9-Software Testing, Verification and Validation

Yan Shi

SE 2730 Lecture Notes
Verification and Validation

- **Verification:**
  - “Are we building the product right?”
  - Check whether the software conforms to its specification.

- **Validation:**
  - “Are we building the right product?”
  - Ensure the system meets the customer’s expectation.
  - More general process.
Two V&V approaches

- Software inspections and reviews:
  - Static approach
  - Can't "execute"!
  - Create group, have them review the documents
  - Goal: identify problems

- Software testing:
  - Dynamic approach.
  - Run an implementation of the software with test data.
Cost of Testing

Half of MS engineers are testers, programmers spend \textit{HALF} their time testing
– Bill Gates

You’re going to spend at least \textit{HALF} of your development budget on testing, whether you want to or not

In the real-world, testing is the \textit{principle post-design} activity
Cost of **NOT** Testing

- Testing is too expensive.
- Software test tools cost around $10,000!
- OK. I will start planning for testing after development.
- Not testing is even more expensive.
- A test station for circuit boards costs half a million dollars.
- It will be PROHIBITIVELY expensive. You should start testing as soon as possible.
Fault, Failure and Error

From standards by IEEE Computer Society

- **Error** – human interaction which produces an incorrect result.
- **Fault** – representation of an error.
- **Failure** – occurs when a fault executes.

We will use fault, defect and bug interchangeably.
Fault & Failure Model

- Three conditions necessary for a failure to be observed
  - **Reachability**: The location or locations in the program that contain the fault must be reached
  - **Infection**: The state of the program must be incorrect
  - **Propagation**: The infected state must propagate to cause some output of the program to be incorrect
Testing & Debugging

- **Testing**: The act of exercising software with test cases to
  - Find failures (detection testing), OR
  - Demonstrate correct execution (validation testing).

  ➔ *These are the two major goals of testing!*

- **Debugging**: The process of finding a fault given a failure
Some Common Sense

- Cost of fixing a defect grows exponentially.
  - One main purpose of testing is to find bugs before it is too late!

- People will find what they are looking for: if they are looking to show something works, they will!
  - We are not the right people to be proofing our own work.
<table>
<thead>
<tr>
<th>Test Cast ID</th>
<th>Purpose</th>
<th>Pre-conditions</th>
<th>Inputs</th>
<th>Expected Outputs</th>
<th>Post-conditions</th>
<th>Execution History</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Date</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Quiz: The Triangle Problem

- The triangle program accepts 3 integers: a, b and c, which are taken to be sides of a triangle. The maximum length of a side is 1000.

- The output of the program is the type of triangle determined by the three sides: Equilateral, Isosceles, Scalene, or NotATriangle.

How many test cases can you think of to test this program?
Suggested Tests of Triangle Program

- Verify that invalid inputs are rejected as expected. 30 test cases

a: not numeric, negative, too many decimal places, common keyboard entry error, etc. Same with b and c.
Suggested Tests of Triangle Program

- Verify that invalid inputs are rejected as expected. 30 test cases
- Verify that only and exactly three numbers can be entered successfully. 2 test cases

More than 3 inputs or less than 3 inputs
Suggested Tests of Triangle Program

- Verify that invalid inputs are rejected as expected. 30 test cases
- Verify that only and exactly three numbers can be entered successfully. 2 test cases
- Verify that the triangle must close. 6 test cases

*a + b < c, a + b = c; same with order “b,c,a” and “c,a,b”*
Suggested Tests of Triangle Program

- Verify that invalid inputs are rejected as expected. **30 test cases**
- Verify that only and exactly three numbers can be entered successfully. **2 test cases**
- Verify that the triangle must close. **6 test cases**
- Verify that equilateral triangles are processed correctly. **1 test case**
- Verify that isosceles triangles are processed correctly. **3 test cases**
- Verify that scalene triangles are processed correctly. **1 test case**
- Verify that extreme lengths are processed correctly. **6 test cases**

