Software Testing

CS2: Data Structures and Algorithms
Colorado State University

Chris Wilcox, Russ Wakefield, Wim Bohm, Dave Matthews, Sudipto Ghosh
Topics

集成电路

计算机体系结构

Software Testing

Black Box Testing

Unit Testing with JUnit

Test Driven Development

White Box Testing

Software Debugging
Faults and reliability

- **Software Faults (aka bugs and defects):** inevitable in a complex software system.
  - 10-50 faults per 1000 lines of code in industry!
  - Faults can be known or remain hidden.
  - Either way, they can cause software to fail.

- **Software Reliability:** probability of failure of a software system over time. Measured using
  - mean time between failures, crash statistics, uptime versus downtime.
Common faults in programs

- 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
Faults in numerical programs

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

- **Software Testing** is a systematic attempt to reveal faults in software by running test programs or scripts (interactively or automated).

- Test case is a test input along with its expected output
  - FAILING TEST: a fault was demonstrated in the software under test.
  - PASSING TEST: no fault was found (even if it existed).

- Dijkstra said: “Program testing can be used to show the presence of bugs, but never to show their absence!”
Software Testing

- **Types**
  - Functional, Usability, Performance, ...

- **Levels**
  - Unit (Method/Class), Integration, System, Acceptance

- **Test case creation methods**
  - Black-box, white-box

- **Processes**
  - Test-Driven Development, Coverage Testing, Regression Testing, …
Functional Testing

- Unit Test
  - Test individual component
- Integration test
  - Test Component groups
- System test
  - Test the integrated system
- Acceptance test
  - Test the final system
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:
  
  ```java
  int divide (int operand1, int operand2);
  ```
  
  $2^{32} \times 2^{32} = 2^{64}$, 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.
- Thus, exhaustive testing isn’t feasible. Need smart ways to select test inputs!
Test case creation methods

- **Black-box testing**
  - Code, design or internal documents unavailable
  - Test inputs obtained from specifications
  - Expected outputs also obtained from specifications

- **White-box testing**
  - Code, design, and internal documents available
  - Test inputs obtained from code structure
  - Expected outputs obtained from specifications
Black-box Testing

- Divide large input domain into a small number of equivalence classes
- Also consider boundaries of equivalence classes
Equivalence classes

- Groups or partitions of inputs to be treated similarly
- Must be complete and disjoint
- Strategy selected is based on the problem to be solved
- Partitioning integers based on
  - Sign:
    - Classes are \{positive ints\}, \{negative int\}, \{0\}
    - Choose 4, -6, and 0 as inputs
  - Even or odd
    - Classes are \{even ints\}, \{odd ints\}
    - Choose 6 and 3 as inputs
Examples of equivalence classes

- Months represented as ints: (Red is invalid input)
  - Partitions: $[-\infty..0]$, $[1..12]$, $[13..\infty]$  
  - Representative values: -4, 5, 15

- Months represented as strings:

- Month numbers grouped by number of days:
  - Partitions $\{1,3,5,7,8,10,12\}$, $\{4,6,9,11\}$, $\{2\}$
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
- Testing array of length 10:
  - Using partitions {0}, {positive}, select indices 0, 4
  - Using boundary values, select indices -1, 9, 10
Boundary value testing example

Test boundaries of the parameter value domain:

```java
// 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));
```
How to specify expected outputs?

- Find the exact expected answer by using the specification (e.g., $\text{gcd}(4,6) = 2$)
  - $\text{gcd}(p,0)$ ($p \neq 0$) = Math.abs($p$)
  - $\text{gcd}(0,q)$ ($q \neq 0$) = Math.abs($q$)
  - $\text{gcd}(0,0) = 0$
  - $\text{gcd}(p,q)$ ($p \neq 0$ and $q \neq 0$) = $d$ ($d > 0$ and $d$ largest int such that $d$ divides $p$ and $d$ divides $q$)

- Find a suitable condition involving the variables (e.g., $\text{gcd}(p, q) \geq 0$)

- Use stronger checks as much as possible to write more powerful test cases
JUnit

- Simple, open source framework to write and run repeatable tests.
- Commonly used in industry for unit testing.
- Typical workflow inside a test case (or test method):
  - Set up the objects involved in the test with appropriate values
  - Call the method under test with appropriate parameters
  - Capture the method return value and/or state information on the object of interest
  - Write assertions about the return value and/or the state information

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

✧ Eclipse project contains a file called GCD.java in package junitintro
✧ Click on File ➔ New ➔ JUnit Test Case to create a file called GCDTest that tests GCD
✧ Remember to include the JUnit 5 library
✧ A JUnit test class is created with the following declarations:

