Software Testing and Quality Assurance

Meti Dejene
tiimeekoo@gmail.com
Haramaya University

Lecture 4 – Software Testing Types and Techniques - 1

Software Testing Techniques

- After all, testing is the design of effective test cases so that most of the testing domains will be covered detecting the maximum number of errors.
- Therefore, the next task is to find the technique which will meet both the objectives of effective test case design, i.e. coverage of testing domain and detection of maximum number of errors.
- Techniques used to design such effective test case are called Software Testing Techniques.
- These software testing techniques are broadly categorized into two types:

    I. Dynamic testing and

    II. Static testing

I. Dynamic Testing

Introduction

- Dynamic testing is a testing technique that executes the software being built with a number of test cases.
- Unlike Static testing, in this technique the program code is run on a number of inputs provided and the corresponding results are checked.
- So, all the testing methods that execute the code to test a software product are known as dynamic testing techniques.

Cont.

- In dynamic testing, test cases can be designed and testing can be conducted in one of two ways:

  1) knowing only the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function;

  2) knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh," that is, internal operations are performed according to specifications and all internal components have been adequately exercised.

- The first test approach is called black-box testing and the second, white-box testing.

A. Black-box Testing

- Black-box testing, also known as functional testing or behavioral testing, focuses on the functional requirements of the software or module.
- This testing method examines functional aspect of a system with little or no regard for the internal logical structure of the software.
- In black-box testing, sets of test cases that will fully exercise all functional requirements for a program are designed based on functional specifications.
- Then, Input test data is given to the system, which is a black box to the tester, and results are checked against expected outputs after executing the software.

Cont.

- In such a way, Black-box tests are used to demonstrate that
  - software functions are operational,
  - input is properly accepted and output is correctly produced, and
  - the integrity of external information (e.g., a database) is maintained.
- Overall, Black-box testing attempts to find errors such as:
  - incorrect or missing functions
  - errors in database access
  - behavior errors and others.

Dynamic Black-box Testing Techniques

1. BOUNDARY VALUE ANALYSIS (BVA)

- Most of the time, a greater number of errors tend to occur at the boundaries of the input domain rather than in the center.
- Based on this, BVA is a technique that uncovers the errors at the boundary of input values by selecting test cases that exercise bounding values.
- For example,
  - If A is an integer between 10 and 255, then boundary checking can be on 10 (9,10,11) and on 255 (256,255,254).
- We can implement BVA using 3 methods.

(1) Boundary Value Checking (BVC) (2) Robustness Testing Method (3) Worst-case Testing Method

Boundary Value Checking (BVC) Method

- In this method, the test cases are designed by holding one variable at its extreme value and other variables at their nominal values in the input domain, repeating this for each variable.
- For a given variable, from the input domain its extreme value are
  - Minimum value (Min)
  - Value just above the minimum value (Min+)
  - Maximum value (Max)
  - Value just below the maximum value (Max−).
- The nominal value for a variable is taken as the middle value of the range.

BVC cont.

- For example, if we have two variables A and B, all the possible test case combination are

1. A_{nom} , B_{min}
2. A_{nom} , B_{min+}
3. A_{nom} , B_{max}
4. A_{nom} , B_{max-}
5. A_{nom} , B_{nom}
6. A_{min} , B_{nom}
7. A_{min+} , B_{nom}
8. A_{max} , B_{nom}
9. A_{max-} , B_{nom}

- So, in general for n variables in a module, 4n + 1 test cases can be designed with boundary value checking method.

Robustness Testing Method

- This method extends the BVC with 2 additional input values
  - A value just greater than the Maximum value (Max+) and
  - A value just less than Minimum value (Min−)
- Then in addition to the previous 9 test cases in BVC, we will have the following 4 additional test cases

10. A_{max+} , B_{nom}
11. A_{min-} , B_{nom}
12. A_{nom}, B_{max+}
13. A_{nom} , B_{min-}

- So, in general for n input variables in a module, 6n + 1 test cases can be designed with robustness testing.

Worst-case Testing Method

- In this method, the test cases are designed by combining each input value (extreme and nominal) of one variable with each input value of the other variable.
- So, in general for n input variables in a module, 5^n test cases can be designed with worst-case testing.

Example 1

- A program reads an integer number within the range [1,100] and determines whether it is a prime number or not. Design test cases for this program using BVC, robust testing, and worst-case testing methods.

