Chapter 18: Testing Testing Conventional Applications

(Part 2 - Black-Box & Advanced Testing)

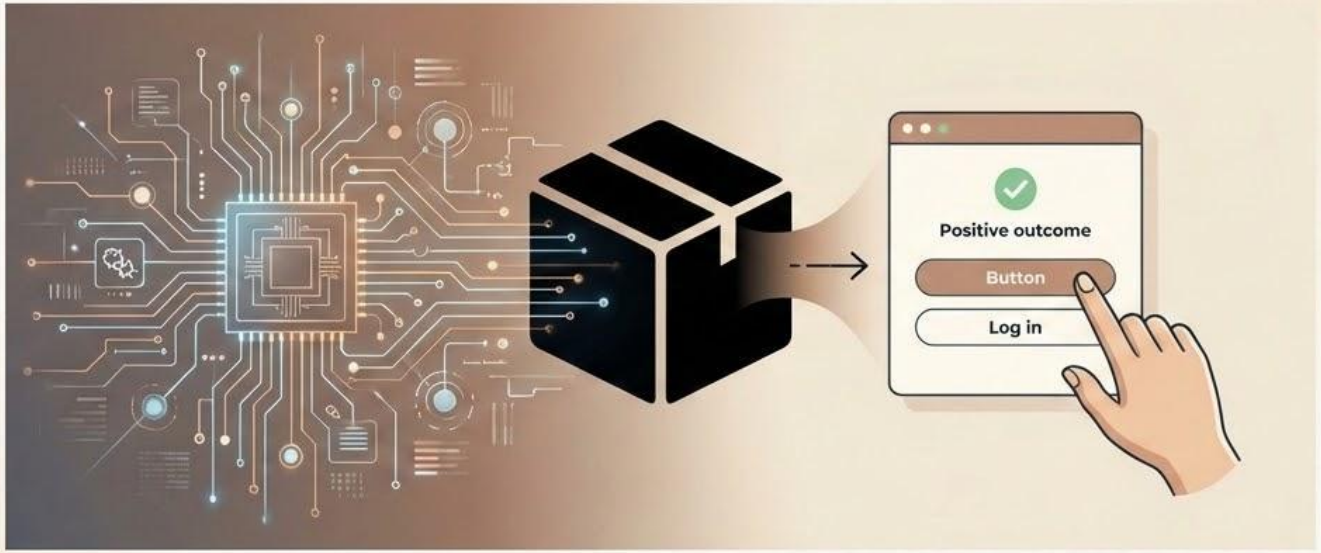
Subject: Software Engineering

Program: BTech Computer Science and Engineering

Duration: 1 Hour

Testing the User's Reality

Validating Software Against Requirements, Not Code

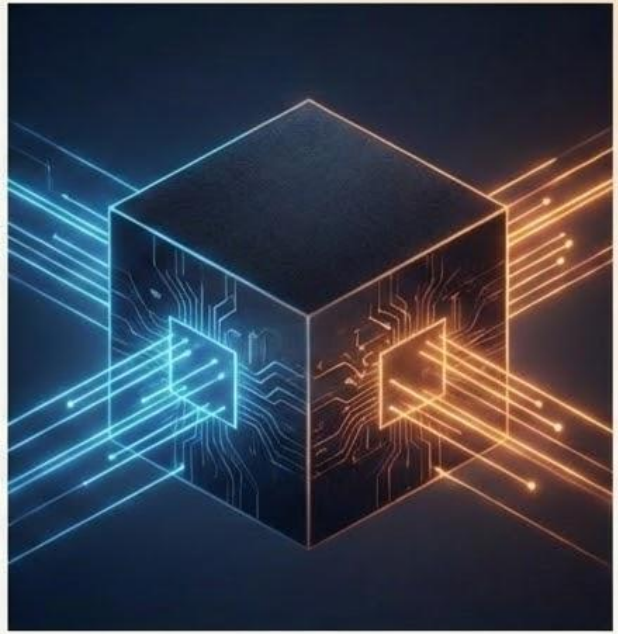


Context: Introduction - Transitioning from internal code structure to the external user experience.

