Notes 3: Verification & Validation (V & V)

James M. Bieman

Software Failures

- IRS Automated Income Tax Form Processing System (Sperry 1980's).
- SDI Star Wars software.
- Ariane-5 Rocket.
- Therac-25 Accidents.
- Year-2000 bug.
- London Ambulance Service Fiasco.
- MS Zune failure on December 31, 2008.
- ASCSU Course Survey failures (2011 - ?).
- Affordable Care Rollout.

Focus: Evaluating Software Quality

- Verification and validation (V & V) techniques and terminology.
- Testing theory.
- Functional (Black box) and structural (white box) testing.
- Test plans.
- Inspections.
- All tied to testing object-oriented software.

Validation

- Validation refers to checking to make sure that we are building what the customer wants.
- We ask:
  
  “Did we build the right thing?”

Verification

- Verification refers to checking to see if we have built the software so that it matches some specification.
- We ask:

  “Did we build the thing right?”

Testing

- Run the program on sample inputs.
- Check the correctness of the output.
- Test run success is evidence of correctness.
- Testing is part of either verification or validation.
**V & V Is Not Just Applied to Code**

- V & V techniques can be applied to non-running software documents.
  - Requirements specifications.
  - Designs.
  - Test plans.

**V & V Techniques**

- Static analysis: we do not run anything.
- Formal verification: mathematical proofs.
- Dynamic analysis: usually testing.
- Inspections: semi-formal study of a software document.

**V & V Terminology**

- Software Fault: a static defect in the software.
- Software error: an incorrect internal state caused by a fault.
- Software Failure: external incorrect program behavior with respect to the explicit or expected requirements.


**Testing Terminology**

- Unit testing: testing a procedure, function, or class.
- Integration testing: testing connection between units and components.
- System testing: test entire system.
- Acceptance testing: testing to decide whether to purchase the software.

**A High-Level View of Testing Object-oriented Systems**

- System tests: may be developed from
  - Use cases, and
  - System sequence diagrams
  More on these UML diagrams later.
- Integration tests:
  - Use cases and system sequence diagrams.
  - Subsystem sequence diagrams.
- Unit tests:
  - Packages of classes.
  - Method combinations.
  - Individual methods.
Testing Terminology (2)

- Alpha testing: system testing by a user group within the developing organization.
- Beta testing: system testing by select customers.
- Regression testing: retesting after a software modification.

Dynamic Fault Classification

- Logic faults: omission or commission.
- Overload: data fields are too small.
- Timing: events are not synchronized.
- Performance: response is too slow.
- Environment: error caused by a change in the external environment.

Who Should Conduct Testing?

- Should the developer do the testing?
- Should we use an independent testing team?
- How is it done in industry?

Test Scaffolding or Test Harness

Allows us to test incomplete systems.

- Test drivers/harnesses: test components.
- Stubs: test a system when some components it uses are not yet implemented.
  - Often a short, dummy program — a method with an empty body.

Test Oracles

- Determine whether a test run completed with or without errors.
- Often a person, who monitors output.
  - Not a reliable or efficient method.
- Automatic oracles check output using another program.
  - Requires some kind of executable specification.
  - Asserts in JUnit.

Testing Theory: Why Is Testing So Difficult?

- Theory often tells us what we can’t do.
- Testing theory main result: perfect testing is impossible.
An Abstract View of Testing

- Let program $P$ be a function with an input domain $D$ (i.e., the set of all ints).
- We seek test data $T$, which will include selected inputs of type $D$.
  - $T$ is a subset of $D$.
  - $T$ must be of finite size.
  Why?

We Need a Test Oracle

- Assume the best possible oracle --- the specification $S$, which is function with input domain $D$.
- On a single test input $i$, our program passes the test when $P(i) = S(i)$

For Perfect Testing

1. If all of our tests pass, then the program is correct.
   - All of our tests $t$ in test set $T$, $P(t) = S(t)$, then we can be sure that the program will work correctly for all elements in $D$.
   - If any tests fail we look for a fault.
2. We can tell whether the program will eventually halt and give a result for any $t$ in our test set $T$.

But, Both Requirements Are Impossible to Satisfy.

- 1st requirement can be satisfied only if $T = D$.
  We test all elements of the input domain.
- 2nd requirement depends on a solution to the halting problem, which has no solution.

