
Testing

January 19, 2005

1

Testing

- Overview
 - Testing -- definitions
 - JUnit
 - The FileReader example
 - The 8 rules of testing

- See Martin Fowler, *Refactoring*

January 19, 2005

2

Testing Objectives

- Testing is a process of executing a program with the intent of finding an error.
- A good test is one that has a high probability of finding an as yet undiscovered error.
- A successful test is one that uncovers an as yet undiscovered error.

The objective is to design tests that systematically uncover different classes of errors and do so with a minimum amount of time and effort.

Secondary benefits include

- Demonstrate that software functions appear to be working according to specification
- That performance requirements appear to have been met.
- Data collected during testing provides a good indication of software reliability and some indication of software quality.

Testing cannot show the absence of defects, it can only show that software defects are present.

January 19, 2005

3

Test Case Design

- Designing good test cases is often as difficult and work intensive as coding the original program
...this is why most of us don't do it, but we should
- Black box testing -- testing that code conforms to a design
- White box testing -- testing that code conforms to its specification

January 19, 2005

4

White box testing

- Testing control structures of a procedural design.
- Can derive test cases to ensure:
 1. all independent paths are exercised at least once.
 2. all logical decisions are exercised for both true and false paths.
 3. all loops are executed at their boundaries and within operational bounds.
 4. all internal data structures are exercised to ensure validity

January 19, 2005

5

White box testing

-
- Why do white box testing when black box testing is used to test conformance to requirements?
 - Logic errors and incorrect assumptions most likely to be made when coding for "special cases". Need to ensure these execution paths are tested.
 - May find assumptions about execution paths incorrect, and so make design errors. White box testing can find these errors.
 - Typographical errors are random. Just as likely to be on an obscure logical path as on a mainstream path.

"Bugs lurk in corners and congregate at boundaries"

January 19, 2005

6

White box testing

Here's code - spot the bug:

```
public class Serial {
    public synchronized int getMessage(byte[] msg) {
        return org.ovmj.boeingDemo2004.GetSetMessage.setMessage(msg);
    }

    public synchronized int setMessage(byte[] msg) {
        return org.ovmj.boeingDemo2004.GetSetMessage.setMessage(msg);
    }
}
```

Tell this horror story in your next software engineering class.

David

January 19, 2005

7

White box testing

• Control flow testing

- **Conditions Testing** -- Condition testing aims to exercise all logical conditions in a program module. Focus on testing each condition in the program. Strategies proposed include:
 - Branch testing - execute every branch at least once.
 - Domain Testing - uses three or four tests for every relational operator.
 - Branch and relational operator testing - uses condition constraints

January 19, 2005

8

White box testing

- Control flow testing
 - Loop testing
 - Simple Loops of size n:
 - Skip loop entirely
 - Only one pass through loop
 - Two passes through loop
 - m passes through loop where $m < n$.
 - (n-1), n, and (n+1) passes through the loop.
 - Nested Loops-
 - Start with inner loop. Set all other loops to minimum values.
 - Conduct simple loop testing on inner loop.
 - Work outwards
 - Continue until all loops tested.

January 19, 2005

9

Black box testing

- Focus on functional requirements
 - Attempts to find:
 - incorrect or missing functions
 - interface errors
 - errors in data structures
 - performance errors
 - initialization and termination errors

January 19, 2005

10

Black box testing

• Equivalence Partitioning

- Divide the input domain into classes of data for which test cases can be generated. Attempting to uncover classes of errors. Based on equivalence classes for input conditions. An equivalence class represents a set of valid or invalid states.
- An input condition is either a specific numeric value, range of values, a set of related values, or a boolean condition.

• Boundary Value Analysis

- Large number of errors tend to occur at boundaries of the input domain.
- BVA leads to selection of test cases that exercise boundary values..
- Rather than select any element in an equivalence class, select those at the "edge" of the class.

January 19, 2005

11

Static Verification and Analysis

• Program Verification

- If the programming language semantics are formally defined, can consider program to be a set of mathematical statements
- Can attempt to develop a mathematical proof that the program is correct with respect to the specification. If the proof can be established, the program is verified and testing to check verification is not required.
- There are a number of approaches to proving program correctness.
- Will only consider axiomatic approach. Suppose that at points $P(1), \dots, P(n)$ assertions concerning the program variables and their relationships can be made. The assertions are $a(1), \dots, a(n)$. The assertion $a(1)$ is about inputs to the program, and $a(n)$ about outputs.
- Can now attempt, for k between 1 and $(n-1)$, to prove that the statements between $P(k)$ and $P(k+1)$ transform the assertion $a(k)$ to $a(k+1)$.
- Given that $a(1)$ and $a(n)$ are true, this sequence of proofs shows partial program correctness. If it can be shown that the program will terminate, the proof is complete.