The Black-Box Perspective

What If We Don’t Have the Code?

- **The Shift**: In our last lecture, we looked directly inside the code using white-box testing. But what happens when we can’t see the code, or when we need to test strictly from the end-user’s perspective?
- **The Definition**: That is black-box testing. We treat the software as a completely closed system.
- **The Goal:** We are no longer testing the implementation or the internal logic. We are testing strictly against the requirements to ensure the software actually does what it is supposed to do.



Learning Objectives

What You Will Master Today

By the end of this lecture, you will be equipped to:

- **Apply Core Techniques:** Utilize foundational black-box methods, including graph-based testing, equivalence partitioning, boundary value analysis, and orthogonal array testing.
- **Understand Models:** Explain model-based testing and identify its key benefits in the development lifecycle.
- **Navigate Specialized Environments**: Describe specific testing approaches for complex setups like graphical user interfaces (GUIs), client-server architectures, and real-time systems.
- **Recognize Patterns**: Understand the concept of testing patterns and how to apply them to recurring problems.

Part 1: Black-Box Testing Fundamentals

Testing from the Outside In

Behavioral, Functional, and Specification-Based Testing

Part 1 - Understanding how to test software based strictly on what it is supposed to do.

What is Black-Box Testing?

Focusing on Behavior, Not Implementation

- **The Core Focus:** We care exclusively about what the software does, entirely ignoring how it actually does it.
- **The Source of Truth:** Test cases are derived directly from the requirements specifications, not the source code.
- **The Primary Purpose:** To uncover specific categories of errors, including:
  - Incorrect or missing functions.
  - Interface and UI errors.
  - Performance bottlenecks.
  - Initialization and termination failures.

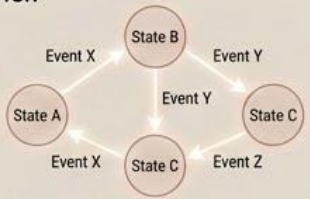
Graph-Based Testing Methods

Mapping the System

The Premise: We model the software as a visual graph of objects and their relationships, then design tests that navigate through this graph.

Cause-Effect Graphing:
Models the relationships between specific inputs (causes) and expected outputs (effects).

State Transition Graphs:
Models the various states of a system and the specific events that trigger a transition from one state to another.



Data Flow Graphs: Models how data objects are transformed as they move through the system.

Timing Graphs: Used primarily for real-time systems to verify strict timing constraints.