Other Undecidable Testing Problems

- Is a control path feasible?
  Can I find data to execute a program control path?
- Is some specified code reachable by any input data?
  These questions cannot, in general, be answered.

Software Testing Limitations

- There is no perfect software testing.
- Testing can show defects, but can never show correctness.
  We may never find all of the program errors during testing.
A Pragmatic Testing Strategy

- Divide domain $D$ into sub-domains $D_1, D_2, \ldots, D_n$, which represents some aspect of the program.
- Select at least one test case from each $D_i$.

We cannot test each sub-domain perfectly, but we can do better on a piece of the functionality.

Software Faults, Errors & Failures

- **Software Fault**: A static defect in the software
- **Software Error**: An incorrect internal state that is the manifestation of some fault
- **Software Failure**: External, incorrect behavior with respect to the requirements or other description of the expected behavior

Faults in software are design mistakes and will always exist.

Fault & Failure Model

Three conditions necessary for a failure to be observed:

1. **Reachability**: The location or locations in the program that contain the fault must be reached.
2. **Infection**: The state of the program must be incorrect.
3. **Propagation**: The infected state must propagate to cause some output of the program to be incorrect.

Black-Box Class Testing

- **Black-Box testing**: test a “component” taking an external view.
  - Use the specification to derive test cases.
  - No access to source code.
- **Black-Box class testing**.
  - Generate tests by analyzing the class interface.
  - Don’t look at method bodies.

Black-Box Class Testing (2)

- Look at the class in isolation, and in conjunction with other associated classes.
- Test each class method, and test sequences of messages that class objects should respond to.
- May need stubs and/or test drivers.

Black-Box Class Testing (3)

- Group class objects into categories.
- Test each method for each category.
- A test plan documents all of the tests to be performed.
An Infinite Number of Possible Inputs, But a Finite Number of Tests

Partition Inputs and Test Boundaries

Determining Equivalence Partitions

- Find an ordering of the class objects.
- Example: Finding Java String objects to test.
  - Canonical ordering of string objects:
    - "", "a", "b", "aa", "ab", "zz...zz"
  - Use the ordering to select test objects.

Use the Ordering to Select Test Cases

- Find objects at extremes and next to extremes:
  - Minimum size: "", "a"
  - Long strings: "zz...zz", "zz...zy"
  - Middle length Strings
  - Different types of Strings:
    - numbers
    - control characters: "^D^C"
    - symbols: "&@+->
  - Invalid strings: a null String variable. For C++, you can set a String variable to an integer.

Ex: Testing Java Class Stack

Stack method interface:
- boolean empty()
- Object peek()
- Object pop()
- void push(Object element)
- int search(Object element)

Boo Hiss! This is non-stack type of operation.
- Object Stack(): The default constructor.

Classify the Operations

- Constructors/Destructors:
  - Stack()
  - ~Stack() in C++
- State changing operations:
  - pop()
  - push(Object e)
- Non-state changing operations:
  - empty()
  - peek()
Test Each Type of Operation

- Constructors: test each constructor with all orderings of parameter boundary values.
- Destructors: test with each constructor.
- State changing operations: try to change the object state from every "state" to every other "state".
- Non-state changing operations: test on stacks in each "state".

"State"

- Really a group of related states.
- Example stack states:
  - Empty stacks,
  - Mid-size stacks,
  - Just under the maximum size stack,
  - Large or full stacks.
  - Empty stacks,
  - Mid-size stacks,
  - Just under the maximum size stacks.

Testing Multiplicity

- Create several stacks and test, alternating between them.
- This will determine whether each stack object has an independent state (independent instance variables).

The Test Oracle

- How do you know if an item is successfully pushed onto a stack?
- Examine the behavior of the resulting stack after the push operation is performed.

Class Testing Plan Structure

- Class name.
- For each public method:
  - Method name.
  - For each test case for the method:
    - A test ID.
    - Test strategy: black-box (BB), white-box (WB), or other; test of valid or invalid input?
    - Test description.
    - Verification: what are the expected outputs? How do you identify success or failure?

