Supplemental Materials: Software Testing

CS2: Data Structures and Algorithms
Colorado State University

Chris Wilcox, Russ Wakefield, Wim Bohm

Topics

- Software Testing
- Functional Testing
- Regression Testing
- Code Coverage
- Preconditions & Postconditions
- Java Assertions
- Loop Invariants

Software Testing

- ***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.

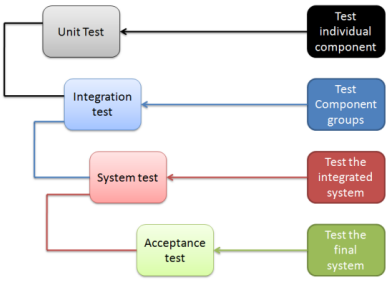
Definition

- ***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.

Software Testing

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

Functional Testing

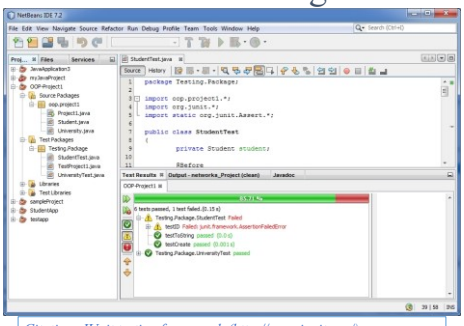


Unit Testing

- 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 Testing



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

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.

Boundary Testing (1)

- Test with the boundaries of the domain of parameter values:

```
// 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));
```


Regression Testing

- Scenario: company with engineers continually writing or modifying code .
  - Question: How to find new defects introduced while enhancing code or fixing old defects?
  - Answer: Regression test the code with a build and verification cycle on a regular basis, using a growing test suite, with known correct outputs.
  - Assumes test automation otherwise not practical.
- Code won't remain stable without a consistent and thorough effort to maintain quality.

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/)

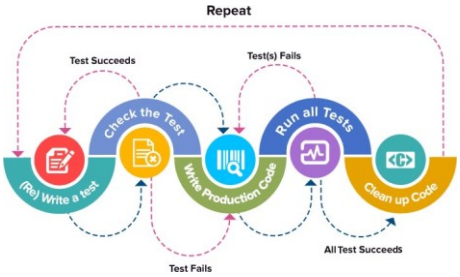
Code Coverage

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

```
55    public Hand evaluate() {
56        Hand result = null;
57
59        isPair = false;
59
60        Iterator iterator = cards.iterator();
61        Card previousCard = (Card) iterator.next();
62
63        while( iterator.hasNext() ) {
64            Card thisCard = (Card) iterator.next();
65
66            if( previousCard.equalsRank(thisCard) ) {
67
68                if( !isPair ) {
69                    // one pair encountered
70                    isPair = true;
71                } else {
72                    // we have already seen one pair, so this 2nd pair => 3 of a kind
73                    isPair = false;
74                }
75            }
76
77            previousCard = thisCard;
78        }
79
80        result = (isPair) ? Hand.PAIR : Hand.HIGH_CARD;
81
82        return result;
83    }
```

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

Test Driven Development



Pre and Post Conditions

- ***Pre-Condition***: a statement of what must be true about the input or variable values at the beginning of the program segment:
  - e.g. Math.sqrt requires the input parameter to be >= 0
- ***Post-Condition***: a statement of what must be true about the output or variable values at the end of the program segment:
  - e.g. Math.sqrt return value squared ~= input parameter
- Java Assertions: can be used to check pre and post conditions.

What is an assertion?

- An *assertion* is a statement in Java that enables you to test your assumptions about your program.
- Each assertion contains a boolean expression that you believe will be true when the assertion executes.
- Confirms your assumptions about the behavior of your program, increasing your confidence that the program is error free.

Citation: assertion slides from Dr. Glenn Blank at Lehigh University

Why use assertions?

- Programming by contract
  - Defines responsibilities of producer and user
- Pre-conditions
  - Assert precondition as requirement for method input
- Post-conditions
  - Assert post-condition as requirement for method output

Citation: assertion slides from Dr. Glenn Blank at Lehigh University

Simple Assertion Form

The assertion statement has two forms:

- The first form is:

```
assert Expression;
```

- Where *Expression* is a boolean expression
- When the system runs the assertion, it evaluates *Expression* and if it is false throws an AssertionError with no details.

Citation: assertion slides from Dr. Glenn Blank at Lehigh University

Complex Assertion Form

- The second form is:

```
assert Expr1 : Expr2 ;
```

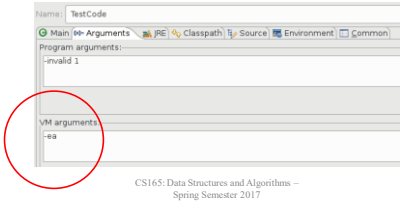
- where:
  - *Expr1* is a boolean expression;
  - *Expr2* is an expression (e.g. a String)

    if(!expr1) throw exception(expr2);

Citation: assertion slides from Dr. Glenn Blank at Lehigh University

Enabling Assertions

- For the virtual machine to enable assertions, use `-ea, -enableassertions`
- In Eclipse, this resides in the VM arguments in the Run Configuration



Performance Problems

- Assertions may slow down execution - why?
- So, assertions may be enabled and disabled
- Assertions, by default, disabled at run-time
- In this case, the assertion is empty statement
- Don't use for user errors - why not?