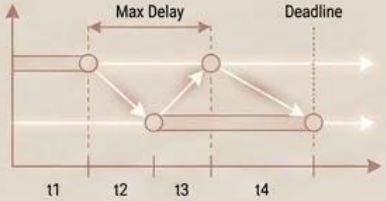
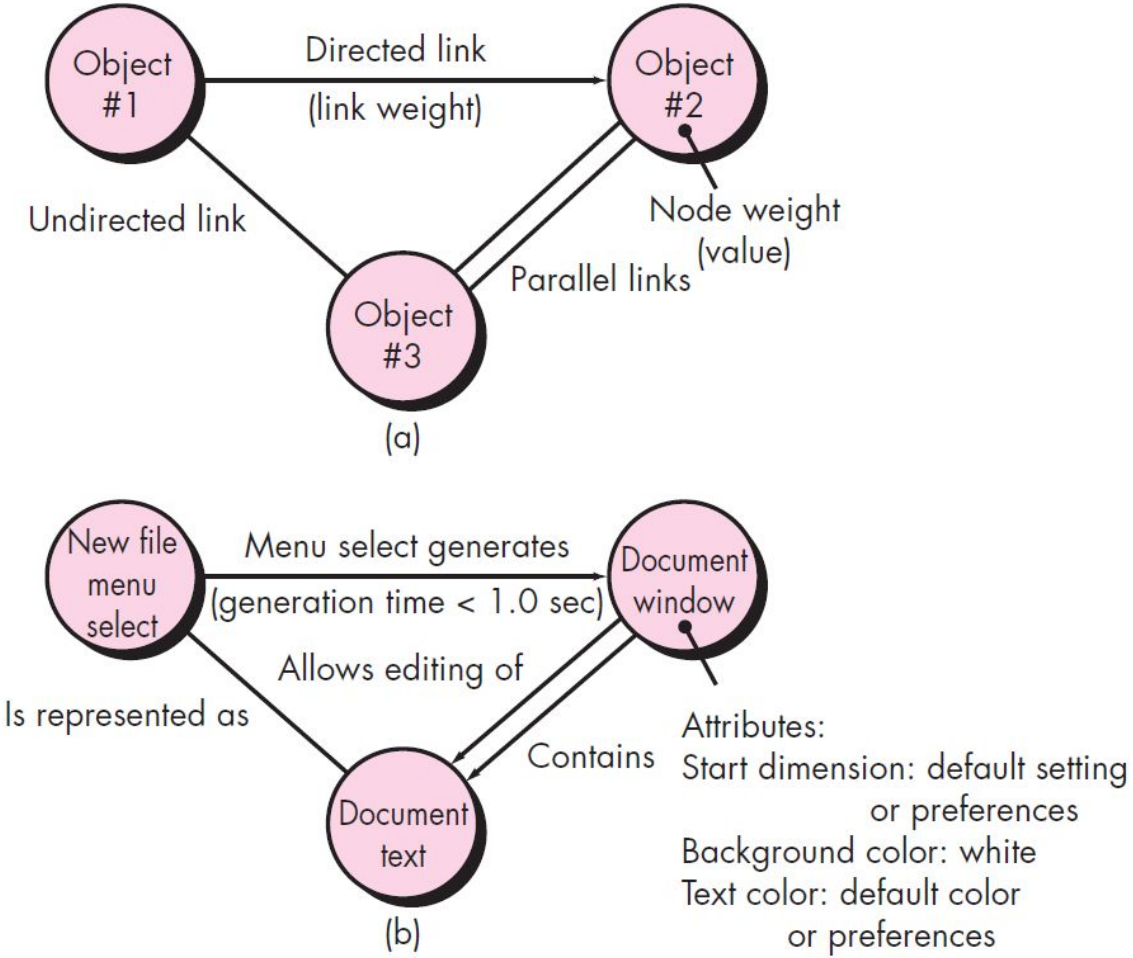


