Supplemental Materials: Software Testing

CS2: Data Structures and Algorithms
Colorado State University

Chris Wilcox, Russ Wakefield, Wim Bohm, Dave Matthews

Topics

- Software Testing
- Test Driven Development
- Black Box Testing
- Unit Testing
- White Box Testing
- Coverage Testing
- Software Debugging

Defects and Reliability

- ***Software Defects:*** are inevitable in a complex software system.
  - In industry: 10-50 bugs per 1000 lines of code!
  - Defects can be obvious or remain hidden.
- ***Software Reliability***: What is the probability of failure of a software package over time:
  - Measurements: mean time between failures, crash statistics, uptime versus downtime.

Common faults in algorithms

- Incorrect logical conditions
- Calculation performed in wrong place
- Non-terminating loop or recursion
- Incorrect preconditions for an algorithm
- Not handling null conditions
- Off-by-one errors
- Operator precedence errors

Numerical faults in algorithms

- Not using enough bits or digits
- Not using enough places before or after the decimal point
- Assuming a floating point value will be exactly equal to some other value
- Ordering operations poorly so errors build up

Other faults in algorithms

- Poor minimal configuration performance
- Handling peak loads or missing resources
- HW and SW configuration incompatibility
- Crash recovery
- Deadlock, livelock, and critical races
- Inappropriate resource management

Definition

- ***Software Testing*** is a systematic attempt to reveal errors in software by running test programs or scripts (interactively or automated).
  - FAILING TEST: an error was demonstrated in the software under test.
  - PASSING TEST: no error was found, at least for this particular situation.
  - Theory of testing says you cannot prove absence of all defects in software.

Exhaustive Testing?

- We consider a program to be *correct* if it produces the expected output **for all inputs**.
- Domain of input values can be very large, e.g. 2^{32} values for an integer or float:

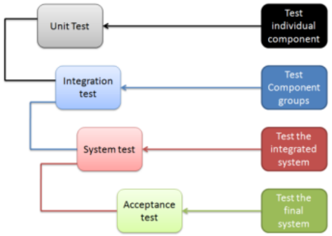
```
int divide (int operand1, int operand2);
```

- $2^{32} * 2^{32} = 2^{64}$, a large number, so we clearly cannot test exhaustively!
- And that is just for one method, in one class, in one package, and relatively simple.

Software Testing

- Types
  - **Functional**, Configuration, Usability, Reliability, Performance, Compatibility, Error, Localization, ...
- Processes
  - **Test-Driven Development**, **Coverage Testing**, Automated Testing, Regression Testing, …
- Methods
  - **Black-box, white-box**
- Levels
  - **Unit** (Method), Module (Class), Integration, System

Functional Testing



Two Kinds of Tests

- Tests that find defects after they occur
  – a waste of time
- Tests that prevent defects
  – the only kind to create

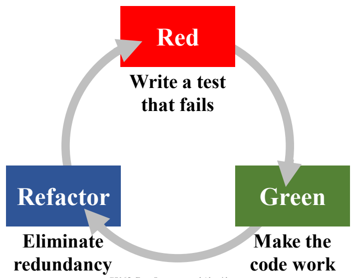
Citation: Study of the Toyota Production System, Shigeo Shingo

Test Driven Development

- Goal: Clean code that works!
- Drive development with automated tests
  – write new code only if tests fail
  – eliminate duplication
- Implies a different order of tasks
  – Red - write a little test that fails first
  – Green - make the test work quickly
  – Refactor - eliminate any resulting duplications

Citation: Test Driven Development, Kent Beck

Test Driven Development



Program testing methods

- Black-box testing
  - Specifications drive test inputs and expected outputs
  - Code, design or internal documents unavailable
- White-box testing
  - Code structure drives test inputs
  - Specifications used to derive expected outputs
  - Code, design, and internal documents available

Black-box Testing

- Specifications drive test inputs and expected outputs
- Code, design or internal documents unavailable

