Software Testing Level
Part 1

Adam Hendra Brata
Agenda : Part 1

- Introduction
- Software Testing Level
  - Unit Testing
  - Integration Testing
  - System Testing
  - Validation Testing
Introduction
Introduction

To enhance the quality of software testing, and to produce a more unified testing methodology applicable across several projects, the testing process could be abstracted to different levels.

This level directly corresponding to Software Testing Life Cycle and Software Development Life Cycle.
Software Testing Level
Testing Activities

- Object Design Document
- System Design Document
- Requirements Analysis Document
- Client Expectation

- Unit Testing
- Integration Testing
- System Testing
- Acceptance Testing

Developer
Client
Types of Testing

- **Unit Testing**
  - Individual component (class or subsystem)
  - Carried out by developers
  - **Goal:** Confirm that the component or subsystem is correctly coded and carries out the intended functionality

- **Integration Testing**
  - Groups of subsystems (collection of subsystems) and eventually the entire system
  - Carried out by developers
  - **Goal:** Test the interfaces among the subsystems.
Types of Testing continued...

- **System Testing**
  - The entire system
  - Carried out by developers
  - **Goal:** Determine if the system meets the requirements (functional and nonfunctional)

- **Acceptance Testing**
  - Evaluates the system delivered by developers
  - Carried out by the client. May involve executing typical transactions on site on a trial basis
  - **Goal:** Demonstrate that the system meets the requirements and is ready to use.
When should you write a test?

- Traditionally after the source code is written
- In XP before the source code written “completely”
  - Test-Driven Development Cycle
    - Add a test
    - Run the automated tests
      => see the new one fail
    - Write some code
    - Run the automated tests
      => see them succeed
    - Refactor code.
Unit Testing
Unit Testing

- **Static Testing (at compile time)**
  - Static Analysis
  - Review
    - Walk-through (informal)
    - Code inspection (formal)

- **Dynamic Testing (at run time)**
  - Black-box testing
  - White-box testing.
Static Analysis with IDE

- **Compiler Warnings and Errors**
  - Possibly uninitialized Variable
  - Undocumented empty block
  - Assignment has no effect

- **Checkstyle**
  - Check for code guideline violations
  - [http://checkstyle.sourceforge.net](http://checkstyle.sourceforge.net)

- **FindBugs**
  - Check for code anomalies
  - [http://findbugs.sourceforge.net](http://findbugs.sourceforge.net)

- **Metrics**
  - Check for structural anomalies
  - [http://metrics.sourceforge.net](http://metrics.sourceforge.net)
Black-box testing

- **Focus:** I/O behavior
  - If for any given input, we can predict the output, then the component passes the test
  - Requires test oracle

- **Goal:** Reduce number of test cases by equivalence partitioning:
  - Divide input conditions into equivalence classes
  - Choose test cases for each equivalence class.
Black-box testing: Test case selection

- Input is valid across range of values
  - Developer selects test cases from 3 equivalence classes:
    - Below the range
    - Within the range
    - Above the range

- Input is only valid, if it is a member of a discrete set
  - Developer selects test cases from 2 equivalence classes:
    - Valid discrete values
    - Invalid discrete values

- No rules, only guidelines.
Black box testing: An example

public class MyCalendar {
    public int getNumDaysInMonth(int month, int year) throws InvalidMonthException {
        ... }
}

Representation for month:
1: January, 2: February, ...., 12: December

Representation for year:
1904, ... 1999, 2000, ..., 2006, ...

How many test cases do we need for the black box testing of getNumDaysInMonth()?
White-box testing overview

- **Function coverage**
  - Has each function in the program been executed?

- **Entry/exit coverage**
  - Has every possible call and return of the function been executed?

- **Statement coverage**
  - Has each line of the source code been executed?

- **Condition coverage**
  - Has each evaluation point (such as a true/false decision) been executed?