FIGURE 18.8

(a) Graph notation; (b) simple example

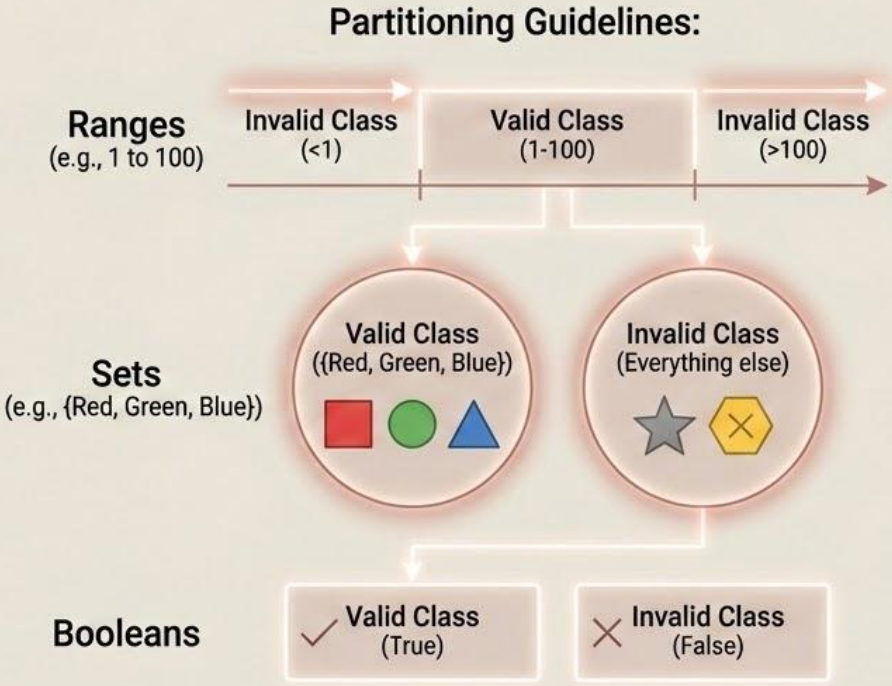


Equivalence Partitioning

Testing Smarter, Not Harder

The Core Idea: Divide the entire input domain into distinct "classes" of data that the software will likely process in the exact same way. You then only need to test one representative value from each class.

The Assumption: If a test case in a specific class uncovers an error, all other cases in that class will likely trigger the same error (and vice versa).

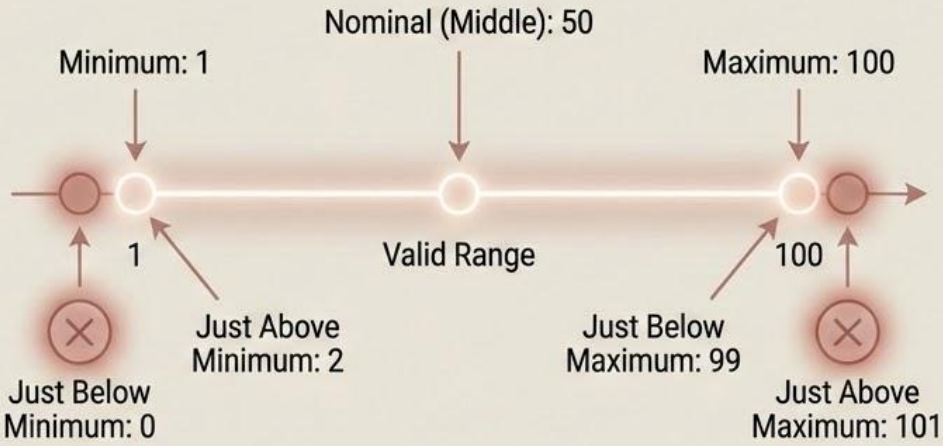


Boundary Value Analysis (BVA)

Hunting Bugs at the Edges

- **The Core Idea**: Experience shows that software errors rarely happen in the middle of an input domain; they cluster at the absolute boundaries.
- **The Approach**: An extension of Equivalence Partitioning. For every partition, we explicitly test the absolute edges.

What to Test (Example: Input range 1-100):



(Note: Always apply BVA to output boundaries as well, not just inputs!)

Orthogonal Array Testing

Solving the Combinatorial Explosion

The Problem:

When testing a system with multiple input parameters, testing every single possible combination is often mathematically impossible due to combinatorial explosion.

The Solution:

Use an **Orthogonal Array** to select a mathematically "balanced" subset of combinations.

Full Factorial
(Explosion)

Orthogonal Array
(Balanced)

Example (3 Parameters, 2 Values Each):

Instead of testing all 8 possible combinations (2^3), an L4 array tests just 4 cases while still guaranteeing coverage of all pairwise combinations.

