Week 4 Summary: Software Quality and Testing

Name: Sushma Singh

Unit: SIT707

Student Id: 223743838

Decision Table Testing

Purpose

Decision table testing is a black-box testing technique used to model and validate logic in systems with multiple conditions. It helps in understanding how different input combinations affect system outputs. It's especially valuable for **rule-based logic** or business rules.

Key Features

- Maps **input conditions** and **output actions** in a tabular format.
- Each **row** represents a condition or action.
- Each **column** is a unique combination of inputs and corresponding output (called a **rule**).
- Also referred to as a **cause-effect table**, where causes are conditions and effects are actions.

Structure

- **Condition Stub**: Lists all input variables.
- **Condition Entry**: The values each condition can take (e.g., T/F or valid/invalid).
- **Action Stub**: Lists possible outcomes or system actions.
- **Action Entry**: Shows which action should occur for a given rule.

Use Cases

- **Login Systems**: Valid vs. Invalid credentials.
- **Troubleshooting**: Example - Printer issues (no print, flashing red light).
- **Eligibility Check**: Library system for borrowing books (registered, no overdue fees, within borrow limit).
- **Date Calculation**: NextDate logic (considering leap years, month lengths).
- **Geometric Logic**: Triangle classification (equilateral, isosceles, scalene).

Advantages

- Helps model **complex decision logic** clearly and exhaustively.
- Ensures **complete test coverage** by accounting for all input combinations.
- Allows simplification by **merging columns** with identical actions.

Limitations

- **Exponential growth** with increase in conditions (2^n combinations).
- Can become **unmanageable** without simplification.

Development Steps

1. Identify all relevant **input conditions**.
2. Determine the **values** each condition can take.
3. Calculate total possible combinations (rules).
4. Define all possible **system actions**.
5. Construct the decision table with rules and actions.
6. **Simplify** by merging columns with identical outcomes.
7. Derive **test cases** from unique rules.

Path Testing (Structural Testing)

- **Purpose**
- Path testing is a **white-box** technique used to ensure that every logical path in the program's source code is exercised. It highlights areas of poor logic coverage and helps find gaps and redundancies.

Program Graphs

- A **program graph** is a directed graph where:
  - Nodes = Code fragments or statements.
  - Edges = Control flow between them.

DD-Paths (Decision-to-Decision Paths)

- These are chains of program statements between two decisions or branching points.
- Defined by:
  - Entry node (start)
  - Exit node (end)
  - Internal nodes (one entry, one exit)

Test Coverage Metrics

- **Gnode**: All nodes (statements) are visited by the test set.
- **Gedge**: All control flow edges are covered.
- **Gchain**: All paths of two or more connected statements are tested.
- **Gpath**: All possible paths from start to end are tested (often infeasible).

Miller's Coverage Metrics

- **C0 (Statement Coverage)**: All statements run at least once.
- **C1 (DD-Path)**: All path segments tested.
- **C1p (Branch)**: True/False branches for each condition covered.
- **C2 (Loop)**: Entry and exit of loops tested.
- **Cd (Dependent Paths)**: Ensures that dependent logic between paths is tested.
- **Cik (Complex Loops)**: Covers complex loop structures like nested or knotted loops.

McCabe's Basis Path Testing

- Treats code as a graph, calculates **cyclomatic complexity**: V(G) = E - N + 2P
  - E = edges
  - N = nodes
  - P = components
- Basis paths = independent logical paths in code.
- Ensures that all logic can be validated with a **minimal set of tests**.

Loop Testing Strategy

- **Huang's Theorem**: Two tests per loop (normal traversal & exit condition).
- **Concatenated Loops**: Tested individually then merged.
- **Nested Loops**: Start testing from innermost loop.
- **Knotted Loops**: Suggest code refactoring.

CMCC (Compound Condition Testing)

- Used for conditions with multiple logical clauses (e.g., A && B || C).
- Can:
  - Use a truth table to expand logic.
  - Convert truth table to a decision table.
  - Generate test cases from these derived rules.

JUnit Assertions and Exception Handling

Assertions in JUnit

Used to verify if actual outcomes match expected ones.

- **assertTrue(condition)**: Passes if condition is true.
- **assertEquals(expected, actual)**: Checks if two values are equal.
- **assertThat(actual, matcher)** (Hamcrest): Expressive, readable assertions.

Exception Testing Styles

JUnit supports **three schools of exception handling**:

- **Simple School:**

```
@Test(expected = ArithmeticException.class)
public void divideByZero() { ... }
```

- **Old School:**

```
try {
        method();
        fail("Should have thrown exception");
    } catch (Exception e) { }
```

- **New School (ExpectedException Rule):**

```
@Rule
public ExpectedException thrown = ExpectedException.none();
@Test
public void test() {
  thrown.expect(MyException.class);
   method();
}
```

Active Learning Tasks

Task 1 – SimpleLoginForm:

- Create login logic in Java
- Define conditions (e.g., valid username & password)
- Create a decision table for login scenarios
- Implement JUnit tests with @Test annotations for each condition

Task 2 – ChatGPT JUnit Test Generation:

- Paste your Java code into ChatGPT
- Observe generated test methods
- Compare quality and structure with manually written tests

Discussion Points:

- Decision tables cover **multiple conditions at once**, unlike boundary value analysis (BVA) or equivalence partitioning.
- Ideal for **logic-heavy inputs** where all combinations and their outcomes matter.
- ChatGPT is useful for **initializing test structure**, though manual review and context-specific tweaking are often necessary.

Decision Table for SimpleLoginPage

Test Case ID	Username	Password	Condition: username = 'testuser'	Condition: password = 'testpassword'	Expected Output	Category
TC1	testuser	testpassword	TRUE	TRUE	username match	Pass login logic
TC2	testuser	wrongpass	TRUE	FALSE	username match	Partial match (username)
TC3	wronguser	testpassword	FALSE	TRUE	password match	Partial match (password)
TC4	wronguser	wrongpass	FALSE	FALSE	success	Pass (no matches)
TC5	null	testpassword	N/A (Exception/Error)	TRUE	Exception (NullPointer)	Invalid value (null input)
TC6	testuser	null	TRUE	N/A (Exception/Error)	username match	Invalid value (null input)
TC7	""	testpassword	FALSE	TRUE	password match	Edge case (empty username)
TC8	testuser	""	TRUE	FALSE	username match	Edge case (empty password

Key Takeaways

Decision Table Testing: Models complex logic compactly and exhaustively.

Path Testing: Ensures thorough execution of all logic flows using control graphs.

JUnit: Enables maintainable, automatable, and readable unit tests.

Coverage Metrics: Help track test thoroughness but don’t guarantee fault detection.

- **Combined Use**: Both functional (black-box) and structural (white-box) techniques improve quality and reliability.