrated into Eclipse
  - create a new package and test class
    - select project, right click New: JUnit Test Case
  - add junit-X.X.jar to the Java Build Path
    - if necessary, use project: Properties: Java Build Path: Libraries: Add Library:JUnit:JUnit4
    - If necessary, the jar file can be obtained from junit.org

JUnit test class

- must **import org.junit.*;**

- can have arbitrary many public void methods with @Test annotation (**after** the comment)
  - executed in some order (tests should be independent so ordering shouldn't matter)
  - any failures are recorded and reported at the end

JUnit test class

can have one public **static** void method with @BeforeClass annotation

- usually listed before the other methods
- it is executed **once** before all of the other methods
- used to initialize the system (but not instance fields)

can have one public **static** void method with @AfterClass annotation

- it is executed **once** after all of the other methods

JUnit test class

- can have several public void methods with @Before annotation
  - they are all executed in some order before **each** of the @Test methods
- can have several public void methods with @After annotation
  - they are all executed in some order listed **each** of the @Test methods
- Execute the test class by selecting the project and then Run … Run as … JUnit Test
- automatically **executes all annotated methods** of the class

Assert

- Special static methods from class Assert (org.junit) for use in a @Test method
  - static void assertTrue(boolean condition)
  - static void assertFalse(boolean condition)
  - static void assertEquals(java.lang.Object expected, java.lang.Object actual)
  - static void assertNotNull(java.lang.Object object)
  - static void assertNull(java.lang.Object object)
  - many others

Ex. Banking System

```
@BeforeClass
public static void setUp()
{
    // Initialize the containers of the application.
    InitializeSystem is = new InitializeSystem();
    String temp = "The initial state is \n" + is.toString() + "\n";
    Assert.assertTrue(temp != null);
}
```

Ex. Banking System

```
@Test
public void testBalanceCommand ()
{
    BalanceCommand balCommand =
                                    new BalanceCommand();
    balCommand.accessBalance(1);
    Assert.assertTrue(balCommand.wasSuccessful());
    Assert.assertTrue(balCommand.amount == 0);
    balCommand.accessBalance(5);
    Assert.assertTrue(! balCommand.wasSuccessful());
}
```

Ex. Banking System

- Can implicitly catch an exception that is thrown

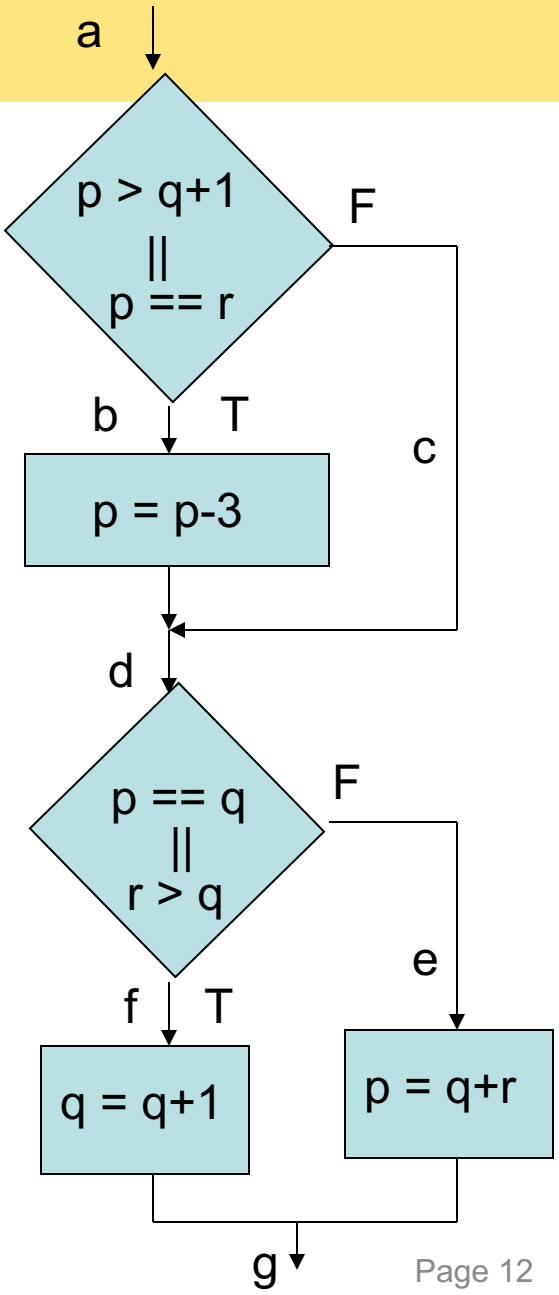
```
@Test(expected=IllegalStateException.class)

public void testLoginException ()
{
    LoginCheckCommand loginCommand =
                                    new LoginCheckCommand();
    loginCommand.login(1, "100");  //Invalid password
    Assert.assertTrue(! loginCommand.wasSuccessful());
    // Test the exception throw from getUser.
    Assert.assertTrue(loginCommand.getUser() == null);
}
```