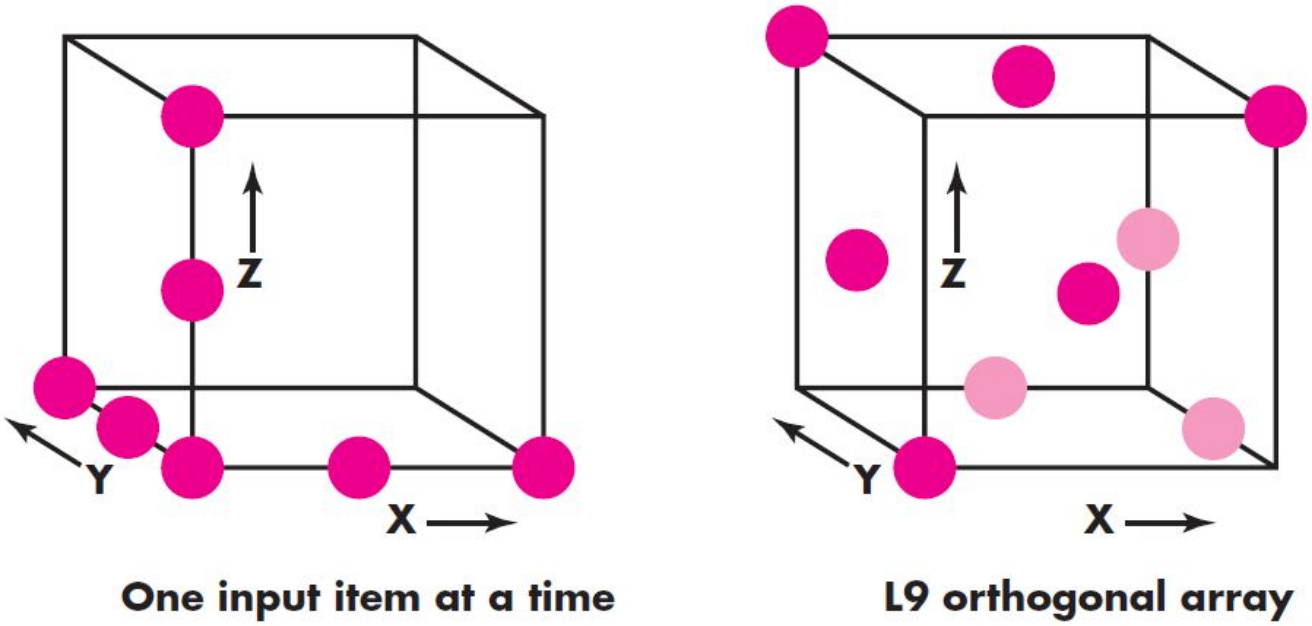
Test Case	Parameter A	Parameter B	Parameter C
1	A1	B1	C1
2	A1	B2	C2
3	A2	B1	C2
4	A2	B2	C1

The Advantage:

Drastically reduces the number of required test cases while maintaining robust interaction coverage. Perfect for GUI and configuration testing.

FIGURE 18.9

A geometric view of test cases
Source: [Pha97]



One input item at a time

L9 orthogonal array

FIGURE 18.10

An L9 orthogonal array

Test case	Test parameters			
	P1	P2	P3	P4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

In-Class Activity

Applying the Techniques

The Scenario:

You are testing a function for a new software system that accepts two inputs:

Age:

Must be an integer between 0 and 120.

Membership Type:

Must be "Basic", "Premium", or "Gold".

Your Task:

Take 5 minutes to apply both Equivalence Partitioning and Boundary Value Analysis to these inputs. List out the specific test cases you would run.

Part 2: Model-Based Testing

From Abstraction to Execution

Generating Tests from System Behavior Models

SYSTEM BEHAVIOR MODEL
ABSTRACTION

TEST EXECUTION
AUTOMATION

Part 2 - Bridging the gap between system design and test execution.

Slide 2: Defining Model-Based Testing

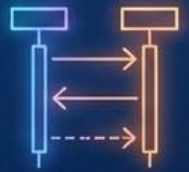
Testing the Blueprint

The Definition:

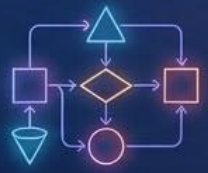
Testing where the test cases are not written from scratch, but are systematically derived from a model that describes the expected, ideal behavior of the system.