Equivalence classes

- Groups of inputs to be treated similarly
  - Numbers: <0, 0, >0
  - Numbers: <0, 0..1, >1
  - Months: [-∞..0], [1..12], [13..∞]
  - Months: “Jan”, ”Feb”, ”Mar”, ”Apr”, ”May”, ”Jun”, ”Jul”, ”Aug”, ”Sep”, ”Oct”, ”Nov”, ”Dec”, any other 3 character string.
  - Years: <0, [0..99], >99
  - Years: <0, [0..9999], >9999

Equivalence partition testing

- Test at least one value of every equivalence class for each individual input.
- Test all combinations where one input is likely to affect the interpretation of another input.
- Test random combinations of equivalence classes.

Boundary value testing

- Expand equivalence classes to test values at extremes of each equivalence class.
- Number ranges:
  - minimum, slightly above minimum, nominal or median value, slightly below maximum, and maximum values
  - values slightly and significantly outside the range

Boundary Value Testing Example

Test boundaries of the parameter value domain:

```
// Boundary testing of Math.floor
System.out.println(Math.floor(Double.MIN_VALUE));
System.out.println(Math.floor(Double.MAX_VALUE));
System.out.println(Math.floor(-987654321.123456789));
System.out.println(Math.floor(-1.999999));
System.out.println(Math.floor(-1.000001));
System.out.println(Math.floor(-1.0));
System.out.println(Math.floor(-0.0));
System.out.println(Math.floor(+0.0));
System.out.println(Math.floor(+1.0));
System.out.println(Math.floor(1.000001));
System.out.println(Math.floor(1.999999));
System.out.println(Math.floor(987654321.123456789));
```


Unit Testing

- JUnit is a simple, open source framework to write and run repeatable tests. JUnit is commonly used in industry for unit testing. Features include:
  - Assertions for testing expected results
  - Test fixtures for sharing common test data
  - Test runners for running tests

Citation: JUnit testing framework (http://www.junit.org/)

JUnit value assertions

```
assertTrue( 'a' < 'b' , "message");
assertFalse( 'b' < 'a' );

assertEquals( 1+1, 2 );

assertEquals( 22.0d/ 7.0d, 3.14159, 0.001 );

assertEquals( "cs165" , "cs165" );
```

Citation: JUnit testing framework (http://www.junit.org/)

JUnit array assertions

```
int[] array1 = { 1, 2, 3 };
int[] array2 = { 1, 2, 3 };

assertNull( null );
assertNotNull( array1 );

assertNotSame( array1, array2 );

assertArrayEquals( array1, array2 );
```

Citation: JUnit testing framework (http://www.junit.org/)

JUnit other assertions

```
Stack<Object> stack = new Stack<>();

assertThrows(EmptyStackException.class,
        () -> stack.pop());

assertThrows(EmptyStackException.class,
        () -> stack.peek());
```

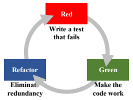
Citation: JUnit testing framework (http://www.junit.org/)

JUnit test class

```
import static
     org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;

class FirstJUnit5Tests {

  @Test
  void myFirstTest() {
    assertEquals(2, 1 + 1);
  }
}
```

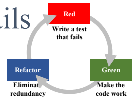
Citation: JUnit testing framework (http://www.junit.org/)

Test Driven Development Example



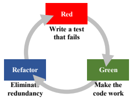

```
class Example {

 /* Performs requested operations on a value
  * @param x is an integer input value
  * @param c1 is a boolean that increments the input value if true
  * @param c2 is a boolean that squares the input value if true
  * @param c3 is a boolean that negates the input value if true
  * @return the modified input value
  */
 int xmpl( int x, boolean c1, boolean c2, boolean c3) {
  return 0;
 }

}
```


RED
Write a test that fails



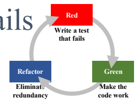

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

/* Test example function*/
class TestExample{

 @Test
 public void testDoNothing() {
  assertEquals(2, Example.xmpl( 2, false, false, false ));
 }
}
```


Green
Make it work



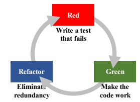

