Software Testing: A Craftsman's Approach

Study Notes

Chapter 1: Introduction to Software Testing

Why Do We Test?

We test software to:

1. **Judge the quality** *or acceptability of a product.*
2. **Uncover issues or problems** *that might exist in the software.*

Reason for Testing:

Humans can make mistakes, especially when creating complex systems. These mistakes, called **errors***, often lead to* **faults (bugs)***. If these faults are not caught, they can cause the software to* **fail***.*

Testing Life Cycle

Testing involves:

1. **Planning***: Define what to test and how.*
2. **Developing Test Cases***: Create structured tests with inputs and expected outputs.*
3. **Running Test Cases***: Execute the tests.*
4. **Analyzing Results***: Check if the software behaves as expected.*

Goal of Testing:

Identify faults before they cause problems.

Handling Faults of Omission

Faults of omission are tricky because they may go undetected until triggered.
Solution: *Use* ***reviews*** *where experts carefully examine the code and design to spot potential issues.*

Test Cases

A ***test case*** *is a structured way to test specific behaviors in software. It includes:*

- *Bug Name*
- *ID*
- *Description*
- *Scenario*
- *Pre-condition*
- *Inputs*
- *Expected Output*
- *Post-condition*
- *Execution History*

Importance: *Test cases are as valuable as the software code itself. They need careful management and review to ensure thorough testing.*

Venn Diagram Approach to Testing

The Venn diagram clarifies the relationship between:

- ***Specified Behaviors (S)***: What the software is supposed to do.
- ***Program Behaviors (P)***: What the program actually does.

Key Points:

1. ***Fault of Omission***: Specified behavior not programmed.
2. ***Fault of Commission***: Software does something not specified.
3. ***Intersection of S and P***: Behaviors that are both specified and correctly implemented.

Specification-Based Testing (Functional/Black Box Testing)

- Focuses on verifying software functionality based on its specifications.
- Treats the program as a **black box** (only cares about inputs and outputs).

Advantages:

- Test cases can be created before or alongside development.

Disadvantages:

- Test cases might overlap.
- Gaps in coverage may leave some behaviors untested.

Techniques:

1. Boundary Value Analysis
2. Worst Case Analysis
3. Robustness Testing

4. Equivalence Class Partitioning
5. Decision Table-Based Testing

Chapter 2: Code-Based Testing (White Box Testing)

What is Code-Based Testing?

- Focuses on examining the **internal structure** of the software.
- Test cases are derived from the code implementation.

Advantages:

- Allows testers to check if all parts of the code are functioning correctly.
- Uses **test coverage metrics** to quantify how much code has been tested.

Fault Taxonomies

Faults can be classified based on:

1. **Development Phase:** Requirement, Design, Coding.
2. **Consequences of Failures:** Impact on the system.
3. **Difficulty to Resolve:** Some faults are easier to fix than others.
4. **Risk to Resolution:** Likelihood of a fault going unresolved.
5. **Occurrence:**
    - One-time
    - Intermittent
    - Recurring
    - Repeatable (easiest to fix)

Logic Faults

Common logic faults include:

- *Missing Case*
- *Duplicate Case*
- *Extreme Condition Neglected*
- *Misinterpretation*
- *Missing Condition*
- *Test of Wrong Variable*
- *Incorrect Loop Iteration*
- *Wrong Operator*

Pseudocode

- *A simplified way to write program logic without worrying about syntax.*
- *Focuses on **what** and **why** instead of **how**.*

Example:

Instead of writing if x > 5, pseudocode might say, "If x is greater than 5."

Chapter 3 & 4: Discrete Mathematics for Testers

Topics Covered:

1. ***Set Theory:** Deals with collections of objects and their relationships.*
2. ***Functions:** Mapping of inputs to outputs.*

3. **Relations:** *Connections between entities.*
4. **Propositional Logic:** *Reasoning about statements and their truth values.*
5. **Probability Theory:** *Assessing the likelihood of outcomes or errors.*
6. **Graph Theory:**
    - **Undirected Graphs:** *Relationships are not directional.*
    - **Directed Graphs:** *Relationships are directional (e.g., Person A follows Person B).*

Chapter 5: Boundary Value Testing

What is Boundary Value Testing?

- *Focuses on testing at the boundaries of input domains.*
- *Input Domain Testing is also called* **Boundary Value Analysis (BVA)**.

Types of BVA:

1. **Normal BVA:** *Tests valid inputs.*
2. **Robust BVA:** *Tests both valid and invalid inputs.*
3. **Worst Case BVA:** *Tests extreme combinations of input values.*
4. **Robust Worst Case BVA:** *Combines invalid inputs with extreme combinations.*

Limitations:

- *May miss gaps in coverage.*

Boundary Value Analysis (BVA) Types with Examples

Boundary Value Analysis (BVA) is a test design technique that focuses on testing the boundary values of input domains, as errors often occur at the edges rather than in the middle of an input range.