Class Stack BB Test Plan

Constructor method Stack() tests:
- Test: Stack.Stack1
  - Strategy: Black Box, Valid.
  - Description: Create a Stack.
  - Verification: A stack object is created; a non-null Stack reference is returned.
- Test: Stack.Stack2
  - Strategy: Black Box, Valid.
  - Description: Create many Stacks.
  - Verification: Many stack objects are created; non-null, not equal Stack references are returned.
Class Stack BB Test Plan (2)
Method push(Object e) tests:
- Test: Stack.push1
  - Strategy: Black Box, Valid
  - Description: Push one item onto a Stack.
  - Verification: The item is on the top of the stack and can be popped off.
- Test: Stack.push2
  - Strategy: Black Box, Valid
  - Description: Push many items onto a Stack.
  - Verification: The items can be popped off in reverse order.

Class Stack BB Test Plan (3)
Method push(Object e) tests:
- Test: Stack.push3
  - Strategy: Black Box, Valid
  - Description: Push many items onto a Stack.
  - Verification: The correct items can be popped off each stack in reverse order.
- Test: Stack.push4
  - Strategy: Black Box, Valid
  - Description: Push a null item onto a Stack.
  - Verification: The null object can be popped off.

Class Stack BB Test Plan (4)
Method push(Object e) tests:
- Test: Stack.push5
  - Strategy: Black Box, Invalid
  - Description: Push an item onto a null Stack.
  - Verification: An exception is raised.
- Test: Stack.push6
  - Strategy: Black Box, Invalid
  - Description: Push a null item onto a non-Stack object.
  - Verification: It won’t compile in Java; An exception is raised in C++.

Class Stack BB Test Plan (5)
Method pop() tests:
- Test: Stack.pop1
  - Strategy: Black Box, Valid
  - Description: Pop 1 item from a Stack.
  - Verification: The item can be popped off.
- Test: Stack.pop2
  - Strategy: Black Box, Valid
  - Description: Pop many items from a Stack.
  - Verification: The items can be popped off.

Class Stack BB Test Plan (6)
Method pop() tests:
- Test: Stack.pop3
  - Strategy: Black Box, Valid
  - Description: Pop many items from a Stack.
  - Verification: The item can be popped off in reverse order.
- Test: Stack.pop4
  - Strategy: Black Box, Valid
  - Description: Pop a null item from a Stack.
  - Verification: The item can be popped off.

Class Stack BB Test Plan (7)
Method pop() tests:
- Test: Stack.pop5
  - Strategy: Black Box, Invalid
  - Description: Pop a null Stack.
  - Verification: An exception is raised.
- Test: Stack.pop6
  - Strategy: Black Box, Invalid
  - Description: Pop a non-Stack object.
  - Verification: Will not compile in Java; raises an exception in C++
**Class Stack BB Test Plan (8)**

Method pop() tests:
- Test: Stack.pop7
  - Strategy: Black Box, Invalid
  - Description: Pop an empty Stack.
  - Verification: An exception is raised.

Method empty() tests:
- Test: Stack.empty1
  - Strategy: Black Box Valid
  - Description: Test a newly created Stack.
  - Verification: Returns true.

**Class Stack BB Test Plan (9)**

Method empty() tests:
- Test: Stack.empty2
  - Strategy: Black Box Valid
  - Description: Test a Stack with a history of 1 push and 1 pop.
  - Verification: Returns true.
- Test: Stack.empty3
  - Strategy: Black Box, Valid.
  - Description: Test a Stack with many pushes, and an equal number of pops.
  - Verification: Returns true.

**Class Stack BB Test Plan (10)**

Method empty() tests:
- Test: Stack.empty4
  - Strategy: Black Box Valid
  - Description: Test after a push-pop-push-pop sequence.
  - Verification: Returns true.
- Test: Stack.empty5
  - Strategy: Black Box, Valid.
  - Description: Test after 1 push.
  - Verification: Returns false.

**Class Stack BB Test Plan (11)**

Method empty() tests:
- Test: Stack.empty6
  - Strategy: Black Box Valid
  - Description: Test after many pushes.
  - Verification: Returns false.
- Test: Stack.empty7
  - Strategy: Black Box, Invalid.
  - Description: Test a null Stack.
  - Verification: Exception.

**Class Stack BB Test Plan (12)**

Method empty() tests:
- Test: Stack.empty8
  - Strategy: Black Box Invalid
  - Description: Test a non-Stack.
  - Verification: Won’t compile in Java. Exception in C++.

