Software Testing

Michael A. Erlinger

Software Testing: Convention Wisdom

95% of errors are in 5% of the code

...or maybe 80% of the errors are in 20% of the code...

Testing:
Can I find that 5% or 20% via a test suite

Software Testing Topics

- Goals
- Types of tests
- Levels of tests
- Test measures
- Test plan
Goals

- Verification: Have we built the software right?
  Bug-free, meets specs

- Validation: Have we built the right software?
  Meets customers' needs/requirements

Are Verification and Validation different or variation of the same idea?
My argument: “meeting specs” should equal “meeting customer needs”...
  not generally true (my kitchen, satellite systems)

Validation

This isn't Sudoku!

Software Testing

Users/Customers' perspective

That should be a 7!

Software Testing

Users/Customers' perspective

That shouldn't take a 7!

... and this kind of problem
Software Testing

• Goals
• Types of tests
• Levels of tests
• Test measures
• Test plan

most of our discussion focuses on Verification (more specifically bug testing)

Types of tests

• Black box
• White box
• Gray box – will not talk about

Black Box Tests

input ➔ interface ➔ output

1. Does it perform the specified functions?
2. Does it handle obvious errors in input?
3. Ariane5 – lousy error handling
4. Classic ints vs floats, yards vs meters
   Black box should catch these if there is adequate “test coverage”

Example: Ordered List of ints

L=create()
L.insert(5)
L.insert(-1)
L.insert(-1)
p=L.getFirst()
print(p)
L.delete(p)
p=L.getFirst()
print(p)
p=L.getNext(p)
print(p)
p=L.getNext(p)

-1
-1
5 error
Try to break my code (please)

Users/Customers' perspective

- Test through interface
- By someone other than the developer (usually)
- After the developer has finished debugging
  - Standard approach in Waterfall, e.g., DoD

"Black box testing" – we test external behavior.

Black Box Tests

Choose good distribution of input – hope good distribution of code tested

How would you choose????

BB - Unnecessary Tests

Large range of input may exercise a small part of code
  e.g., operator test of satellite control stations, run through each
  input and output light/key option. Testing same functions, whereas
  no one had a test for my map function.

BB: Insufficient Tests

A small range of input may exercise a large range of code
  but can you ‘know’ this without knowing the code? Did we miss the 20%
BB - Sufficient Tests

Input

Code

complex code

a small range of input may exercise a small but important/error-prone region of code....who knows where this is?

Black box tests

- **Advantage**: black box tester≠developer is unbiased by implementation details. e.g., Use Case testing, just work through all the Use Cases, building tests for each...what middle school students will do
- **Disadvantage**: black box tester is uninformed about implementation details
  - unnecessary tests – test same thing in different way
  - insufficient tests – can miss the extremes, especially if actual use follows a different pattern – generally impossible to check all inputs and all paths

White Box Tests

Based on code

- Test through code
- By the developer (usually)
- Throughout development

“White (or glass) box testing” – we test internal behavior.

Developers’ perspective

```java
(bool) testIsConsistentForBadRow
{
  [testModel isConsistentAtRow: -1 column: 5 forValue: 7];
  // evaluate whether error occurred
  ...
}
```
Levels of tests

• Unit
• Integration
• System
• GUI

“White (or glass) box testing” – we test internal behavior.

Developers’ perspective

How can we assess the “quality” of our white box testing?

• Test through code
• By the developer (usually)
• Throughout development

doSomething (x,y,z)
{
  if (x<=y)
    x++;
  if (y<=z)
    y++
  if (z<=x)
    z++
}
doSOMething(5,5,5) achieves 100% code coverage
All lines of code are tested. Is that good?

doSOMething(5,5,5) covers one execution path (all ifs true)

doSOMething(6,5,4) covers a different path (no ifs true, or ?)

Can we achieve 100% path coverage?
**Stats**

If

- 80% of errors are in 20% of the code

or

- 50% of the errors are in 5% of the code

then what do we want to do??

---

**Identifying “complex” code**

- Cyclomatic complexity measure (McCabe)
  - Counts linearly independent paths
  - Tools exist

- Developer’s best guess – what regions are likely to be problematic

- Test results – tell us a lot about the code

---

**Complexity based coverage:**
Test the most complex, error-prone code the most

```
Code
```

Where most of the bugs are.

---

**Test Results: Code**

```
A successful test is one that finds a bug!
```

```
Successful test

Failed test
```
A successful test is one that finds a bug!

The fact that we've found lots of bugs here means we should design more tests for this region or that we do not understand the problem and solution.

Therefore in writing a method... First I write the tests which specify how the method should behave.

- (void) isConsistentAtRow: (int) row Column: (int) column forValue: (int) value

Testing on random input does not (typically) work well. Looks like Black Box

Error handling

- During development
- Release code

WB: unnecessary tests?
WB: insufficient tests?

Input

Code

Unit test view: test small piece of code (method, class)
Not enough.....

Code

- Successful test
- Failed test

WB: Integration

Code

What happens when they interact?
White Box Tests

• Advantage:
  – design tests to achieve good code coverage and avoid duplication
  – can stress complicated, error-prone code
  – Can be driven from Design, e.g., Use Cases or other Models
  – can stress boundary values (fault injection)

• Disadvantage:
  – tester=developer may have bias (team alternatives to test developer)
  – if code changes, tests may have to be redesigned (is this bad?)