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

class GCDTest {
    @Test
    void test() {
        fail("Not yet implemented");
    }
}
```

This test will fail because nothing is implemented yet
Selecting inputs for greatest common divisor (gcd)

- gcd takes two ints
- What is a good partitioning strategy?
  - positive/negative useful
  - even/odd NOT useful
- Use domain knowledge: presence or absence of common factors in the numerator/denominator
  - No common factor: 11, 13. Expected result 1
  - Some common factor: 16, 20. Expected result 4
public class GCD {
    public int gcd(int p, int q) {
        int a = Math.abs(p), b = Math.abs(q);
        if (b==0) return a;
        else if (a==0) return b;

        int rem=1, result=1;
        while(rem!=0) {
            rem = a % b;
            if(rem==0) result=b;
            a = b;  b = rem;
        }
        return result;
    }
}

@Test
void testNoCommonFactors() {
    GCD g = new GCD();
    int result = g.gcd(11, 13);
    assertEquals(result, 1);
}

@Test
void testSomeCommonFactors() {
    GCD g = new GCD();
    int result = g.gcd(16, 20);
    assertTrue(result==4);
}

@Test
void testNegativeNegativeNoCommonFactor() {
    GCD g = new GCD();
    int result = g.gcd(-13, -20);
    assertEquals(result,1,
                "Expected 1");
}
More JUnit value assertions

```java
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

```java
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/)
Two Kinds of Tests

✧ Tests that find defects after they occur
  – Often written by other developers/testers
  – Or as an afterthought

✧ Tests that prevent defects
  – Help you think about coding specific types of cases/conditions while you are coding
  – Often used in modern software development
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**
  1. Write a test that fails first
  2. Make the test work in the code

*Citation: Test Driven Development, Kent Beck*
Using TDD:
Creating a simple constructor

```java
public class Rational {
    private int numerator, denominator;
}
```

- Develop the constructor and `toString` code
  - Let the constructor handle integers of the form \( \frac{p}{q} \) where \( p \) and \( q \) are positive and have no common factors
  - `toString` returns a string in the form of \( \frac{p}{q} \)
First step: Simple constructor

```java
public class RationalTest {
    @Test
    void testNoCommonFactor() {
        Rational r = new Rational(3, 5);
        String result = r.toString();
        assertEquals(result, "3/5");
    }
}

public class Rational {
    private int numerator, denominator;

    public Rational(int n, int d) {
        numerator = n;
        denominator = d;
    }

    public String toString() {
        return new String(numerator + 
                          "/" + denominator);
    }
}
```
Using TDD: Handle zero denominator

- Let the constructor also handle integers of the form p/q where p>0 and q==0
- This needs to throw an exception because the number is not valid
- Since such a number can’t be created, toString doesn’t need to handle this case
Second step: Handle zero denominator

```java
public class Rational {
    private int numerator, denominator;
    public Rational(int n, int d) {
        if (d==0)
            throw new ArithmeticException();
        numerator = n;
        denominator = d;
    }
    public String toString() {
        return new String(numerator + "\n/" + denominator);
    }
}
```

```java
public class RationalTest {

    @Test
    void testNoCommonFactor() {
        Rational r = new Rational(3, 5);
        String result = r.toString();
        assertEquals(result, "3/5");
    }

    @Test
    void testZeroDenominator() {
        try {
            Rational r = new Rational(3, 0);
            fail("Did not throw an arithmetic exception");
        } catch (ArithmeticException e) {
        }
    }
}
```
Using TDD: Handle special cases

- Let the constructor handle integers of the form \( p/q \) where \( p \) and \( q \) are any integers but have no common factors
  - If numerator is 0, then the denominator is stored as 1
  - The sign is stored in the numerator.
  - The denominator is always positive.
- `toString` doesn’t need to handle this case any differently because the constructor takes care of the representation
Third step: Handle special cases

```java
public class Rational {
    private int numerator, denominator;

    public Rational(int n, int d) {
        if (d==0) throw new ArithmeticException();
        if (n==0) {
            numerator = 0;
            denominator = 1;
        } else {
            denominator = Math.abs(d);
            numerator = (d > 0)? n: -n;
        }
    }

    public String toString() {
        return new String(numerator + "/" + denominator);
    }
}

@Test
void testPositiveNegative() {
    Rational r = new Rational(3, -5);
    String result = r.toString();
    assertEquals(result, "-3/5");
}

@Test
void testNegativePositive() {
    Rational r = new Rational(-3, 5);
    String result = r.toString();
    assertEquals(result, "-3/5");
}

@Test
void testNegativeNegative() {
    Rational r = new Rational(-3, -5);
    String result = r.toString();
    assertEquals(result, "3/5");
}

@Test
void testZeroNumerator() {
    Rational r = new Rational(0, -5);
    String result = r.toString();
    assertEquals(result, "0/1");
}
```
Using TDD: Handle common factors

- Let the constructor handle integers of the form $p/q$ where $p$ and $q$ are positive but have common factors
- We need to normalize (i.e., reduce $p$ and $q$ to the lowest common denominator)
- `toString` doesn't need to handle this case any differently because the constructor takes care of the reduction
Fourth step: Handle common factors

```java
public class Rational {
    private int numerator, denominator;

    public Rational(int n, int d) {
        if (d==0) throw new ArithmeticException();
        if (n==0) {
            numerator = 0;  denominator = 1;
        } else {
            denominator = Math.abs(d);
            numerator = (d > 0)? n: -n;
            reduce();
        }
    }

    private void reduce () {
        int common = gcd(numerator, denominator);
        numerator = numerator / common;
        denominator = denominator / common;
    }