Citation: assertion slides from Dr. Glenn Blank at Lehigh University

Pre-Condition Example

```
public ArrayList<String> readFile(String filename) {

    ArrayList<String> contents = new ArrayList<>();
    try {

        File file = new File(filename);
        assert (file.exists()) : "readFile: file does not exist: " + filename;
        assert (file.canRead()) : "readFile: file is not readable: " + filename;
        assert (file.length() > 0) : "readFile: file is empty: " + filename;

        Scanner reader = new Scanner(file);
        // Code to read file
        reader.close();
    } catch (IOException e) {
        System.out.println(e.getMessage());
    }
    return contents;
}
```


Post-Condition Example

```
public void sortNames(ArrayList<String> names) {

    Collections.sort(names);

    // Check sorting
    for (int i = 1; i < names.size(); i++) {
        assert (names.get(i).compareTo(names.get(i-1)) > 0) : "names sorted incorrectly!";
    }
}
```


Loop invariants

- We use **predicates** (logical expressions) in assertions to reason about our programs.
- A **loop invariant** is a predicate
  - that is true directly before the loop executes
  - that is true before and after the loop body executes
  - and therefore true directly after the loop has executed

i.e., it is kept invariant by the loop.

Loop invariants cont'

- Combined with the loop condition, the loop invariant allows us to reason about the behavior of the loop:

```
<loop invariant>
while(test){
    <test AND loop invariant>
     S;
    <loop invariant>
}
< not test AND loop invariant>
```

What does it mean...

```
<loop invariant>
while(test){
   <test AND loop
   invariant>
      S;
   <loop
   invariant>
}
< not test AND
 loop invariant>
```

If we can prove that

. the loop invariant holds before the loop

and that

. the loop body keeps the loop invariant true

i.e. <test AND loop invariant> S; <loop invariant>

then we can infer that

. not test AND loop invariant holds after the loop terminates

Example summing

```
int total (int[] elements){
  // pre: elements.length > 0
  int sum = 0,i = 0, n = elements.length;
  // invariant?
   while (i < n){
     // i<n and invariant?
      sum  += elements [i];
      i++;
     // invariant?
  }
  // i==n  (previous example)  AND invariant
  // → sum == sum of int[] elements
 return sum;
}
```

Example summing

```
int total (int[] elements){
  pre n>0
  int sum = 0,i = 0, n = elements.length;
  // sum has sum of elements from 0 to i-1     the empty set  AND i<=n
   while (i < n){
     // sum == sum of elements 0..i-1  AND i<n
      sum  += elements [i];
      i++;
     // sum == sum of elements 0..i-1  AND i<= n
  }
  // !(i<n) AND i<=n implies  i==n AND
  // sum has sum elements 0..i-1  →  sum == sum of elements 0..n-1
  //                                 → sum == sum of int[] elements
 return sum;
}
```

Example: Egyptian multiplication

	A	B	
	19	5	
/2	9	10	*2
/2	4	20	*2
/2	2	40	*2
/2	1	80	*2

throw away all rows with even A:

	A	B
	19	5
	9	10
	1	80
add B's		95

--> the product !!

Egyptian multiplication

	A	B		
	19	5		19*5 = 95
/2	9	10	*2	9*10 = 90
/2	4	20	*2	4*20 = 80
/2	2	40	*2	2*40 = 80
/2	1	80	*2	1*80 = 80

throw away all rows with even A:

	A	B
	19	5
	9	10
	1	80
add B's		95

--> the product !!

Can we show it works? Loop invariants!!

```
// pre:   left >=0 AND right >=0
int a=left, b=right, p=0;
// p+(a*b) == left * right        loop invariant
while (a!=0){
  // a!=0 and p+a*b == left*right   loop condition and loop invariant
  if (odd(a))  p+=b;
  a/=2;
  b*=2;
  // p+(a*b) == left*right
}
// a==0 and p+a*b == left*right  →  p == left*right
```

Try it on 7 * 8

left	right	a	b	p
7	8	7	8	0
		3	16	+=b: 8
		1	32	+=b: 24
		0	64	+=b: 56

Try it on 8*7

left	right	a	b	p
8	7	8	7	0
		4	14	0
		2	28	0
		1	56	0
		0	118	+=b: 56

Relation to int representation 19*5

```
   00101
   10011
  ------
     101   5
    1010  10
   00000
  000000
 1010000  80
 -------
 1011111  95 = 64 + 31
```

Summary: Loop Invariant Reasoning

```
//loop invariant true before loop
while (b){
  // b  AND loop invariant
     S;
  // loop invariant
}
// not b  AND loop invariant
```

not b helps you make a stronger observation than loop invariant alone.

Software Debugging

- Related topic - you cannot test code until you get it working.
- Possible methods for debugging:
  - Examining code (weak)
  - Print debugging (better)
  - Using debugger (strong)
- Computer Science department has print debugging package – Debug.java

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