- **Path coverage**
  - Has every possible route through a given part of the code been executed?
<table>
<thead>
<tr>
<th></th>
<th>Unit Testing Heuristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Create unit tests when object design is completed</td>
</tr>
<tr>
<td></td>
<td>- Black-box test: Test the functional model</td>
</tr>
<tr>
<td></td>
<td>- White-box test: Test the dynamic model</td>
</tr>
<tr>
<td>2.</td>
<td>Develop the test cases</td>
</tr>
<tr>
<td></td>
<td>- Goal: Find effective number of test cases</td>
</tr>
<tr>
<td>3.</td>
<td>Cross-check the test cases to eliminate duplicates</td>
</tr>
<tr>
<td></td>
<td>- Don't waste your time!</td>
</tr>
<tr>
<td>4.</td>
<td>Desk check your source code</td>
</tr>
<tr>
<td></td>
<td>- Sometimes reduces testing time</td>
</tr>
<tr>
<td>5.</td>
<td>Create a test harness</td>
</tr>
<tr>
<td></td>
<td>- Test drivers and test stubs are needed for integration testing</td>
</tr>
<tr>
<td>6.</td>
<td>Describe the test oracle</td>
</tr>
<tr>
<td></td>
<td>- Often the result of the first successfully executed test</td>
</tr>
<tr>
<td>7.</td>
<td>Execute the test cases</td>
</tr>
<tr>
<td></td>
<td>- Re-execute test whenever a change is made (“regression testing”)</td>
</tr>
<tr>
<td>8.</td>
<td>Compare the results of the test with the test oracle</td>
</tr>
<tr>
<td></td>
<td>- Automate this if possible.</td>
</tr>
</tbody>
</table>
Integration Testing
Integration Testing

- Integration testing is the phase of the overall testing process in which individual software modules are combined and tested to evaluate interactions between them.
- In Integration testing, individual software modules are integrated logically and tested as a group.
- A typical software project consists of multiple software modules, coded by different programmers.
- Integration testing focuses on checking data communication amongst these modules.
Software Integration Strategy

● What is a software integration strategy?
  - Software test strategy provides the basic strategy and guidelines to test engineers to perform software testing activities in a rational way.
  - Software integration strategy usually refers to an integration sequence (or order) to integrate different parts (or components) together.

● A test model is needed to support the definition of software integration test strategies.
  - control flow graph
  - object-oriented class diagram
  - scenario-based model
  - component-based integration model
  - architecture-based integration model
Software Integration Strategy

- There are two groups of software integration strategies:
  - Non Incremental software integration
    - Big Bang
  - Incremental software integration
    - Top-down software integration
    - Bottom-up software integration
    - Sandwich integration
Software test **stubs** are programs which simulate the behaviors of software components (or modules) that are the *dependent* modules of a *under test module*.

Typical stubs relates to a *under test module* in the following ways:

- Functional call without parameters and return values
- Functional call with parameters and return values
- Functional call with input parameters only
Software test **drivers** are programs which simulate the behaviors of software components (or modules) that are the control modules of a under test module.

Typical drivers relates to a under test module in the following ways:
Driver & Stub

- A test **driver** calls the software under test, passing the test data as inputs
- A **stub** is temporary or dummy software that is required by the software under test for it to operate properly.
Driver & Stub

- **Driver**: Calls with test input values to the Software under Test.
- **Software under Test**: Calls to stub software.
- **Stub**: This accepts input values, checks they are in the testset \{0,100\}, and then returns the hard-coded Fahrenheit equivalent values.
- **Test Driver**: This emulates user input by entering the test values to the correct test-input boxes in the user-interface, and checks the result.
- **Software Under Test**: The GUI displays the input form, and accepts input data. It then calls the stub to perform the conversion, and displays the result.
Non Incremental : Big Bang Approach

Big Bang Integration Testing

Module 1

Module 2

Module 3

Module 4

Module 5

Module 6

Combine (or integrate) all parts at once
Big Bang Approach

- Combine (or integrate) all parts at once.

**Advantages:**
- Convenient for small systems.

**Disadvantages:**
- Fault localization is difficult.
- Since all modules are tested at once, high risk critical modules are not isolated and tested on priority.
- Hard to debugging, not easy to isolate errors.
- Not easy to validate test results.
Incremental

Test sequence 1

Test sequence 2

Test sequence 3
Top Down Approach

It requires the highest-level modules be test and integrated first. This allows high-level logic and data flow to be tested early in the process and it tends to minimize the need for drivers.
Top Down Approach

- Modules are integrated by moving downward through the control structure.
- Modules subordinate to the main control module are incorporated into the system and represented into tree structure.

Integration process:
1. The main control module is used as a test driver, and the stubs are substituted for all modules directly subordinate to the main control module.
2. Subordinate stubs are replaced one at a time with actual modules.
3. Tests are conducted as each module is integrated.
4. On completion of each set of tests, another stub is replaced with the real module.
5. Regression testing may be conducted.
Top Down Approach Example

Select the integration sequence:
- Depth-First
- Breadth-First

Select the direction:
- Left → Right
- Right → Left
Top Down Approach Example

- **Integration Order**: Breadth-First (Left Order)
  - IS : Integrated System
  - M[i] ’ : software stub for Module M[i]

- **Steps**
  - Step #1: IS = Main + M1 (need: M2’, M3’, M4’ and M5’)
  - Step #2: IS = IS + M2 (need: M4’, M5’, M6’, and M3’)
  - Step #3: IS = IS + M3 (need: M4’, M5’, M6’, and M7’)
  - Step #4: IS = IS + M4 (need: M5’, M6’, and M7’)
  - Step #5: IS = IS + M5 (need: M8’, M6’, and M7’)
  - Step #6: IS = IS + M6 (need: M7’, and M8’)
  - Step #7: IS = IS + M7 (need: M8’)
  - Step #8: IS = IS + M8
Top Down Approach

- **Advantages**
  - The top layer provides an early outline of the overall program helping to find design errors early on and giving confidence to the team, and possibly the customer, that the design strategy is correct.