January 19, 2005

12

Static Verification and Analysis

• Static Program Analysers

- Static analysis tools scan the source code to try to detect errors.
- The code does not need to be executed.
- Can check:
 - Syntax, Unreachable code, Unconditional branches into loops,
 - Undeclared variables, Uninitialised variables, Parameter type mismatches,
 - Uncalled functions and procedures,
 - Variables used before initialisation,
 - Non usage of function results, Possible array bound errors,
 - Misuse of pointers.

January 19, 2005

13

Testing strategies

• testing is usually in the order:

1. **Unit testing** -- Module level testing with heavy use of white box testing techniques
 - Exercise specific paths in the modules control structures for complete coverage and maximum error detection.
2. **Integration Testing** -- Dual problems of verification and program construction
 - Heavy use of black box testing techniques
 - Some use of white box testing techniques to ensure coverage of major control paths.
3. **Validation Testing** -- Testing of validation criteria (established during requirements analysis).
 - Black box testing techniques used.
4. **System Testing** -- Part of computer systems engineering. Considering integration of software with other system components.

January 19, 2005

14

Testing

- **Unit testing:**
 - test individual (stand-alone) components
- **Module testing:**
 - test a collection of related components (module, package)
- **Sub-system testing**
 - test sub-system interface mismatches
- **System testing:**
 - (i) test interactions between sub-systems, and
 - (ii) test that the complete systems fulfill requirements
- **Acceptance testing (alpha/beta testing)**
 - test system with real rather than simulated data

Testing is iterative!

January 19, 2005

15

Regression testing

- **Regression testing** means testing that everything that used to work *still works* after changes are made to the system!
- tests must be deterministic and repeatable
- should test "all" functionality
 - every interface
 - all boundary conditions
 - every feature
 - every line of code
 - everything that can conceivably go wrong!

It costs extra work to define tests up front, but they pay off in debugging and maintenance!

Testing can only reveal the presence of defects not their absence!

January 19, 2005

16

The problem

"Testing is not closely integrated with development. This prevents you from measuring the progress of development—you can't tell when something starts working or when something stops working."

- Interactive testing is *tedious* and *seldom* exhaustive.
- Automated tests are better, but
 - how to introduce tests *interactively*?
 - how to organize *suites* of tests?

January 19, 2005

17

JUnit, a testing framework

- JUnit is an open-source testing framework (Gamma, Beck) that provides:
 - classes for writing Test *Cases* and Test *Suites*
 - methods for *setting up* an *cleaning up* test data ("fixtures")
 - methods for making *assertions*
 - textual and graphical tools for *running tests*
- It simplifies and automates the task of writing repeatable unit test.
- JUnit distinguishes between failures and errors:
 - A *failure* is a failed assertion, i.e., an anticipated problem that you check
 - An *error* is a condition you didn't check for.

January 19, 2005

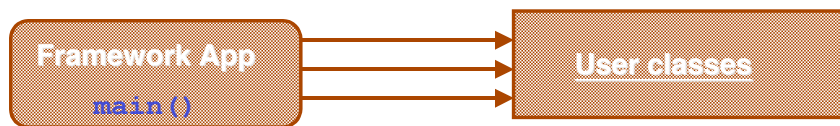
18

Frameworks vs Libraries

- In traditional application architectures, user-code uses library functionality by invoking methods defined in library classes



- A framework reverses the usual relationship between generic and application code. Frameworks provide both generic functionality and application architecture:



- Essentially a framework says: "Don't call me, I'll call you"
(*Frameworks are much harder to write*)

January 19, 2005

19

JUnit example

- Testing a FileReader class

```
class FileReaderTester extends TestCase {
    public FileReaderTester(String name) {
        super(name);
    }
    FileReader _input;
}
```

NB placing tests in a separate file implies a tradeoff.

- Separate the test code from the application code (+)
- No space overhead in the final application (+)
- No access to private/protected members (-)
- One more file to manage (-)

January 19, 2005

20

Setting up a test case

- The `TestCase` class provides two methods for manipulating test fixtures: `setUp` creates the objects used by the `TestCase`, `tearDown` removes them

```
class FileReaderTester...
    protected void setUp() {
        try { _input = new FileReader("data.text");
        } catch (FileNotFoundException e) {
            throw new RuntimeException("Unable to open file");}
    }
    protected void tearDown() {
        try { _input.close();
        } catch (IOException e) {
            throw new RuntimeException("Error closing file");}
    }
}
```

NB make sure to reset all static variables to their initial state in `tearDown`.