Common Models Used:

State Machines:
(The most common approach)
Tracking how a system moves from one state to another based on events.



Sequence Diagrams:
Mapping the chronological flow of messages and interactions.



Activity Diagrams:
Visualizing workflows, decision points, and parallel processing.

Decision Tables:
Charting complex business rules and logic combinations.

Formal Specifications:
Mathematically rigorous models (e.g., Z, VDM).

Slide 3: The MBT Process

From Theory to Test Case

1. Build the Model

Create a complete behavioral model of the system directly from the gathered requirements.

2. Specify Criteria

Determine your test selection criteria (e.g., mandating that the tests must "cover every possible state" or "cover every transition").

3. Generate Tests

Automatically (or manually) generate the actual test cases directly from the logic embedded in the model.

4. Execute & Compare

Run the generated tests on the real software and compare the actual behavior to what the model originally predicted.

Slide 4: Benefits and Challenges

Evaluating the Trade-offs

While highly systematic and powerful, MBT comes with distinct advantages and real-world hurdles:

The Benefits (Why we use it)

Automation:
Enables the automated generation of massive, comprehensive test suites.

Traceability:
Every generated test is inherently traceable back to the core requirements.

Dual Purpose:
The model is highly useful for both upfront system analysis and backend testing.

Early Detection:
Forces teams to find inconsistencies in requirements before coding even begins.

The Challenges (What to watch out for)

Complexity:
Building highly accurate models is difficult, requiring specialized skills.

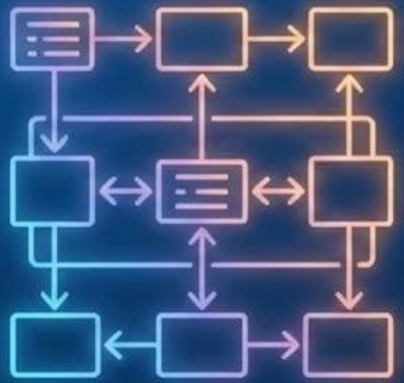
Model Errors:
If the blueprint model has an error, the generated tests will validate that error.

State Explosion:
Complex systems can generate an impossibly large number of state combinations.

Slide 1: Part 3: Testing Specialized Environments

Adapting to the Environment

Architectures, Interfaces, and Real-Time Systems



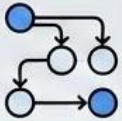
Context: Part 3 - How to adjust black-box strategies when the testing environment becomes complex.

Testing GUIs (Graphical User Interfaces)

Taming the Combinatorial Explosion

The Challenge

GUIs are heavily event-driven. Because users can click, scroll, and type in almost any order, the number of possible event sequences creates a combinatorial explosion. You also have to account for visual layout and platform/browser dependencies.



Event-Based Testing

Model the GUI as a series of states and events, often using orthogonal arrays to test balanced sequences of user actions.

Capture/Playback Tools

Use automated tools (like Selenium or QTP) to record a human's interactions and flawlessly replay them for regression testing.

Visual Testing

Utilize automated scripts to compare actual screenshots pixel-by-pixel against expected baseline images.

Usability Testing

Bring in actual humans to evaluate the effectiveness and intuition of the interface.

Slide 3: Testing Client-Server Architectures

Navigating Distributed Failure Modes

Client-Server setups introduce entirely new variables that a standalone application never faces.

Client-Side Testing

Validating the local GUI, local functionality, and how effectively the client caches data.

Server-Side Testing

Pushing the backend through heavy load testing, stress testing, and scalability benchmarks.

Network Testing

Actively simulating network chaos: packet loss, high latency, and random disconnections.