- **Disadvantages**
  - Difficulty with designing stubs that provides a good emulation of the interactivity between different levels
  - If the lower levels are still being created while the upper level is regarded as complete, then sensible changes that could be made to the upper levels of the program that would improve its functioning may be ignored
  - When the lower layers are finally added the upper layers may need to be retested
Bottom Up Approach

The bottom-up approach requires the lowest-level units be tested and integrated first.
Bottom Up Approach

- Modules at the lowest levels are integrated at first, then by moving upward through the control structure.
- These modules are frequently referred to as utility modules. By using this approach, utility modules are tested early in the development process and the need for stubs is minimized.
- Integration process
  1. Low-level modules are combined into clusters that perform a specific software sub-function.
  2. A driver is written to coordinate test case input and output.
  3. Test cluster is tested.
  4. Drivers are removed and clusters are combined moving upward in the program structure.
Bottom Up Approach Example
Bottom Up Approach Example

- Integration Order: Breadth-First (Left Order)
  - IS: Integrated System
  - M[i]': software stub for Module M[i]

- Steps
  Step #1: \( IS_1 = M_8 + M_4 \) (need: M_5" and M_1")
  Step #2: \( IS_1 = IS_1 + M_5 \) (need: M_1")
  Step #3+4: \( IS_1 = IS_1 + M_1 \) (need: Main")
  Step #5: \( IS_2 = M_2 + M_6 \) (need: Main")
  Step #6: \( IS_3 = M_3 + M_7 \) (need: Main")
  Step #7: \( IS = IS_1 + \text{Main} \)
  Step #7: \( IS = IS + IS_2 \)
  Step #8: \( IS = IS + IS_3 \)
Top Down Approach

● Advantages
  - Overcome the disadvantages of Top-Down Testing. Additionally, drivers are easier to produce than stubs and because the tester is working upwards from the bottom layer, they have a more thorough understanding of the functioning of the lower layer modules and thus have a far better idea of how to create suitable tests for the upper layer modules.

● Disadvantages
  - It is more difficult to imagine the working system until the upper layer modules are complete
  - The important user interaction modules are only tested at the end
  - Drivers, often many different ones with varying levels of sophistication, must be produced
Sandwich Approach

- It is a combination of both Top-down and Bottom-up integration testing.
- A target layer is defined in the middle of the program and testing is carried out from the top and bottom layers to converge at this target layer.
- It has the advantages that the top and bottom layers can be tested in parallel and can lower the need for stubs and drivers.
- However, it can be more complex to plan and selecting the “best” target layer can be difficult.
Integration Testing Steps

1. Based on the integration strategy, select a component to be tested. Unit test all the classes in the component.
2. Put selected component together; do any preliminary fix-up necessary to make the integration test operational (drivers, stubs)
3. Test functional requirements: Define test cases that exercise all use cases with the selected component
4. Test subsystem decomposition: Define test cases that exercise all dependencies
5. Test non-functional requirements: Execute performance tests
6. Keep records of the test cases and testing activities.
7. Repeat steps 1 to 7 until the full system is tested.

The primary goal of integration testing is to identify failures with the (current) component configuration.
Integration Testing Simple Example

//function1
public int getMaxInTwo(int a, int b)
{
    if (a >= b)
        return a;
    else
        return b;
}

//function2
public int getMaxInThree(int a, int b, int c)
{
    a = a + 1;
    int max = getMaxInTwo(a, b);
    max = getMaxInTwo(max, c);
}

Unit Testing test cases?
Integration Testing test cases?
Integration Testing Simple Example

- **Test Requirements**
  - **Unit Testing**
    - In the function 1, if input a is greater than b, return output a, otherwise return b
    - In function 2, if a is maximum among inputs, return a
  - **Integration Testing**
    - In function 2, if input a, b & c, it should call the function 1, check if the return each result is correct and if the final result is correct

**Sample Test Case Table:**

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Test Case Description</th>
<th>Input Data</th>
<th>Expected Result</th>
<th>Actual Result</th>
<th>Pass/Fail</th>
<th>Remarks</th>
</tr>
</thead>
</table>
Integration Testing Simple Example