January 19, 2005

21

Writing tests

- A test file: "data.text"
`'Bradman 12 45 16\EOF'`

- A test

```
public void testRead() throws IOException {
    char ch = '&';
    for (int i=0; i < 4; i++)
        ch = (char) _input.read();
    assert('d'==ch);
}
```

- Invoking the test

```
class FileReaderTester
    public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTest(new FileReaderTester("testRead"));
        return suite; }
}
```

January 19, 2005

22

Running tests

- The main method

```
class FileReaderTester...
    public static void main(String[] args) {
        junit.textui.TestRunner.run(suite());
    }
```

- A run

```
.
Time 0.11

OK (1 tests)
```

Failures

- A deliberate bug:

```
public void testRead() throws IOException {
    char ch = '&';
    for (int i=0; i < 4; i++)
        ch = (char) _input.read();
    assert('#'==ch);
}
```

- Results

```
.F
Time: 0.22

!!!FAILURES!!!
Test Results:
Run: 1Failures: 1 Errors: 0
There was 1 failure:
1) FileReaderTester.testRead test.framework.AssertionFailedError
```

Failures (bis)

- A deliberate bug:

```
public void testRead() throws IOException {
    char ch = '&';
    for (int i=0; i < 4; i++)
        ch = (char) _input.read();
    assertEquals("fourth char read", '#', ch);
}
```

- Results

```
.F
Time: 0.22
!!!FAILURES!!!
Test Results:
Run: 1Failures: 1 Errors: 0
There was 1 failure:
1) FileReaderTester.testRead:
    fourth char read expected "#" but was "d"
```

January 19, 2005

25

Failures (bis²)

- An error:

```
public void testRead() throws IOException {
    char ch = '&';
    _input.close();
    for (int i=0; i < 4; i++)
        ch = (char) _input.read();
    assertEquals("fourth char read", '#', ch);
}
```

- Results

```
....
Run: 1 Failures: 0 Errors: 2
There was 1 error:
1) FileReaderTester.testRead: java.io.IOException: Stream closed
```

NB always start by triggering an error, to be sure a test is actually run.

January 19, 2005

26

Testing Style

"The style here is to write a few lines of code, then a test that should run, or even better, to write a test that won't run, then write the code that will make it run."

- write unit tests that thoroughly test a single class
- write tests as you develop (even before you implement)
- write tests for every new piece of functionality

"Developers should spend 25-50% of their time developing tests."

January 19, 2005

27

What to test?

- FileReader returns -1 at the end of a file

```
public void testReadEnd() throws IOException {
    char ch = '&';
    for (int i=0; i < 16; i++)
        ch = (char) _input.read();
    assertEquals(-1, ch);
}
```

- Invoking the test

```
class FileReaderTester
    public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTest(new FileReaderTester("testRead"));
        suite.addTest(new FileReaderTester("testReadEnd"));
        return suite; }
}
```

January 19, 2005

28

Shortcut

- Instead of using the suite() method, write

```
public static void main (String[] args) {  
    junit.textui.TestRunner(new TestSuite(FileReaderTester.class));  
}
```

tests are extracted by reflection

What to test?

- Test boundary conditions

```
public void testReadBoundaries() throws IOException {  
    assertEquals("read first char", 'B', _input.read());  
    int ch;  
    for (int i=0; i < 15; i++)  
        ch = _input.read();  
    assertEquals("read last char", '6', _input.read());  
    assertEquals("read at end",-1,_input.read());  
}
```

What to test?

- Test special conditions

```
public void testEmptyRead() throws IOException {
    File empty = new File("empty.text");
    FileOutputStream out = new FileOutputStream(empty);
    out.close();
    FileReader in = new FileReader(empty);
    assertEquals(-1, in.read());
}
```

What to test?

- Test exceptions

```
public void testReadAfterClose() throws IOException {
    _input.close();
    try {
        _input.read();
        fail("no exception for read past end");
    } catch (IOException io) {}
}
```

8 rules of testing

1. Make sure all tests are fully automatic and check their own results
2. A test suite is a powerful bug detector that decapitates the time it takes to find bugs
3. Run your tests frequently - every test at least once a day.
4. When you get a bug report, start by writing a unit test that exposes the bug
5. Better to write and run incomplete tests than not run complete tests
6. Think of boundary conditions under which things might go wrong and concentrate your tests there
7. Don't forget to test exceptions raised when things are expected to go wrong
8. Don't let the fear that testing can't catch all bugs stop you from writing tests that will catch most bugs

- Martin Fowler, *Refactoring*