Transaction Testing

Ensuring the database maintains strict ACID properties (Atomicity, Consistency, Isolation, Durability) across distributed clients.

Slide 4: Testing Documentation & Help Facilities

The Overlooked Deliverable

The Reality:

Documentation is often an afterthought, but it is absolutely critical for system usability and reducing support tickets. It must be treated as a primary software deliverable.

What to Test Systematically:

Accuracy:

Does the documentation actually match what the software currently does?

Completeness:

Is every single feature and edge case documented?

Usability:

Can a frustrated user easily search for and find what they need?

Links:

Are there any dead hyperlinks in the help files?

Examples:

If you provide step-by-step examples or code snippets, do they actually work if a user copies them?

Slide 5: Testing Real-Time Systems

When "Late" Means "Wrong"

The Core Concept: In a real-time system, a logically correct answer delivered past the deadline is considered a complete system failure. Correctness depends entirely on time.

The Challenges: Strict deadlines, asynchronous concurrency, interrupt handling, and heavy hardware dependencies.

Testing Approaches:

Task Timing Tests: Measuring the exact execution time of tasks under varying system loads.

Synchronization Tests: Hunting for race conditions in semaphores, monitors, and message passing.

Interrupt Tests: Intentionally triggering system interrupts at the most critical moments of execution.

Deadline Tests: Verifying that absolutely all tasks meet their strict timing deadlines.

Hardware-in-the-Loop (HIL): Bypassing software simulators to test the code against the actual physical hardware it will control.

Slide 1: Part 4: Patterns for Software Testing

Standardizing the Approach

Reusing Proven Solutions for Recurring Problems

Context: Part 4 - How to apply established industry knowledge to speed up the testing process.

What Are Testing Patterns?

Capturing Expert Knowledge

The Concept:

Just as software design patterns capture widely accepted solutions to common architectural problems, testing patterns capture proven, repeatable solutions to recurring testing problems.

The Goal:

Reinventing the Wheel

Applying a Recognized Pattern

Instead of reinventing the wheel every time a new feature needs to be tested, QA engineers can apply a recognized testing pattern to immediately deploy a reliable strategy.

Categories of Testing Patterns

Organizing the Solutions

Test Case Design

Example Pattern:
Boundary Tester

The Problem

How to find errors at input limits?

The Solution

Apply Boundary Value Analysis.

Test Strategy

Example Pattern:
Crash Early Tester

The Problem

How to find serious defects quickly?

The Solution

Focus testing efforts strictly on high-risk areas first.

Test Execution

Example Pattern:
Record and Playback

The Problem

How to automate GUI regression testing?

The Solution

Utilize capture/playback automation tools.

Test Organization

Example Pattern:
Independent Test Team

The Problem

How to ensure strict testing objectivity?

The Solution

Create a separate, isolated test group away from the developers.

The Benefits of Testing Patterns

Why We Use Them

Captures Expert Knowledge

Institutionalizes the tactics used by senior QA engineers so junior members can easily apply them.

Provides a Common Vocabulary

Allows the team to communicate complex strategies quickly (e.g., "Let's use a Boundary Tester approach here").

Accelerates Test Design

Speeds up the planning phase because the foundational strategy is already defined.

Improves Test Quality

Ensures comprehensive coverage by relying on proven, industry-standard methodologies.

Pattern in Action: “Four-Eyed Tester”

A Real-World Example

Pattern Name: “Four-Eyed Tester”

The Context

The team is dealing with highly complex business rules that require strict validation.

The Problem

A single tester reviewing the system alone is highly likely to miss subtle, cascading errors due to fatigue or oversight.