- **Unit Testing Coverage**
  - If during this execution function1 was called at least once, then function coverage for this function is satisfied.
  - Entry/Exit coverage will be satisfied if the function can receive some various inputs and have an output as result.
  - Statement coverage for this function will be satisfied if it was called with various input, e.g. function1(3,2), function1(1,3) and function1(2,2).
  - Tests calling function1(3,2), function1(1,3) and function1(2,2) will satisfy condition coverage.
Integration Testing Simple Example

- Driver and Stub

```java
int test1()
{
    int a=m; // m,n are test case inputs
    int b=n;
    Int max = getmaxintwo(a,b); // driver
    System.out.println(max); // stub
}

int test2()
{
    int a1=x; // x,y,z are test case inputs
    int b1=y;
    Int c1=z;
    Int max = getmaxinthree(a1,b1,c1); // driver
    System.out.println(max); // stub
}
```

Simulate these functions as a part of big program
In integration testing, test cases are developed with the express purpose of exercising the interface between the components.

- function 2 will pass 2 parameters to function 1, the conditions in function 1 are:
  1. $a > b$
  2. $a < b$
  3. $a = b$

How to design the test cases covering all 3 conditions?

$(1, 2, 3)$ can cover 1 and 2

We need another one to cover 3

$(2, 2, 2)$
System Testing
Validation & Verification

- Software Verification: The process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.
- Software Validation: The process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements.
Validation & Verification

- Validation and Verification
  - In other words, software verification is ensuring that the product has been built according to the requirements and design specifications, while software validation ensures that the product meets the user's needs, and that the specifications were correct in the first place. Software verification ensures that "you built it right". Software validation ensures that "you built the right thing". Software validation confirms that the product, as provided, will fulfill its intended use.
  - **Verification** is testing that your product meets the specifications / requirements you have written. “Did I build what I said I would?”.  
  - **Validation** tests how well you addressed the business needs that caused you to write those requirements. It is also sometimes called acceptance or business testing. “Did I build what I need?”
V Model
V&V Toolbox

Testing = Verification + Validation
System Testing

- System Testing
  - The process of testing of an integrated hardware and software system to verify that the system meets its specified requirements. (IEEE)
  - It is performed when integration testing is completed.
  - This testing evaluates working of system from user point of view, with the help of specification document.
  - It contains functional and non-functional areas of application/product.
System Testing

Complete ver. : check Software Quality Factors slide
System Testing

- Installation Testing
  - To make sure that product / software can be installed on specific or support defined system, can be configured and can be brought into an operational mode.

- Functionality Testing
  - To make sure that functionality of product are working as per the requirements defined, within the capabilities of the system,

- Recoverability Testing
  - To make sure how well the system recovers from various input errors and other failure situations.

- Interoperability Testing:
  - To make sure whether the system can operate well with third party products or not.
System Testing

- **Performance Testing**
  - To make sure system’s performance under various conditions, in terms of performance characteristics.

- **Scalability Testing**
  - To make sure system’s scaling abilities in various terms like user scaling, geographic scaling, and resource scaling.

- **Reliability Testing**
  - To make sure system can be operated for longer duration without developing failures.

- **Regression Testing**
  - To make sure system’s stability as it passes through integration of different sub systems and maintenance tasks.
System Testing

- **Documentation Testing**
  - To make sure that system’s user guide and other help topics documents are correct and usable.

- **Security Testing**
  - To make sure that system does not allow unauthorized access to data and resources.

- **Usability Testing**
  - To make sure that system is easy to use, learn and operate.
System Testing Tips

- Tip #1: Replicate real time scenarios rather than doing ideal testing as the system is going to be used by an end user and not by the trained tester.
- Tip #2: Verify system’s response in various terms as human does not like to wait or to see wrong data.
- Tip #3: Install and configure system as per the documentation because that is what end user is going to do.
- Tip #4: Involving people from different areas like business analysts, developers, testers, customers can end in a better system.
- Tip #5: Regular testing is the only way to make sure that littlest change in code to fix the bug has not inserted another critical bug into the system.
System Integration Testing

System 1 + System 2 + System n = “Combined & Big” System

Example: Enterprise-level Application
Validation Testing
Validation Testing

- The process of evaluating software during the development process or at the end of the development process to determine whether it satisfies specified business requirements.

- Validation Testing ensures that the product actually meets the client's needs. It can also be defined as to demonstrate that the product fulfills its intended use when deployed on appropriate environment.

- In fact, validation testing is all of the testing stages itself
Validation Testing
Thank You