White-box Testing

Testing based on looking at the code

1. Statement coverage
   Execute each statement at least once.
   e.g., paths: abdfg and abdeg
   data: p=5, q=2, r=4
   p=5, q=0, r=-1
2. Branch coverage
   Execute every branch at least once.
   e.g., paths: abdfg and acdeg
   data: p=5, q=2, r=4
   p=3, q=2, r=1
3. Condition coverage
   Try the true and false cases for each condition.
   For the pair of if conditions, try all combinations of true and false.
   TT, TF, FT, FF
   Need at least 4 paths.
   Try each value for each boolean expression (8 cases).
4. Path coverage
   Execute every path.
   Paths: abdfg, abdeg, acdfg, acdeg

Note: path coverage does not imply condition coverage, and condition coverage might not imply path coverage



White-box Testing (cont.)

- Sometimes not all possible paths or condition values are feasible,
  i.e., it is impossible to find data that causes the execution of some paths,
  or some combinations of condition values.
- Handling loops
  Usually impossible to test all paths: infeasible paths or too many paths.
  e.g., put a counted loop from 1 to n around the previous construct
  4^n paths, many of which will be infeasible for n non-trivial
  Boundary cases for testing a loop:
  number of times that the loop is executed: 0, 1, 2, many, max, max -1
- Handling recursive methods
  - Usually impossible to test all paths, probably infinite.
  - Boundary cases for testing recursion:
    number of recursive calls: 0, 1, 2, many
  - If more than one base case,
    do boundary testing leading to each base case (if feasible).

Grey-box Testing

- The tester has partial knowledge of the system being tested.

  Examples:

  - testing a sort algorithm,

    know the technique used

    bubble sort, selection sort, merge sort, quick sort
  - testing insertion into a list

    know the data representation used

    linked list, array
  - testing insertion into a dictionary

    know the data representation

    linked list, array, binary tree

    hash table with linear probing

    with double hashing

Object-oriented Testing

- Test all routines of a class

  Order for testing:

  - test routines that don't invoke other routines
  - test routines that only invoke routines within the class already tested (intra-class)
  - test routines that only invoke routines within other classes already tested (inter-class)
  - test recursive routines

  Sometimes 2 or more routines might need to be tested at the same time.

  - they work closely together
  - need a second one to test correctness of the first one

Object-oriented Testing

- Test inheritance
  - Test new methods
  - Test overridden methods
  - Test an inherited method that is not overridden

    if it invokes an overridden method

    or it accesses a field that is modified by a new or overridden method.

Object-oriented Testing

- State-based testing
  - An object has a state, as specified by its field values
  - Results of a method often depend upon the state when it was invoked
  - Organize the states into equivalence classes
  - Test state transitions (moving from one equivalence class to another equivalence class)