CS265 Software testing
-- Integration and System Testing

Fangli Ying
Email: fying@cs.nuim.ie
Test phases

Unit testing on individual units of source code (=smallest testable part).

Integration testing on groups of individual software modules.

System testing on a complete, integrated system (evaluate compliance with requirements)
**Integration testing** is the phase of the overall testing process in which individual software modules are combined and tested to evaluate interactions between them.

It occurs after unit testing and before validation testing. Input modules that have been unit tested.

Integrated aggregate of more than one unit into even larger parts of program

Delivering its output (the integrated system) ready for system testing.
Drivers & Stubs

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.
Common strategies of integration testing

1. Big bang approach
2. Top-down approach
3. Bottom-up approach
4. Sandwich approach
Big bang (non-incremental) testing
& incremental testing

Big Bang Integration Testing

Module 1
Module 2
System
Module 3
Module 4
Module 5
Module 6

Test sequence 1
Test sequence 2
Test sequence 3
It requires the highest-level modules be tested 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 Integration:

**Advantages:**
The strengths of using the Top-Down strategy is that 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:

1. difficulty with designing stubs that provides a good emulsion of the interactivity between different levels

2. 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

3. When the lower layers are finally added the upper layers may need to be retested
• In depth first approach all modules on a control path are integrated first. See fig. 9.6. Here sequence of integration would be
  • (M1, M2, M3), M4, M5, M6, M7, and M8.

• In breadth first all modules directly subordinate at each level are integrated together.
  • Using breadth first for fig. 9.6 the sequence of integration would be
  • (M1, M2, M8), (M3, M6), M4, M7, and M5.
The bottom-up approach requires the lowest-level units to be tested and integrated first. These units 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.
Bottom-up Integration:

Advantages:
The strengths of Bottom-Up Testing 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:

1. It is more difficult to imagine the working system until the upper layer modules are complete.

2. The important user interaction modules are only tested at the end.

3. Drivers, often many different ones with varying levels of sophistication, must be produced.
Bottom-up Integration example
Top-Down Integration vs. Bottom-Up Integration

• Top-down integration has the "advantage" (over bottom-up integration) that it starts with the main module, and continues by including top level modules - so that a "global view" of the system is available early on.

• It has the disadvantage that it leaves "utility modules" until the end, so that problems with these won't be detected early. If "bottom-up" integration is used then these might be found soon after integration testing begins.

• Top-down integration also has the disadvantage or requiring stubs - which can sometimes be more difficult to write than drivers, since they must simulate computation of outputs, instead of their validation.
Sandwich Integration

• 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.
How do I get started?
Integration Testing Steps:

1. Identify Unit Interfaces
2. Reconcile Interfaces for Completeness
3. Create Integration Test Conditions
4. Evaluate the Completeness of Integration Test Conditions
Example:

//function 1
Public int getMaxInTwo (int a, int b)
{
    if (a>=b) return a;
    else return b;
}

//function 2
Public int getMaxInThree (int a, int b, int c)
{
    a=a+1;
    int max=getMaxInTwo(a, b);
    max=getMaxInTwo(max, c);
}

Unit test cases?
Integration test cases?
Test requirements

**UT :**

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...

**IT :**

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
UT Test Cases:

getMaxInTwo UT
(3, 2)
(1, 3)
(2, 2)
coverage 100%;

getMaxInThree UT
(1, 2, 3)
coverage 100%

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>

Perform UT for function1 & function2, the coverage is 100%

if only UT for getMaxInThree, the coverage of getMaxInTwo is 50%. Why?
Driver & stubs for integration testing

```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
}
```
Integration testing: interface analysis

# In integration testing, test cases are developed with the express purpose of exercising the interface between the components

function2 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)
Additional exercises:

// input x=1; y=2;
Public void A(int x, int y,)
{
    B(x,y);
    System.out.println("x=");
    System.out.println("+x+" y= +y);
}

Public void B(int a, int b,)
{
    b=a+b;
}

// input x=1; y=2; or x=2, y=0
Public void A(int x, int y,)
{
    Int result = B(x,y);
    System.out.println(result);
}

Public int B(int a, int b,)
{
    Int c = (int)a/(int)b
    Return c;
}
Learn it, Use it!
Additional contents:
Testing with JUnit

JUnit
Tests in JUnit

Tests are realized as `public void testX()` methods.