---

1. Normal BVA (Tests only valid boundary values)

- **Tests only the valid boundary values** (minimum, maximum, just below, just above).
- **Example**: If an age input field accepts values from **18 to 60**, test cases would be:
  - **Lower Bound**: 18, 19
  - **Upper Bound**: 59, 60

---

2. Robust BVA (Tests both valid and invalid boundary values)

- **Tests valid and invalid boundary values** (just outside the valid range).
- **Example**: For an age range of **18 to 60**, test cases would include:
  - **Lower Bound**: **17 (invalid), 18 (valid), 19 (valid)**
  - **Upper Bound**: **59 (valid), 60 (valid), 61 (invalid)**

---

3. Worst Case BVA (Tests extreme combinations of valid boundary values)

- **Tests all possible combinations of min/max values together.**
- **Example**: *A system accepts inputs for* **age (18-60) and salary ($30,000-$100,000)**.
  - *Test Cases:*
    - *(18, $30,000)*
    - *(18, $100,000)*
    - *(60, $30,000)*
    - *(60, $100,000)*

---

4. Robust Worst Case BVA (Combines invalid values with extreme cases)

- **Tests all extreme valid cases plus invalid boundary values.**
- **Example**: *Using the same* **age (18-60) and salary ($30,000-$100,000)** *constraints:*
  - *Test Cases:*
    - *(17, $29,999) → Invalid*
    - *(17, $100,001) → Invalid*
    - *(61, $30,000) → Invalid*
    - *(61, $100,001) → Invalid*

BVA Type	Tests Valid Inputs	Tests Invalid Inputs	Tests Extreme Combinations

Normal BVA	✅	❌	❌
Robust BVA	✅	✅	❌
Worst Case BVA	✅	❌	✅
Robust Worst Case BVA	✅	✅	✅

Chapter 6: Equivalence Testing

What is Equivalence Testing?

- Organizes inputs into distinct groups (equivalence classes) to reduce redundancy.
- Based on two factors:
    1. Single vs. Multiple Fault (Weak vs. Strong).
    2. Normal vs. Invalid.

Types:

1. Weak Normal
2. Strong Normal
3. Weak Robust
4. Strong Robust

1. Weak Normal Equivalence Testing

- ***Tests one valid input per equivalence class*** *without considering multiple conditions simultaneously.*
- ***Example****: A login form that requires a* ***username*** *and* ***password****.*
  - *Equivalence classes:*
    - *Valid username (e.g., "user123")*
    - *Valid password (e.g., "Pass@123")*
  - *Test Case: Enter a valid username and a valid password separately.*

2. Strong Normal Equivalence Testing

- ***Tests multiple valid inputs together*** *to check how they interact.*
- ***Example****: In the same login form:*
  - *Test case: Enter a valid username ("user123")* ***and*** *a valid password ("Pass@123") at the same time.*

3. Weak Robust Equivalence Testing

- ***Tests one invalid input per equivalence class at a time****, keeping other inputs valid.*
- ***Example****: Login form*
  - *Invalid username ("" empty string) with a valid password ("Pass@123")*
  - *Valid username ("user123") with an invalid password ("")*

4. Strong Robust Equivalence Testing

- ***Tests multiple invalid inputs together*** *to check system behavior under extreme conditions.*
- ***Example****: Login form*

- *Invalid username ("") **and** invalid password ("") entered together to check the system's response*

Chapter 7: Decision Table Testing

What is Decision Table Testing?

- *A tool to represent complex logical relationships and analyze actions under different conditions.*

Structure:

1. ***Stub Portion (Left Side):*** *Lists conditions and actions.*
2. ***Entry Portion (Right Side):*** *Contains rules for actions based on conditions.*
3. ***Condition Portion (Above Horizontal Line):*** *Specifies conditions being evaluated.*
4. ***Action Portion (Below Horizontal Line):*** *Lists resulting actions.*

Rules:

- *Each column in the entry portion is a rule.*
- ***Don't Care Entries:*** *Represent irrelevant conditions (denoted by n/a).*

Decision Table Testing

Decision Table Testing *is a black-box testing technique used to represent and analyze complex logical relationships between conditions and actions. It helps ensure all possible combinations of inputs are tested.*

Structure of a Decision Table

1. **Stub Portion (Left Side):** Lists all conditions and corresponding actions.
2. **Entry Portion (Right Side):** Defines different rules by specifying values for conditions and their resulting actions.
3. **Condition Portion (Above Horizontal Line):** Specifies conditions being evaluated.
4. **Action Portion (Below Horizontal Line):** Lists actions that occur based on the condition values.