Fore each side: 1 test on the boundary and 1 test right outside the boundary
Suggested Tests of Triangle Program

- Verify that invalid inputs are rejected as expected. 30 test cases
- Verify that only and exactly three numbers can be entered successfully. 2 test cases
- Verify that the triangle must close. 6 test cases
- Verify that equilateral triangles are processed correctly. 1 test case
- Verify that isosceles triangles are processed correctly. 3 test cases
- Verify that scalene triangles are processed correctly. 1 test case
- Verify that extreme lengths are processed correctly. 7 test cases
- Verify that the result is displayed in an acceptable manner.

~50 test cases should give us confidence about the triangle program.
What Does the Triangle Example Tell Us?

- Design test cases according to requirements.
  - There are always ambiguity in requirements.
  - This means you should refine the requirements.
- No duplicated test cases for the same purpose.
- The purpose of testing a program:
  - Check if it does what are required;
  - Check if it does not do what it should not do;
  - Check if it can handle exception;
Activities of Test Engineers

- Test Designs
- Executable Tests
- Computer
- Outputs

Test Engineer designs Test Designs, which are instantiated to create Executable Tests. The Executable Tests are executed on the Computer to produce Outputs. The Test Engineer evaluates the Outputs.
Activities of Test Engineers

Test Design
- Test Engineer
  - design
  - Test Design

Test Automation
- Test Engineer
  - generate
  - Executable Tests

Executable Tests
- instantiate
- Test Engineer
- execute
- Outputs

Outputs
- evaluate
- Test Engineer

Test Execution
- Computer
- Test Execution
Test Case Design Techniques

- Functional Testing: black-box
  - Partition Testing
  - Boundary Testing
  - ...

- Structural Testing: white-box
  - Statement coverage
  - Path coverage
  - Control-flow coverage
  - Data-flow coverage
Functional Testing

- The program can be viewed as a function that maps values from input domain to values in its outputs.

- A.K.A. black-box testing, requirement based testing

- Required Information: only requirement specification

- Independent of the implementation

- Test design can be in parallel with implementation
The triangle program accepts 3 integers: a, b and c, which are taken to be sides of a triangle.

The output of the program is the type of triangle determined by the three sides: Equilateral, Isosceles, Scalene, or NotATriangle.

- \(<1,3,4>, \text{Scalene}\>
- \(<3,3,3>, \text{Equilateral}\>
- \(<2,2,3>, \text{Isosceles}\>
- \(<1,3>, \text{Error: missing side}\>
- \(<1.2, 3.5, 2,1>, \text{Error: integer input only}\>
- ...
Partition Testing

- If we should accept voltages in the range of 3.5 to 6.0, do we need to test for 3.7, 3.8, 4.1, 5.6 and 5.9?

- Partition testing:
  - Partition same behaviors into one equivalence class.
  - ONLY one test case is generated from one equivalence class.
Boundary testing consists of testing on the edge (and either side of the edge) of legal input values.

- If we should accept voltages in the range of 3.5 to 6.0 volts, inclusive, one edge would be 3.5 volts.
- We would test 3.49 volts – it should be rejected, another value is 3.50 volts – it should pass.
- Same tests should be designed for 6.0 edge.

These boundaries should be available in use cases and HCI design documents.
Structural Testing

- The implementation is known and structure of the program is used to design test cases.
- A.K.A. white-box testing (clear-box testing)
- Required information: coverage
- More theoretical: graph theory, coverage metrics

**Benefits:**
- Explicitly state the extent to which the software is tested
- Makes testing management more meaningful
Triangle Example