A test typically calls a few methods, then checks if the state matches the expectation. If not, it fails.

*Example:* Call `empty()` on the shopping cart and check the state with `isEmpty()`

```java
// Tests the emptying of the cart.
public void testEmpty() {
    _bookCart.empty();
    assertTrue(_bookCart.isEmpty());
}
```
JUnit Checklist

☐ Initialize
☐ Write test cases
☐ Write a test suite
☐ Execute test suite
Example: Shopping cart

1. Set of products
2. Number of products
3. Balance

1. Add product
2. Remove product
Part 1: Initialize

Constructor + Set up and tear down of fixture.

```java
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class ShoppingCartTest extends TestCase {

    private ShoppingCart _bookCart;

    // Creates a new test case
    public ShoppingCartTest(String name) {
        super(name);
    }
}
```
Test fixture

// Creates test environment (fixture).
// Called before every testX() method.
protected void setUp() {
    _bookCart = new ShoppingCart();

    Product book = new Product("Harry Potter", 23.95);
    _bookCart.addItem(book);
}

// Releases test environment (fixture).
// Called after every testX() method.
protected void tearDown() {
    _bookCart = null;
}
Part 2: Write test cases

Help methods inherited from TestCase:

fail(msg) – triggers a failure named msg
assertTrue(msg, b) – triggers a failure, when condition b is false
assertEquals(msg, v1, v2) – triggers a failure, when v1 ≠ v2
assertEquals(msg, v1, v2, ε) – triggers a failure, when |v1 − v2| > ε
assertNull(msg, object) – triggers a failure, when object is not null
assertNonNull(msg, object) – triggers a failure, when object is null

The parameter msg is optional.
Test: Add product

// Tests adding a product to the cart.
public void testProductAdd() {
    Product book = new Product("Refactoring", 53.95);
    _bookCart.addItem(book);

    assertTrue(_bookCart.contains(book));

    double expected = 23.95 + book.getPrice();
    double current = _bookCart.getBalance();
    assertEquals(expected, current, 0.0);

    int expectedCount = 2;
    int currentCount = _bookCart.getItemCount();
    assertEquals(expectedCount, currentCount);
}
Test: Remove product

// Tests removing a product from the cart.
public void testProductRemove() throws NotFoundException {
    Product book = new Product("Harry Potter", 23.95);
    _bookCart.removeItem(book);

    assertTrue(!_bookCart.contains(book));

    double expected = 23.95 - book.getPrice();
    double current = _bookCart.getBalance();
    assertEquals(expected, current, 0.0);

    int expectedCount = 0;
    int currentCount = _bookCart.getItemCount();
    assertEquals(expectedCount, currentCount);
}
Test: Exception handling

// Tests removing an unknown product from the cart.
public void testProductNotFoundException() {
  try {
    Product book = new Product("Bones", 4.95);
    _bookCart.removeItem(book);

    fail("Should raise a NotFoundException");
  } catch (NotFoundException nfe) {
    // Should always take this path
  }
}
Part 3: Write a test suite

The method `suite()` assembles the test methods into a test suite – using reflection all methods named `testX()` will be part of the test suite.

```java
public static Test suite() {
    // Here: add all testX() methods to the suite (reflection).
    TestSuite suite = new TestSuite(ShoppingCartTest.class);

    // Alternative: add methods manually (prone to error)
    // TestSuite suite = new TestSuite();
    // suite.addTest(new ShoppingCartTest("testEmpty"));
    // suite.addTest(new ShoppingCartTest("testProductAdd"));
    // suite.addTest(...);

    return suite;
}
```
Part 4: Execute test suite

// Returns the name of the test case.
public String toString() {
    return getName();
}

// Main method: Call GUI
public static void main(String args[]) {
    String[] tcName = { ShoppingCartTest.class.getName() };
    junit.swinggui.TestRunner.main(tcName);
    // junit.textui.TestRunner.main(tcName);
}

Execute the main() method to run the tests:
$ java ShoppingCartTest
Demo

```java
};
```
Summary

• Test Myths
  Top-down, bottom-up, sandwich...
  Test Steps

• JUnit
  Test cases
  Test suites
  Fixtures