Solution

(a) Test cases using BVC

- Since there is one variable, the total number of test cases will be $4n + 1 = 5$.

- So, the set of minimum and maximum values is

Min value = 1
Min^+ value = 2
Max value = 100
Max^- value = 99
Nominal value = 50–55

- Using these values, test cases designed are

Test Case ID	Integer Variable	Expected Output
1	1	Not a prime number
2	2	Prime number
3	100	Not a prime number
4	99	Not a prime number
5	53	Prime number

(b) Test cases using robust testing

- Since there is one variable, the total number of test cases will be $6n + 1 = 7$.
- The set of boundary values are

Min value = 1
Min^- value = 0
Min^+ value = 2
Max value = 100
Max^- value = 99
Max^+ value = 101
Nominal value = 50–55

- Using these values, test cases designed are

Test Case ID	Integer Variable	Expected Output
1	0	Invalid input
2	1	Not a prime number
3	2	Prime number
4	100	Not a prime number
5	99	Not a prime number
6	101	Invalid input
7	53	Prime number

Solution

(c) Test cases using worst-case testing

- Since there is one variable, the total number of test cases will be $5^n = 5$.
- Therefore, the number of test cases will be same as BVC.

- For further examples and explanation
  - Refer Example 4.2 , Example 4.3 and Example 4.4, in "Software Testing: Principles and Practices" by Naresh Chauhan.

2. EQUIVALENCE CLASS PARTITIONING

- Equivalence class partitioning is a method that divides or partitions the input domain of a program based on a common feature into classes of data from which test cases can be derived.
  - These partitioned class of data are called Equivalence Classes.
- **Equivalence class**: A set of input data or output results that the system is expected to handle identically.
  - Valid Equivalence Class: A set of input values that are defined and accepted by the specifications.
  - Invalid Equivalence Class: A set of input values that are explicitly outside the allowed specifications and should be rejected by the system.

Cont.

- The underlying principle is simple:
  - If a software component works correctly for one test case in an equivalence class, it will likely work correctly for all other test cases in that same class. Conversely, if one test case from a partition causes an error, all others in that partition are also expected to cause the same error.
- So, instead of testing every input, only one test case from each partitioned class can be executed.
  - It means only one test case in the equivalence class will be sufficient to find errors.

Cont.

Equivalence class partitioning involves two steps:

Step 1: Identify equivalence classes

- Partition the whole input domain into different equivalence classes.
- Two types of equivalence classes can always be identified :
  - *Valid equivalence classes*: These classes consider valid inputs to the program.
  - *Invalid equivalence classes*: These classes consider invalid inputs that will generate error conditions or unexpected behavior of the program.
- Assign a unique identification number to each equivalence class.

Cont.

Step 2 : Design test cases

- Drive test cases from both the valid and invalid equivalence class that can represent, cover and exercise each identified equivalence classes appropriately.

3. USE CASE TESTING

- Use Case Testing is a technique where test cases are designed based on use cases.
- Use cases
  - describe the interactions between a user (or actor) and the system to achieve a specific goal.
  - provide a detailed description of how the system is expected to behave in different scenarios, which makes them a valuable source for creating comprehensive test cases.
- Use Case Testing is a powerful technique that leverages the detailed scenarios described in use cases to create thorough and effective test cases.
- It ensures that all functional requirements are tested, including main and alternate flows and the system handles various user interactions correctly.

Steps to Apply Use Case Testing

1. Identify Use Cases:
   - Gather all the use cases from the system's requirements documentation.
   - Ensure that you cover all the main functionalities and scenarios the system should handle.
2. Understand the Flow:
   - For each use case, understand the main flow (normal scenario) and alternate flows (exception scenarios).
   - Pay attention to preconditions (what must be true before the use case starts) and postconditions (what should be true after the use case completes).

Cont.

3. Extract Scenarios:
   - Break down each use case into individual scenarios, covering both the main flow and any alternative flows.
   - Include variations like different paths the user can take and error handling.
4. Design Test Cases:
   - For each scenario, design test cases that verify the system's behavior.
   - Include detailed steps, input data, expected results, and any specific conditions that need to be met.
5. Prioritize and Execute:
   - Prioritize test cases based on criticality, frequency of use, and risk.
   - Execute the test cases, ensuring that each scenario behaves as expected.