Unit Test

11/2006

Tuan Le

Contents

- Objectives
- Testing Responsibilities
- Main types of Testing
- Black Box Testing
- White Box Testing
- Levels of Testing
- Build scaffolding for incomplete programs
- 100% Method Coverage
- Defining UT Items – Regulations
- Defining UT Items – Quality Standards
- UT Reporting – Regulations
- Unit Test Package
- References

Objectives

- To help the participants gain knowledge about Software Testing Principles and the Technology
- To implement Testing techniques and processes

Software testing

- The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software item (IEEE, 1986; IEEE, 1990).
- An activity that should be done throughout the whole development process (Bertolino, May 2001).

Testing Responsibilities (1)

- **Verification** is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.
- **Validation** is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements

Testing Responsibilities (2)

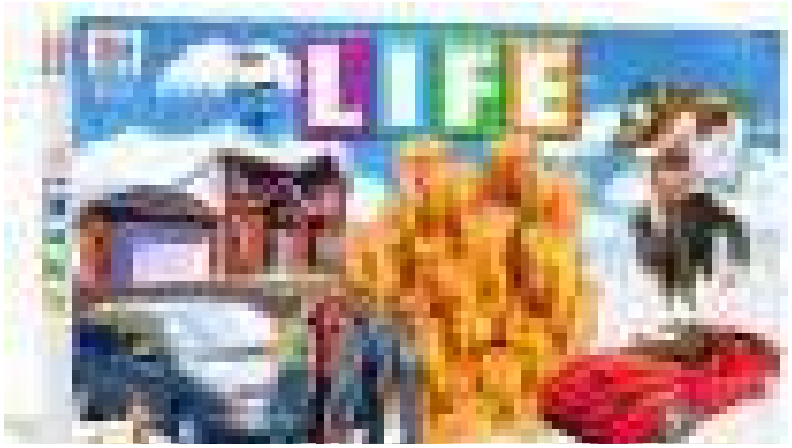
Verification

Are we building the product **right**?

"I landed on "Go" but didn't get my $200!"

Validation

Are we building the **right** product?



"I know this game has money and players and "Go" – but this is not the game I wanted."

What Is Unit Testing?

- Unit Testing
  - A test designed to demonstrate that a given unit or component performs correctly
  - Although many testing techniques apply at all levels, they are easiest to understand, to apply, and to use tool support at the unit level

Why Unit Test?

- Unit testing tests each component in isolation from the system
  - Focus is on correct functionality of unit
  - Detect many failures before integration
    - Usually catches more obvious bugs
    - Usually done by developer, if done at all

When to Unit Test

- Unit Test Planning should occur before coding
  - Not typically done formally
  - Unit testing usually unstructured, ad hoc
- Unit Testing should occur right after unit coding
  - Rework based on test results should also be unit tested

Unit Test Case Design

- Techniques
  - Black-box analysis of specification
  - White-box analysis of program structure
- Unit Testing considerations
  - Environmental limits (e.g., memory)
  - State transitions that change behavior
  - Historical defects (same/similar units)
  - Error handling

Main types of Testing

The two main types of testing are white box and black box testing.

- **White box:**
  - Used to test specific paths through the code.
  - At decision points you can:
    - Test the boundaries of the decision (boundary testing)
    - Test The partitions of the decision (partition testing).
- **Black box:**
  - What happens in the program is invisible and unimportant to the user.
  - Black box test cases only look at specific inputs and outputs of an application

Levels of Testing

Unit Testing Techniques

- **Structural Techniques:**
  - It is a White box testing technique that design test cases based on internal structure.
- **Functional testing techniques**
  - These are Black box testing techniques which tests the functionality of the application
- **Error based Techniques**
  - The best person to know the defects in his code is the person who has designed it.

Structural Techniques

- **Statement Testing:** A test strategy in which each statement of a program is executed at least once.
- **Path Testing:** Testing in which all paths in the program source code are tested at least once.
- **Condition Testing:** Condition testing allows the programmer to determine the path through a program by selectively executing code based on the comparison of a value
- **Expression Testing:** Testing in which the application is tested for different values of Regular Expression.

Functional testing techniques

- **Input domain testing:** This testing technique concentrates on size and type of every input object in terms of boundary value analysis and Equivalence class.
- **Boundary Value:** Boundary value analysis is a software testing design technique in which tests are designed to include representatives of boundary values.
- **Syntax checking:** This is a technique which is used to check the Syntax of the application.
- **Equivalence Partitioning:** This is a software testing technique that divides the input data of a software unit into partition of data from which test cases can be deriveda

Error based Techniques

- **Fault seeding** techniques can be used so that known defects can be put into the code and tested until they are all found.
- **Mutation Testing:** This is done by mutating certain statements in your source code and checking if your test code is able to find the errors. Mutation testing is very expensive to run, especially on very large applications.
- **Historical Test data:** This technique calculates the priority of each test case using historical information from the previous executions of the test case.