- statement coverage

```cpp
int main()
{
    int a, b, c;
cin >> a >> b >> c;
if (a + b > c && b + c > a && c + a > b)
{
    if (a==b)
        if (b==c)
            cout << "Equilateral" << endl;
        else
            cout << "Isosceles" << endl;
    else if (b==c)
        if (c==a)
            cout << "Equilateral" << endl;
        else
            cout << "Isosceles" << endl;
    else if (a==c)
        if (c==b)
            cout << "Equilateral" << endl;
        else
            cout << "Isosceles" << endl;
    else
        cout << "Scalene" << endl;
}
else
    cout << "NotATriangle" << endl;
return 0;
}
```
Triangle Example

- statement coverage
  - <<1,3,4>, NotATriangle>
  - <<3,3,3>, Equilateral>
  - <<2,2,3>, Isosceles>
  - <<3,4,5>, Scalene>
  - <<4,2,4>, Isosceles>
  - <<2,3,3>, Isosceles>

```cpp
int main()
{
    int a, b, c;
    cin >> a >> b >> c;
    if (a + b > c && b + c > a && c + a > b)
    {
        if (a==b)
            if (b==c)
                cout << "Equilateral" << endl;
            else
                cout << "Isosceles" << endl;
        else if (b==c)
            if (c==a)
                cout << "Equilateral" << endl;
            else
                cout << "Isosceles" << endl;
        else if (a==c)
            if (c==b)
                cout << "Equilateral" << endl;
            else
                cout << "Isosceles" << endl;
        else
            cout << "Scalene" << endl;
    }
    else
        cout << "NotATriangle" << endl;
    return 0;
}
```
Functional vs. Structural

Neither approach alone is sufficient.
They should work together.

If not all specified behaviors are implemented, structural test will NEVER recognize this.

If a program implements what is not specified, functional test will NEVER reveal it. For example, virus!
V model: Testing Levels Based on Software Activity

Test Design Information

- Requirement Analysis
- Architectural Design
- Subsystem Design
- Detailed Design
- Implementation
- Unit Test
- Module Test
- Integration Test
- System Test
- Acceptance Test
Unit Testing

- As each unit (could be a function or class or module) is implemented, it is desirable to test them before more code gets added or the module is integrated with others.

- Generally, this is done as both white box testing (based on knowledge of the code) and black box testing.

- Unit tests are often parts of detailed design documents and may be a responsibility of the development team.

- JUnit is an example of Unit Testing framework.
Sometime **test harnesses or simulators** are necessary to perform unit testing,

- **Test Harness** – code is written to simulate other parts of a system that are not yet functional or to stress code harder than can be could normally be done in a testing lab.

- **Test Stubs** – minimal function implementations to check the logic of calling functions and interfaces between calling and called functions.

When **proof of passing a test** is available, the unit software can be released for integrating into Subsystems.
Module/Subsystem Testing

- Rather than dealing with individual classes we are dealing with groups of classes and often a complete executable. It is smaller than the entire system.

- Often must treat a subsystem as a Black Box.

- Encapsulating code limits what program elements we have access to for testing.
  - We may require Test Harnesses to drive an entire subsystem when other subsystems are not available yet.
Integration Testing

- The **incremental** testing of a system as subsystems are sequentially added.
- It is more than module/subsystem testing and less than system level testing.
- Integration testing
  - determines how well subsystems play together.
  - identify subsystem interface problems.
System Testing

- **System Level** testing involves **testing only after the system is completed**.

- System tests are done at the user’s level. Tests should cover real-world user scenarios.

- **Tests should cover non-specified events**.
  - If a user might **possibly hit a keyboard with a fly swatter**, then we should see what happens.
  - The system should handle unexpected events.
  - Often **defects found at the system level are harder to fix**. This is because we dealing with a large body of code rather than a smaller body and it may be harder to find the specific cause when millions of lines of code may be involved.
Acceptance Testing

- **Let the customer test it!**

- **Alpha and Beta Testing:**
  - **Alpha testing**: all of the major features, but perhaps not all of the minor features are working.
    - At least one system level testing pass has been made.
    - There may be known critical defects in the system.
    - The code is sent to a very limited number of “friendly” users who test it under real-world conditions.
  - **Beta testing**: all major and minor features have been implemented and few if any major defects (but possibly some minor defects) exist.