- **Rules:** Each column in the entry portion represents a unique rule (test case).
- **Don't Care Entries:** Represent conditions that don't affect the outcome (denoted by `-` or `n/a`).

Example: Online Shopping Discount System

Scenario:

A store offers a discount based on **membership status** and **purchase amount**:

- **Gold members** get a **20% discount** on any purchase.
- **Silver members** get a **10% discount** if they spend **$100 or more**.
- **Non-members** get no discount.

Conditions	Rule 1	R2	R3	R4	R5	R6
Is Customer Gold?	Yes	No	No	No	No	No
Is Customer Silver?	-	Yes	Yes	Yes	No	No
Purchase =>100	-	Yes	No	Yes	Yes	No
Actions						
Give 20% Discount	✅	❌	❌	❌	❌	❌
Give 10% Discount	❌	✅	❌	✅	❌	❌
No Discount	❌	❌	✅	❌	✅	✅

Test Cases Derived from Decision Table:

1. **Gold Member:** Always gets 20% discount.
2. **Silver Member, Purchase ≥ $100:** Gets 10% discount.
3. **Silver Member, Purchase < $100:** No discount.
4. **Non-Member, Purchase ≥ $100:** No discount.
5. **Non-Member, Purchase < $100:** No discount.

Benefits of Decision Table Testing:

✔ Ensures coverage of all possible conditions.

✔ Helps in systematic test case design.

✔ Clearly represents complex decision logic.

Chapter 8: Path Testing

Path Testing

Path Testing is a white-box testing technique that focuses on ensuring that all possible execution paths in a program are tested at least once. It helps detect unreachable code, missing logic, and redundant paths.

Common Forms of Path Testing

01. **Style Choice:**

    a. Focuses on selecting specific paths to test based on coding style or logical structure.
    b. Used in structured programming to ensure clean and maintainable code.

02. **Decision-to-Decision (DD) Path:**

    a. Involves testing paths between decision points (if-else, loops, etc.).
    b. Ensures all logical branches in the control flow are covered.

Example of Path Testing

Scenario: A Simple Login System

```
def login(username, password):
    if username == "admin":
        if password == "admin123":
            return "Login Successful"
        else:
            return "Incorrect Password"
    else:
        return "Invalid Username"
```

Control Flow Diagram

1️⃣ Start

2️⃣ Check if username is "admin"

- Yes → Go to step 3
- No → Return "Invalid Username"

3️⃣ Check if password is "admin123"

- Yes → Return "Login Successful"
- No → Return "Incorrect Password"

Possible Execution Paths:

Path	Condition Evaluation	Output
P1	username = "admin", password = "admin123"	"Login Successful"
P2	username = "admin", password ≠ "admin123"	"Incorrect Password"
P3	username ≠ "admin"	"Invalid Username"

Benefits of Path Testing:

✔ *Ensures all decision points and branches are tested.*
✔ *Helps detect untested or redundant code.*
✔ *Increases code coverage and reduces risk of defects.*

Chapter 9: Data Flow Testing

What is Data Flow Testing?

Data Flow Testing *focuses on how variables (like numbers or text) are defined (given a value) and used (read or changed) in a program. The goal is to make sure that variables are assigned values before they are used.*

Key Concepts:

1. **Define-Use Path (DU Path):**
   - ***Define*** *means when a variable gets a value.*
   - ***Use*** *means when a variable is read or used.*
   - *A* ***DU Path*** *is the path from where a variable gets a value to where that value is used in the program.*
2. **Slice-Based Testing:**
   - *This method focuses on specific parts of the program that are important to test, based on how variables are used in those parts.*

Simple Example:

```
def calculate_bonus(salary, performance):
    bonus = 0  # 'bonus' is defined here
    if performance > 80:
        bonus = salary * 0.1  # 'bonus' is updated here
    return bonus

def main():
    salary = 50000  # 'salary' is defined here
    performance = 85  # 'performance' is defined here
    bonus = calculate_bonus(salary, performance)  # 'bonus' is used here
    print("Bonus:", bonus)
```

Data Flow Testing Example:

1. **Define:**
   - salary is defined as 50000 in main().
   - performance is defined as 85 in main().
   - bonus is defined as 0 inside the calculate_bonus() function.
2. **Use:**
   - salary is used to calculate the bonus: bonus = salary * 0.1.
   - performance is used in the condition if performance > 80: to decide whether the bonus will be given.
   - bonus is returned and printed in main().