Build scaffolding for incomplete programs

- Scaffolding
  - Defined as computer programs and data files built to support software development.
  - Testing but not intended to be included in the final product.
  - Scaffolding code involves the creation of stubs and test drivers.

- Stubs and drivers are code that are (temporarily) written in order to unit test a program

Driver and Stub

- **Driver** is a software module used to invoke a module under test and often, provide test inputs, controls, and monitor execution and report test results

```
main () {
        movePlayer(Player, diceRoll);
}
```

- **Stub** is a module that simulates components that aren't written yet, formally defined as a computer program statement substituting for the body of a software module that is or will be defined elsewhere

```
void movePlayer(Player player, int diceValue) {
        player.setPosition(1);
}
```

100% Method Coverage

- All methods in all classes have been called
- Test case 1: Foo(0, 0, 0, 0, 0) = 0.0
- float foo (int a, b, c, d, e) {

```
    if (a == 0) {
        return 0.0;
    }
    int x = 0;
    if ((a==b) OR ((c==d) AND bug(a) )) {
        x =1;
    }
    e = 1/x;
    return e;
}
```

100% Method Coverage

- All lines in a method have been executed
- Test case 2: Foo(1, 1, 1, 1, 1) = 1.0
- void foo (int a, b, c, d, e) {

```
    if (a == 0) {
        return;
    }
    int x = 0;
    if ((a==b) OR ((c==d) AND bug(a) )) {
        x =1;
    }
    e = 1/x;
}
```

100% Method Coverage

- All predicates have been true and false
- Test case 3: Foo(1, 2, 1, 2, 1) ← division by zero!
- void foo (int a, b, c, d, e) {

```
    if (a == 0) {
        return;
    }
    int x = 0;
    if ((a==b) OR ((c==d) AND bug(a) )) {
        x =1;
    }
    e = 1/x;
}
```

Loops

- **Write a test case such that you:**
  - **Don’t go through the loop at all**
  - **Go through the loop once**
  - **Go through the loop twice**

Defining UT Items – Regulations

- UT Items of a class must be in the same XLS file
- UT Item File Name, following format
  UT_<Component/Class name>_<Author>
  - Component/Class Name: CM, CC, SL, SCRUtil,…
  - Author: DuongLe, DungNguyen, MaiTran, …
    - Example: UT_CM_DungNguyen.XLS
- UT Items Definition Files must be checked in SC, at folder/subfolder “Test Suite” of corresponding component

Defining UT Items – Quality Standards

Total number of UT Items must be at least 1/10 of LOC of the relevant code under test

UT Reporting – Regulations(1)

- UT results (passed/failed items) for a specific version of a component/classes must be kept in a separate XLS file
- Report file name: following template

UT_REPORT_<Component/Class>_TESTER.XLS

To indicate UT Items Define file that is used

- Example:

UT_REPORT_CM_DungNguyen.XLS

UT Reporting – Regulations (2)

- Unexpected errors (may or may not related to a failed items): must be recorded into a separate Bug List (XLS)
  - Template: BugList(template).xls
- Bug List (XLS) have similar file name as the related Test Report:

UT_BUGLIST_<Component/Class>_<Tester>.XLS

To indicate UT Items Define file that is used

UT Reporting – Quality Standards

- Total Bugs = Total 'Fail' Items + Total bugs (in Bug List)
- Bug Density (BD): Total Bugs / 1 KLOC
- BD < 6
  - Not effective UT
  - TO-DO: define/revise more UT Items
- BD > 14
  - Source code is not good
  - TO-DO: Code Inspection again (may be rewrite the related component/class)

Unit Test Package

- Compiles with production compiler options (no warning messages).
- Version control applied.
- All tests in the unit test plan have passed.
- Code walkthrough completed.
- Acceptable complexity levels.
- Follows department coding standards.
- Documentation complete.
- Source code uses comments.
- 100% statement coverage in testing.
- Test databases or other data structures modified.
- No undocumented features in program.
- Source control entered.

Unit Testing Best Practices

- Ensure each Unit Test case is independent of each other.
- Name your unit tests clearly and consistently.
- Before changing a module interface or implementation, make sure that the module has test cases and that it passes its tests before changing the implementation.
- Always ensure the bug identified during Unit Testing is fixed before moving it to the next phase.

Reference

- **Unit Test Quality Standards**

  Author: Tien Duong
- **White Box**

  Author: Laurie Williams 2004
- **Unit Testing template**

  UT_CSTA_TuanLe.xls
- **Unit Testing report template**

  UT_Report_CS_TuyenNguyen.xls
- **Bug List template**

  UT_BugList_CM_DungNguyen.xls

Document Revision History

Date	Version	Description	Revised by
7 Nov 2006	1.0	First version	Tuan Le

<NOTE: Delete this slide if your document is Not necessary to control its version>