- **Major benefit** of Alpha and Beta testing is that your system will be tested by many more people than you could test in house and in many different real-world ways that you cannot simulate in your test lab.
Review the V model

Test Design Information

- Requirement Analysis
- Architectural Design
- Subsystem Design
- Detailed Design
- Implementation
- Unit Test
- Module Test
- Integration Test
- System Test
- Acceptance Test
Regression Testing

- It is possible to introduce new bug when fixing a bug.
- Use regression testing to ensure it doesn’t happen.
  - Rerun previously run tests and check whether program behavior has changed and whether previously fixed faults have re-emerged.
Regression Testing (2)

- Regression testing is often *automated*.  
  - At some places there is a nightly build, install, regression test, and defect reports sent to team leaders each morning.
  - This strategy makes fixing defects easy: if it worked one day but didn’t the next you can easily isolate what has changed.

- Regression test suites are *growing* as new ways to break a system are discovered.
  - In verifying that you have fixed a defect, you generate a test. When done, add this test to your regression suite.
Test Plan and Management

- **Test Results Need to be Recorded**
  - Legal evidence that due diligence has been served.
  - **Some regulated industries** (FAA, FDA, DoE, DoD, etc.) contractually require testing data to be delivered with the product.
  - Individual test engineers and their managers can be held professionally liable for falsification.
Test Plan and Management (2)

- Make sure there is time scheduled for developing and executing tests.

- Start testing as early as possible
  - Finding bugs at the unit level makes it easier to correct the bugs (less code to search).
  - Testing early can detect misunderstandings in requirements and specifications with time to fix them.

- Make sure there is time to fix defects that are found and rerun tests.
Traceability between test cases and requirements is very important!

<table>
<thead>
<tr>
<th>Test Case\Requirement</th>
<th>F01</th>
<th>F02</th>
<th>NF01</th>
</tr>
</thead>
<tbody>
<tr>
<td>T001</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T002</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>T003</td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Make sure all requirements are sufficiently tested.
Summary

- Two V&V approaches: inspection and testing.
- Error, Fault and Failure
- Four types of testing activities
- Different test case design techniques
- Different levels of testing
- Regression testing
- Test plan and management
Four Types of Test Activities

- Test Design
  - Criteria-based
  - Human-based
- Test Automation
- Test Execution
- Test Evaluation

Each type of activity requires different skills, background knowledge, education and training.
Test Design – (a) Criteria-Based

- Design test values to satisfy coverage criteria or other engineering goal.

- This is the most technical job in software testing
- Requires knowledge of:
  - Discrete math
  - Programming
  - Testing
- Requires much of a traditional CS or SE degree
- Test design is analogous to software architecture on the development side
Test Design – (b) Human-Based

- Design test values based on domain knowledge of the program and human knowledge of testing.

- This is much harder than it may seem to developers
- Criteria-based approaches can be blind to special situations
- Requires knowledge of:
  - Domain, testing, and user interfaces
- Requires almost no traditional CS or SE
  - A background in the domain of the software is essential
  - An empirical background is very helpful (biology, psychology, ...)
  - A logic background is very helpful (law, philosophy, math, ...)
Test Automation

- Embed test values into executable scripts

- This is slightly less technical
- Requires knowledge of **programming**
  - Fairly straightforward programming – small pieces and simple algorithms
- Requires very little theory
- Very **boring** for test designers
- Programming is out of reach for many **domain experts**
Test Execution

- Run tests on the software and record the results.

- This is easy – and trivial if the tests are well automated.
- If tests are not well automated, this requires a lot of manual labor.
- Requires basic computer skills
  - Interns
  - Employees with no technical background.
- Asking qualified test designers to execute tests is a sure way to convince them to look for a development job.
Test Evaluation

- Evaluate results of testing, report to developers

- This is much **harder** than it may seem.
- Same requirement as Human-based Test Design.