```
class Example {

 /* Performs requested operations on a value
  * @param x is an integer input value
  * @param c1 is a boolean that increments the input value if true
  * @param c2 is a boolean that squares the input value if true
  * @param c3 is a boolean that negates the input value if true
  * @return the modified input value
  */
 int xmpl( int x, boolean c1, boolean c2, boolean c3) {
  return x;
 }

}
```

RED
Write a test that fails



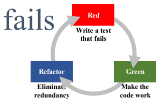

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

/* Test example function*/
class TestExample{

 @Test
 public void testIncrement() {
   assertEquals(1, Example.xmpl( 0, true, false, false ));
   assertEquals(3, Example.xmpl( 2, true, false, false ));
 }
}
```


Green
Make it work



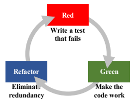

```
class Example {

 /* Performs the requested operations on a value
  * @param x is an integer input value
  * @param c1 is a boolean that increments the input value if true
  * @param c2 is a boolean that squares the input value if true
  * @param c3 is a boolean that negates the input value if true
  * @return the modified integer input value
  */
 int xmpl( int x, boolean c1, boolean c2, boolean c3) {
   if ( c1 ) x++;
   return x;
 }
}
```


RED
Write a test that fails




```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

/* Test example function*/
class TestExample{

 @Test
 public void testSquare() {
   assertEquals(0, Example.xmpl( 0, false , true, false ));
   assertEquals(4, Example.xmpl( 2, false , true, false ));
 }
}
```

Green
Make it work

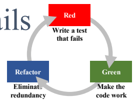



```
class Example {

 /* Performs requested operations on a value
  * @param x is an integer input value
  * @param c1 is a boolean that increments the input value if true
  * @param c2 is a boolean that squares the input value if true
  * @param c3 is a boolean that negates the input value if true
  * @return the modified input value
  */
 int xmpl( int x, boolean c1, boolean c2, boolean c3) {
   if ( c1 ) x++;
   if ( c2 ) x *= x;
   return x;
 }
}
```

CS165: Data Structures and Algorithms – Spring Semester 2018

31

RED
Write a test that fails

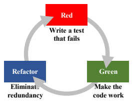



```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

/* Test example function*/
class TestExample{

 @Test
 public void testNegate() {
   assertEquals( 0, Example.xmpl( 0, false, false, true));
   assertEquals(-2, Example.xmpl( 2, false, false, true));
 }
}
```

CS165: Data Structures and Algorithms – Spring Semester 2018

32

Green
Make it work

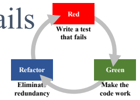



```
class Example {

 /* Performs requested operations on a value
  * @param x is an integer input value
  * @param c1 is a boolean that increments the input value if true
  * @param c2 is a boolean that squares the input value if true
  * @param c3 is a boolean that negates the input value if true
  * @return the modified input value
  */
 int xmpl( int x, boolean c1, boolean c2, boolean c3) {
   if ( c1 ) x++;
   if ( c2 ) x *= x;
   if ( c3 ) x = -x;
   return x;
 }
}
```

CS165: Data Structures and Algorithms – Spring Semester 2018

33

RED
Write a test that fails




```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

/* Test example function*/
class TestExample{

 @Test
 public void testCombination() {
  assertEquals(-9, Example.xmpl( 2, true, true, true));
 }
}
```


White-Box Testing

- White-box testing
  - Code structure drives test inputs
  - Specification used to derive tests outputs
  - Code, design, and internal documentation are available
  - Goal is to ”cover” the code to gain confidence and detect defects.