    // code for toString not shown...
}
```

```java
@Test
void testCommonFactorPositivePositive() {
    Rational r = new Rational(16, 20);
    String result = r.toString();
    assertEquals(result, "4/5");
}

@Test
void testCommonFactorPositiveNegative() {
    Rational r = new Rational(16, -20);
    String result = r.toString();
    assertEquals(result, "-4/5");
}

@Test
void testCommonFactorNegativePositive() {
    Rational r = new Rational(-16, 20);
    String result = r.toString();
    assertEquals(result, "-4/5");
}

@Test
void testCommonFactorNegativeNegative() {
    Rational r = new Rational(-16, -20);
    String result = r.toString();
    assertEquals(result, "4/5");
}
```
Using TDD:
String representation for special cases

- Modify toString to print special cases
  - When the numerator is 0, print 0
  - When the denominator is 1 in the reduced form, just print the numerator.
public class RationalTest {

    // include all the previous tests
    // May need to adapt prior tests
    // that has zero numerator

    @Test
    void testNumeratorZero() {
        Rational r = new Rational(0, 20);
        String result = r.toString();
        assertEquals(result, "0");
    }

    @Test
    void testDenominatorOne() {
        Rational r = new Rational(-16, 1);
        String result = r.toString();
        assertEquals(result, "-16");
    }
}

public class Rational {
    private int numerator, denominator;

    // include other methods

    public String toString() {
        if (numerator==0 || denominator==1)
            return new Integer(numerator).toString();
        else
            return new String(numerator + "/" + denominator);
    }
}
Using TDD: Ability to check equality of numbers

- Add an equals method
- Needed if you further implement add, subtract, multiple, and divide operations and must check their results
- Since the constructor takes care of normalizing, we can just compare the numerators and denominators.

Several test cases:
- Two numbers with the same numerator and denominator
- Two numbers with different numerator and denominators
- With and w/o $\text{gcd} > 1$
Sixth step:
Adding the equals method

```java
public class Rational {
    private int numerator, denominator;

    // include other methods

    public boolean equals (Object other) {
        if(other instanceof Rational) {
            return (numerator == ((Rational)other).getNumerator() &&
                    denominator == ((Rational)other).getDenominator());
        } else {
            return false;
        }
    }

    @Test void testTwoEqualRationalNumbers() {
        Rational r1 = new Rational (16, 20);
        Rational r2 = new Rational (20, 25);
        assertEquals(r1, r2);
    }

    @Test void testTwoEqualRationalNumbersDifferentSigns() {
        Rational r1 = new Rational (-16, 20);
        Rational r2 = new Rational (20, -25);
        assertEquals(r1, r2);
    }

    @Test void testTwoIdenticalRationalNumbers() {
        Rational r1 = new Rational (16, 20);
        Rational r2 = new Rational (16, 20);
        assertEquals(r1, r2);
    }

    @Test void testTwoUnequalRationalNumbers() {
        Rational r1 = new Rational (16, 20);
        Rational r2 = new Rational (6, 10);
        assertEquals(r1, r2);
    }
}
```
White Box Testing

- Goal is to “cover” the code to gain confidence and detect defects.
  - 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
Doing white box testing on gcd

- Often parts of the implementation are not executed by the test cases you have written using blackbox strategies
- Run Eclipse coverage tool (EclEmma) using the same JUnit test cases as before
- What is not covered? Suggest test inputs to cover those statements and branches
Code Coverage

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

```java
public class GCD {
    public int gcd (int p, int q) {
        int a = Math.abs(p);
        int b = Math.abs(q);
        if(b==0)
            return a;
        else if (a==0)
            return b;
        int rem=1;
        int result=1;
        while(rem!=0) {
            rem = a % b;
            if(rem==0) result=b;
            a = b;
            b = rem;
        }
        return result;
    }
}
```
Software Debugging

Possible methods for debugging:
- Examine code by hand
- Look at stack trace if program crashed with an exception to find out where the last method call happened.
- Use *Print* statements to show intermediate values
- Use built-in debugger in eclipse
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 a faulty program

- Use the Data.java file in the debugging package.
- The bubblesort method in the Data.java file has a fault but the programmer doesn’t know that.
- Some tests pass but others fail.
- Let’s debug the failing tests.
- Set a debug configuration in eclipse.
- Put a breakpoint at the bubblesort declaration.
Debugging in Eclipse

```
package debugging;

/* Buggy implementation of bubble sort */
public class Data {
    public void bubbleSort (int [] array) {
        for (int i = 0 ; i < array.length-1; position>=0; position++) {
            for (int i = 0 ; i < array.length-1; i++) {
                if (array[i]< array[i+1]) {
                    // faulty condition: should be > instead of <
                    swap(array, i, i+1);
                }
            }
        }
    }

    private void swap(int[] array, int i, int j) {
        int tmp = array[i];
        array[i] = array[j];
        array[j] = tmp;
    }
}
```

State of variables

Breakpoint

Line getting executed

objc[4909]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_151/jdk/Contents/Home/bin/java (Jan 21, 2020, 8:15:17 PM)