**Test Drivers/Harnesses**

- Must be able to:
  - Build the test cases.
  - Log testing results.
  - Make success or failure observable.
- Can be
  - Hard-coded.
  - Reads tests from a file.
  - Interactive.
  - Built using a tool like Junit.
- Each test should run independently.
Example Class Test Driver

Handcrafted test driver:
- No use of JUnit or similar tool.
- Not recommended. For demonstration purposes.

```java
public class StackTest {
    public static void main (String[] args)
        throws IOException{
        /** We run the tests **/
        push1();
        push5(); // No crash after invalid test!
        push2();
        push3();
        push5();
    }
}
```

Test Stack.push.1 Test Plan

```java
public static void push1() {
    System.out.println( "\n"
    + "Test StackPush1. Test Type: BB, Valid \n"
    + "  Description:  Push 1 item onto a stack. \n"
    + "  Verification: Item is there and can be 
"
    + "  popped off.");
}
```

Now Run Test Stack.push.1

```java
try{
    Stack s = new Stack();
    String i1 = "Item 1";
    s.push(i1);
    System.out.println( "Testing Results: \n"
    + "  Item i1: " + i1 + "\n"
    + "  After s.push(i1); The top is: " + s.peek()
    + "\n"
    + "  Is i1==s.pop()? " + (i1==s.pop()));
} catch (Exception e){
    System.out.println("Testing Results: Error. \n   " + e);
}
```

Test Stack.push.2 Plan

```java
public static void push2() {
    System.out.println( "\n"
    + "Test Stack.push.2, Test Type: BB, \n"
    + "Valid \n"
    + "  Description: Push many items onto 
"
    + "  a stack. \n"
    + "  Verification: Items can be popped 
"
    + "  off in reverse order.");
}
```

Now Run Test Stack.push.2

```java
try{
    Stack s = new Stack();
    String i1 = "Item 1";
    String i2 = "Item 2";
    String i3 = "Item 3";
    String i4 = "Item 4";
    String i5 = "Item 5";
    s.push(i1);
    s.push(i2);
    s.push(i3);
    s.push(i4);
    s.push(i5);
}
```

Completing Stack.test.2

```java
Completing Stack.test.2

System.out.println( "Testing Results: \n"
    + "  After pushing items 1 through 5 onto 
"
    + "  Stack s, \n"
    + "  we try to pop them off:"); 
    for (int i=1; i<6; i++)
        System.out.println(" + " + i + ", s.pop(): " + 
(s.pop()));
} // end try
```
Catch Stack.test.2 Failures

```java
try {
    catch (Exception e)
    {
        System.out.println("Testing Results:" + " Error. \n " + e);
    }
}
```

Test Stack.push.3

```java
public static void push3()
    System.out.println("\n" + 
    "Test Stack.push.3, Test Type: BB, Valid \n" + 
    " Description: Push many items onto many stacks. \n" + 
    " Verification: Items can be popped off in reverse order.");
```

Run Test Stack.push.3

```java
try {
    Stack s1 = new Stack();
    Stack s2 = new Stack();
    Stack s3 = new Stack();
    String i1 = "Item 1"
    String i2 = "Item 2";
    String i3 = "Item 3";
    String i4 = "Item 4";
    String i5 = "Item 5";
    String i6 = "Item 6";
    s1.push(i1);
    s2.push(i2);
    s3.push(i3);
    s3.push(i4);
    s2.push(i5);
    s1.push(i6);
    for (int i=1; i<3; i++)
        System.out.println(" 
    " + i + ". s1.pop(): " + (s1.pop()));
    for (int i=1; i<3; i++)
        System.out.println(" 
    " + i + ". s2.pop(): " + (s2.pop()));
    for (int i=1; i<3; i++)
        System.out.println(" 
    " + i + ". s3.pop(): " + (s3.pop()));
} catch (Exception e)
    System.out.println("Testing Results: Error. \n " + e);
```

More Test Stack.push.3

```java
System.out.println( "Testing Results:
" + " After pushing items 1 through 6 onto Stack s1, s2, s3,
" + " we try to pop them off.");
    System.out.println(" s1 has items pushed in order 1, 6:");
    System.out.println(" s2 has items pushed in order 2, 5:");
    for (int i=1; i<3; i++)
        System.out.println(" s3 has items pushed in order 3, 4:");
    for (int i=1; i<3; i++)
        System.out.println(" " + i + ", s1.pop(): " + (s1.pop()));
    System.out.println( + " + i + ", s2.pop(): " + (s2.pop()));
    System.out.println( + " + i + ", s3.pop(): " + (s3.pop()));
```