White Box Testing

- Statement Coverage (most common)
  - Requires all statements to be executed
- Branch Coverage
  - Require decisions evaluate to true and false at least once
  - Implies statement coverage
- Path Coverage (least common)
  - Require all possible paths to be executed
  - Implies branch coverage

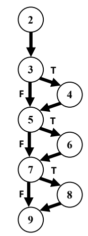
Coverage Problems

- Statement
  - May not exercise all the conditions in condition predicates
- Branch
  - May not exercise all combinations of branches
- Path
  - combinatorial explosion
  - infinite paths for loops
  - not all paths are feasible

Statement Coverage

All statements must be executed.

```
public class Example {
  public int xmpl( int x,
     boolean c1, boolean c2, boolean c3) {
    if (c1)
      x++;
    if (c2)
      x *= x;
    if (c3)
      x = -x;
    return x;
  }
}
```



What tests are required?

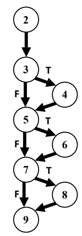
Statement Coverage

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

/* Test example function*/
class TestExample{

  @Test
  public void statementCoverage() {
    assertEquals(-9, Example.xmpl( 2, true, true, true));
  }
}
```

Branch Coverage

All decisions must evaluate to both true and false.

```
public class Example {
  public int xmpl( int x,
      boolean c1, boolean c2, boolean c3) {
    if (c1)
      x++;
    if (c2)
      x *= x;
    if (c3)
      x = -x;
    return x;
  }
}
```



What tests are required?

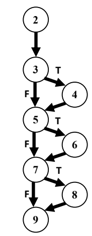
Branch Coverage

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

/* Test example function*/
class TestExample{

  @Test
  public void branchCoverage() {
    assertEquals( 2, Example.xmpl( 2, false, false, false ));
    assertEquals(-9, Example.xmpl( 2, true, true, true));
  }
}
```


Path Coverage

All paths must be executed.

```
public class Example {
  public int xmpl( int x,
      boolean c1, boolean c2, boolean c3) {
    if (c1)
      x++;
    if (c2)
      x *= x;
    if (c3)
      x = -x;
    return x;
  }
}
```



What tests are required?

Path Coverage

```
class TestExample{
 @Test
 public void pathCoverage() {
  assertEquals( 2, Example.xmpl( 2, false, false, false ));
  assertEquals( 3, Example.xmpl( 2, true, false, false ));
  assertEquals( 4, Example.xmpl( 2, false, true, false ));
  assertEquals(-2, Example.xmpl( 2, false, false, true));
  assertEquals( 9, Example.xmpl( 2, true, true, false ));
  assertEquals(-3, Example.xmpl( 2, true, false, true));
  assertEquals(-4, Example.xmpl( 2, false, true, true));
  assertEquals(-9, Example.xmpl( 2, true, true, true));
 }
}
```

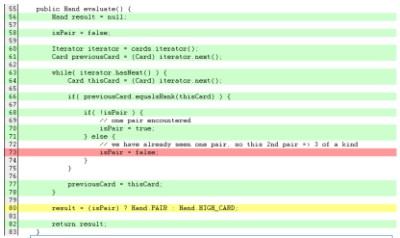

Code Coverage

- Code coverage improves software quality by illumination: it shines light on executed code, and reveals dark corners that are untested or never used. This software metric can enhance many projects, from standard business apps to those with ultra-low tolerance for error, for example medical devices.

Citation: Emma code coverage tool (http://emma.sourceforge.net/)

Code Coverage

Green = executed, Yellow = partial, Red = not executed



Citation: Emma code coverage tool (http://emma.sourceforge.net/)

Software Debugging

- Possible methods for debugging:
  - Using debugger
  - Print debugging
  - Examining code
- IDEs have built-in debuggers
- Computer Science department has print debugging package – Debug.java
- Code inspections done by teams in industry

Print Debugging

```
public static void readFile (String filename) {

    try {
        Scanner reader = new Scanner(new File(filename));
        while (reader.hasNextLine()) {
            String line = reader.nextLine();
            System.out.println(line); // debug print
            contents.add(line);
            reader.close(); // code defect
        }
    } catch (IOException e) {
        System.out.println(e.getMessage());
    }
}
```


Debugging Tools