Software Testing Rule

Agile: test continuously

Good

• Write code
• Write tests
• Test code

Better

• Write tests
• Write & test code

TDD: Test-driven development

-(void) isConsistentAtRow: (int) row Column: (int) column forValue: (int) value

If I ask if 7 is consistent for row 6, column 3 the method should say no.
-(void) isConsistentAtRow: (int) row Column: (int) column forValue: (int) value

If I ask if 6 is consistent for row 6, column 3 the method should say yes.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>8</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>4</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If I ask if 7 is consistent for row -1, column 3 the method should say ____.

A well-defined test must know what the outcome of the test should be...sometimes the hard part

-(bool) isConsistentAtRow: (int) row Column: (int) column forValue: (int) value

TDD forces us to consider how the method behaves on good input and bad before we code it

An Asside....

Model* theModel = [Model alloc];
if (theModel)
{
... 
}
else {
...
}

Error: Your are here!

In release:
Need to Handle it gracefully

If we cannot recover: save our data, write to a log, then give up, but generate

EXCEPTION

In development:
Shout!

Model* theModel = [Model alloc];
if (theModel)
{
...
}
else {
...
}
setValueAtRow: row Column: column to: value
{
if (row<0 || row>8)
{
Shout
}
... 

Shout
Model* theModel = [Model alloc];
assert(theModel!=nil)
...

In development:
Our Shout!

setValueAtRow: row Column: column to: value
{
    assert (row>=0 && row<=8)
    ...}

9/21/13

In case you haven’t noticed, in Xcode you can build a debug version or a release version. During development we want to build debug versions.

A well-defined test must know what the outcome of the test should be.

-(void) isConsistentAtRow: (int) row Column: (int) column forValue: (int) value

If this is my current grid and I ask if 7 is consistent for row -1, column 3 the method should ____.

Assert a problem.

- (bool) isConsistentAtRow: (int) row Column: (int) column forValue: (int) value

1. Write tests.
2. Write & test code.

testsConsistentForYesCase
testsConsistentForNoCase
testsConsistentForBadRow
testsConsistentForBadColumn
testsConsistentForBadValue

Code written and passes all the tests
-(void) isConsistentAtRow: (int) row Column: (int) column forValue: (int) value

Test Suite

testIsConsistentForYes
testIsConsistentForNo
testIsConsistentForBadRow
testIsConsistentForBadColumn
testIsConsistentForBadValue
testSetValueForGoodInput
...

1. Write tests.
2. Write & test code.
3. Add tests to test suite.

Goal of regression testing: make sure changes don’t break anything

Summary: WB
Testing measures

• Code coverage – individual modules
• Path coverage – sequence diagrams
• Code coverage based on complexity – test of the risks, tricky part of code (e.g., Unix “you are not expected to understand this” code)

Summary WB: Code Coverage

• How much of code is “tested” by tests?
  – manually
  – profiling tools
• Design new tests to extend coverage
• Is 100% good? Is it possible to achieve?
Summary WB: Path Coverage
• How many execution paths have been exercised by tests?
• 100% path coverage is usually impossible
• Aim to cover common paths and error prone (complex) paths – sometimes discovered too late
• Aim to break code with tests – good testers are not liked by developers....

Summary WB: Code Complexity Measures
• cyclomatic complexity measure (McCabe)
  – measures number of linearly independent paths through program
  – ignores asynchronous interaction, fallibility of services
  – etc.
• Developer’s best guess – what problems are likely and which will be hardest to diagnose
• started with Knuth and others who gathered stats on programs – used in my dissertation

Summary WB: Test plan
• Collection of tests: unit, integration, system
• Rationale for test: why these tests?
• Strategy for developing/performing tests
  – e.g., test units as developed, test integration at each build, run the same tests over each build, etc.
  – Final test plan for Version 1 should do all the unit tests, integration tests, etc.

Testing Strategy
• TDD – test driven development
  Write test BEFORE your write the code!
• Regression
• Test harness
• Bug tracking
• User tests
TDD – test Driven

Unit tests are written first by the SEs. Then as code is written the code passes incrementally larger portions of the test suites. The test suites are continuously updated as new failure conditions and corner cases are discovered, and integrated with regression tests. Unit tests are maintained like software and integrated into the build process. The goal is to achieve continuous deployment with frequent updates.

Testing Strategy

- TDD – test driven development
- Regression
- Test harness
- Bug tracking
- User tests

Test Harness

Useful for interactive graphics applications
automated test framework, built one for satellite console to run through all console interactions

Test harnesses allow for the automation of tests. They can call functions with supplied parameters and print out and compare the results to the desired value. The test harness is a hook to the developed code, which can be tested using an automation framework.
Strategy

- TDD – test driven development
- Regression
- Test harness
- Bug tracking

  Design system for tracking
  Scrupulously record problems, their context, the effect, ideas on the cause, attempts to fix, create new tests as bugs uncovered

- User test

9/21/13

Test Plan

- Collection of tests: unit, integration, system
- Rationale for test: why these tests?
- Strategy for developing/performing tests

  Be thoughtful in designing
  Be diligent in executing

9/21/13

Example: ordered list of ints

class ordInts {
  public: ... 

  private: 
  int vals[1000];
  int maxElements=1000; 
  ...
}

9/21/13
White Box TestExample: ordered list of ints

bool testMax()
{
    L=create();
    num=maxElements;
    for (int i=0; i<=num; i++)
        print i
        L.insert(i)
    print maxElements;
}

Types of Tests

- Black box: test based on interface, through interface
- White box: test based on code, through code

Testing strategy should include all approaches!

My experience:
Black Box = non developer, outside testing “idiots” who follow specifications/requirements.
in your case who?
White Box = developer, part of development process
in your case who?