Complete Stack.push.3

```java
System.out.println( " s3 has items pushed in order 3, 4:");
    for (int i=1; i<3; i++)
    { System.out.println(" + i + ", s3.pop(): " + (s3.pop()));
    }
    catch (Exception e)
    { System.out.println(Testing Results: Error. \n " + e);
    }
```

Stack.push.5

```java
public static void push5()
    System.out.println( "\n" + "Test Stack.push.5, Test Type: BB, Invalid \n" + 
    " Description: Try to push onto a stack variable which \n" + 
    " is not instantiated (a null stack).\n" + 
    " Verification: Exception is raised.");
```
Run Stack.push.5

```java
try{
    Stack s = null;
    String theItem = "The Item";
    s.push(theItem);
    System.out.println("Testing Results: \n" + " Item theItem: " + theItem + "\n" + "After s.push(i); The top is: " + s.peek() + "\n" + "Is theItem==s.pop()? " + (theItem==s.pop()));
}
```

Now We Really Catch an Exception

```java```
catch (Exception e){
    System.out.println("Testing Results
Error. \n" + e);
}
```

Black box testing example

Program specification:
- The program receives an invoice as input (invoice structure is excluded here).
- The invoice must be inserted into an invoice file that is sorted by date.
  - It must be inserted in the appropriate position: if other invoices exist in the file with the same date, then the invoice should be inserted after the last one.
  - Consistency checks must be performed: the program should verify whether the customer is already in a corresponding file of customers, whether the customer’s data in the two files match, …

Test set
- Invoice whose date is the current date
- Invoice whose date is before the current date
  - Invoice whose date is the same as that of some existing invoice
  - Invoice whose date does not exist in invoice file
- Incorrect invoices that can be used to check different types of inconsistencies

Testing boundary conditions

Some programming errors are on the boundary of input domains/partitions used for testing.

```java```
if x > y then
dosomething;
else
dosomethingelse
end if;
```

Input domains
- \( D_1: \{ x > y \} \)
- \( D_2: \{ x \leq y \} \)

Rule of thumb: test using values at the boundaries of the input domains.

Example: String Reversal

Program Error.

Algorithm:
1. Divide input string into fixed-sized pages.
2. Push each page onto a stack.
3. Pop the characters out in reverse order.
Black Box Tests

- Vary length strings:
  - Empty strings,
  - Short strings,
  - Long strings,
  - Medium length strings.
- All might pass the tests.

Hidden Bug (Fault)

- The programmer assumed that the last page is partially full.
  - The program appends a "null" termination character, only when the last page is partially full.
- If the input string is an exact multiple of the page size, there is no partial page.
- The termination character ends execution. Without it the program fails.

Failures Occur "Rarely"

- Assume that the page length is 100 characters.
  - There is a 1% chance that, during black-box testing, the fault is revealed.
- The specifications do not include mention of the termination character.
- With white-box testing we would develop test cases around the specific code dealing with the termination character.
  - Test a case where the termination character is not appended.

Structural (White Box) Test Coverage Criteria

- Statement or node coverage.
- Branch coverage, edge coverage, or decision coverage.
- Condition coverage.
- Definition/Use (DU) Pair coverages.

Test Coverage Strength

- Branch coverage is stronger than statement coverage.
- Condition coverage is stronger than branch coverage, and
- Definition/Use coverage is stronger than branch coverage.

If tests satisfy a coverage criteria, they also satisfy all weaker ones.

Example

- Look at the code:
  - if (A) S1;
  - S2;
- We can cover both S1 and S2 with 1 test. Just set A=true.
- To cover all branches, we must also test the path that skips S1.
  - We need another test case where A=false.
Sometimes Stronger Coverage is Needed

- Buggy code:
  
  ```
  i = 0;
  if (A) i = 1;
  x = y/i;
  ```

- No error when you test with A=true.
- Bombs if you test with A=false.

Branch coverage reveals the error, but statement coverage may not!