Why This is Important:

We want to make sure that:

- The variable salary is given a value **before** it's used to calculate the bonus.
- The variable performance is given a value **before** it's checked to see if it qualifies for a bonus.
- The variable bonus is correctly updated and printed.

Summary:

- **Data Flow Testing** helps ensure variables are given values before being used.
- It checks the **path** from where a variable gets a value to where it is used.
- This type of testing helps avoid mistakes like using uninitialized variables.

This method makes sure everything flows correctly and variables are properly handled!

Chapter 10: Unit Testing

When to Stop Unit Testing?

1. When you run out of time.
2. When no new failures are found.
3. When no new faults appear.
4. When no new test cases come to mind.
5. When diminishing returns are reached.
6. When test coverage is met.
7. When all faults are removed (ideal but impossible).

Chapter 11: Life Cycle-Based Testing

Software Development Models:

1. **Waterfall Model:** *Sequential phases (Requirements > Design > Implementation > Testing > Deployment).*
2. **Agile Testing:** *Continuous testing integrated throughout development.*
3. **Model-Based Testing (MBT):** *Uses software models to generate test cases.*

Chapter 13: Integration Testing

What is Integration Testing?

- *Ensures different components work together as intended.*
- **Example:** *The Mars Climate Orbiter failure occurred due to a lack of integration testing (one system used pounds, the other used newtons).*

Types:

1. **Top-Down Integration:** *Starts with the main program and integrates lower-level units.*
2. **Bottom-Up Integration:** *Starts with lower-level units and integrates upwards.*

Top-Down Integration Testing

- ***Example:*** *Start testing the "**OrderProcessing**" module (main program) first, using **stubs** for the "**Payment**" and "**Inventory**" modules.*

Bottom-Up Integration Testing

- ***Example:*** *Start testing the "**Inventory**" and "**Payment**" modules first, and then integrate them with the "**OrderProcessing**" module.*

Exploratory Testing

Imagine someone rushes into the hospital emergency room (ER) because they are having trouble breathing , the doctor job is to figure out what wrong and how to treat

Step 1 : gather information
So the doctor first ask about the medical history to understand thier condition later

Step 2 : Run General Test Since there are many possible reason for breathing problems the doctor start with specific test to pinpoint the issue

Step 3 : follow up test , based on the result of the first test the doctor conducts more specific test to pinpoint the exact issue

Step 4 : Use Experience : The doctor relies on their knowledge and past experience to make decision throughout the process

Now same approach is use in exploratory testing

Testing first gather basic information about the software

They perform general test to identify common issue

They run more specific test based on what they find

Their experience helps them navigate the process efficiently

Test Driven Development

Test Driven Development (TDD)

Write a test , write some code , write another Test , write more code and repeat

How TDD works

Write a test first - Before writing any actual code ,a test case is created based on what code should do

Test Fail Intailly - since the code doesn not exist yet the test will fail

Write Just Enough Code : Just Enough code is written to make the test pass

Refactor if needed : as a codebase grows it can be cleaned up (refactored) but all previous test must still pass to ensure nothing breaks

(Why TDD is more popular in Agile Development)

It avoids writing unnecessary code (the principle of you arent going to need it)

It makes debugging easier , if a new test fails the issue is likely in the most recent written code

TDD follows user stories : which are short description of what a user wants the software to do

Test Driven Development (TDD) works best when its easy to write and run code

To support this special tools called test framework exist for different framework

How TDD is supported

Test Execution Frameworks : these are the tools that help run test case automatically

Test Driver Programs : Developer write small inputs that include test input and expected output to check if the code work correctly

Example : Java and Junit : Junit is a popular testing framework that makes TDD easier

Popular TDD Framework

C - Cunit ,CuUnit

C++ - CPPUnit

Java - Junit

Python - PyUNit

Php - Phpunit

Javascript - Jsunit

.Net Languages (C# ,.Net) - Nunit , csunit

Objective C -OCunit

Fortran - Fruit ,funit

Databases - DBUNit

Basics of All pair Testing

Identify Input Parameter

Example : A mobile app screen might have

Browser : Chrome , firefox , edge

Devveice Type : Mobile

Network : Wifi and mobile data

Generate All Possible Combination

If we test every combination we get 3x2x2= 12 test cases

That is a lot and testing them can be time consuminhg

There we make use of All pair testing

Instead of testing all 12 combination we only test unique pairs of value

Benefits of All pair Testing

Fewer Test cases : saves time and efforts

Better defect detection : Covers all two factor interactions

Applicable to many scenarios : useful for web apps , mobile apps and hardware testing

Works well in agile Development : Helps quickly find and fix issue