An Error, Not Detected by Branch Coverage

```java
... if (A() && B()) x = y + z;
... boolean B() {
  if (F2) {
    q = 0;
    return true;
  } else {
    x = 10/q;
    return false;
  }
}
```  

We Test the Code

- Branch coverage is satisfied with 2 tests:
  - F1=true and F2=true: takes the true path.
  - F1=false and F2=false: takes the false path.
- The error occurs when F1=true & F2=false.
- Condition coverage or DU pairs coverage would require this test.

Tests and Test Paths

- **Test Path**

  - Test 1
  - Test 2
  - Test 3

Deterministic software – a test always executes the same test path

  - many-to-one

Non-deterministic software – a test can execute different test paths

  - many-to-many

Definitions & Uses

- **Definition**: the point in a program where a variable’s value is set or changed.
- **Use**: the point where a variable’s value is used.
- **DU-path**: a program path from a variable definition to a use, without an intervening definition to the variable.
Definition/Use (DU) Pair Coverage

The All-Uses Coverage Criterion

- For each variable definition:
  Test a def-free path to each use of the definition.
  (Test one DU path for each DU-pair for each variable.)
- In prior example: we would need include a DU path from definition “q = 0;” to the use “x = 10/q;”

Another Example Program

```java
while (notDone) do {
    if (A) x = f(x);
    else   x = g(x);
    ...
}
```

- then branch:
  First references the prior value of x (a use of x) & then redefines x (a definition of x).
- else branch:
  Does the same thing.

Test paths required by the all-uses criterion:
- Loop through the then branch twice in a row.
- Loop through the else branch twice in a row.
- 1 cycle through the then branch followed by a cycle through the else branch.
- A cycle through the else branch followed by a cycle through the then branch.

Testing Limitations

- If our testing results in:
  - 100% statement coverage,
  - 100% branch coverage,
  - 100% condition coverage,
  - 100% DU-pair coverage.
- The program may still have hidden faults.
  Why is that true?

White Box Testing Support Tools

- Instrument source code to report on program items that are "covered" during testing.
- Many tools exist. Search with the following search words: "java test coverage tools"
  - EMMA: statement coverage.
  - EclEmma, which is similar to Emma, but works with Eclipse.
  - CodeCover: Includes statement, branch, and condition-term coverage.

Software Inspections

- Semi-formal evaluation of software products for V&V.
- Organized with 2 or more “inspectors”.
- Objective: find errors early.

What to Inspect

All software documents can be reviewed:
- Requirements specifications: are they complete? Are they correct?
- Designs: do they satisfy all requirements? Is the design too complex? Are there errors?
- Code: look for faults.
- Test plans: completeness, correctness.
Inspections Focus on Goals

• Find and record errors.
• Don’t repair them.
• Participants review software documents independently and then meet to review & report findings. (Meetings can be virtual).

Review Guidelines [Pressman]

1. Review the product, not the producer.
2. Set an agenda and maintain it.
3. Limit debate and rebuttal.
4. Enunciate problem areas.
5. Take written notes.

Review Guidelines [Pressman]

6. Limit the number of participants & insist on advance preparation.
7. Develop & use a review checklist.
8. Allocate resources & time schedule.
9. Conduct training for all reviewers.
10. Review your early reviews.

Software Documents Are Meant to Be Read by People

• Commercially successful software will be modified many times over many years by many people.
• Inspections are more effective when documents are readable.

Software Documents Are Meant to Be Read by People

• Documents should have a simple structure, and not be verbose.
• Comments should add to understandability & not restate the obvious.
• Avoid overly complex structures without very strong justification. Document these complex solutions.

Software That Can Be Verified

• Is simply structured.
• Has a written, valid requirements specification.
• Evolved to its current form following a well-defined development process.
Summary

- V & V involves making sure that:
  - We built the right software (validation).
  - We built the software right (verification).
- Perfect testing is impossible.
- Testing has many facets:
  - What we test: from system testing to unit testing.
  - When we test: from alpha testing to regression testing.

Summary

- Black box testing involves developing test cases in terms of the specification.
- White box (structural) testing involves using test cases to cover all parts of the program.
- Rigorous testing requires a comprehensive test plan.
  We saw a detailed example of a test plan for conducting black-box class testing, and an associated test driver.
- Software inspections can find faults early.