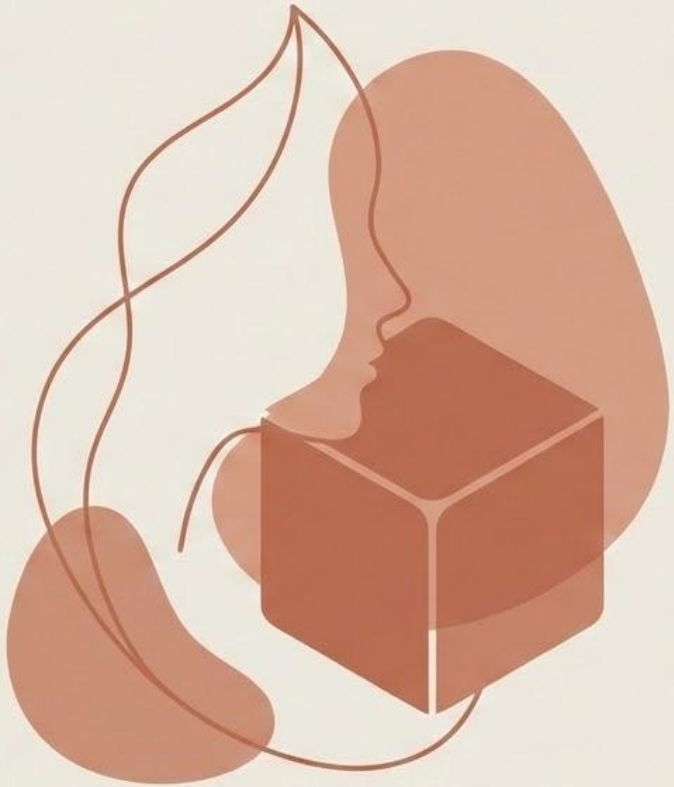
The Solution

Implement pair testing—two human testers work together simultaneously on the exact same test cases to ensure nothing slips through the cracks.

Part 5 - Wrapping up the core concepts and methodologies.

The User's Advocate

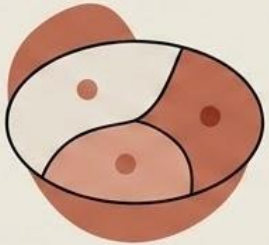
Synthesizing Black-Box Testing Strategies



Core – Core Black-Box Techniques

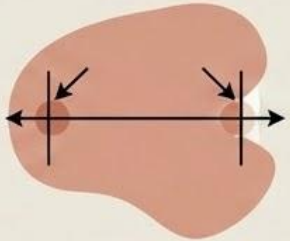
Testing the Requirements, Not the Code

Black-box testing strictly derives test cases from what the software is supposed to do. The foundational techniques include:



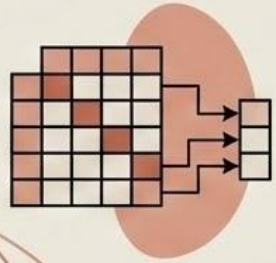
Equivalence Partitioning

Dividing the entire input domain into distinct classes and testing just one representative value per class.



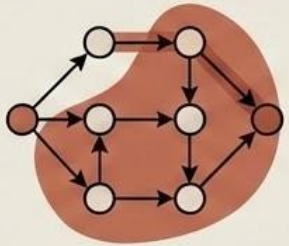
Boundary Value Analysis

Rigorously testing at the extreme edges of those partitions where errors are statistically most common.



Orthogonal Array Testing

Efficiently testing complex input combinations using mathematically balanced arrays to avoid combinatorial explosion.

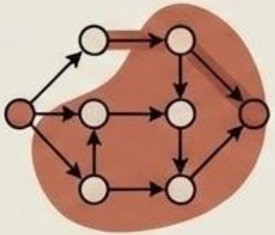


Graph-Based Methods

Modeling the system as a visual graph to systematically test logical paths, causes, and effects.

Models and Specialized Environments

Adapting to Complexity



Model-Based Testing

Automates the generation of test cases directly from formal behavioral models (like state machines and activity diagrams), creating a direct link between requirements and tests.

Documentation

Systematically testing help files and manuals as primary software deliverables.

Specialized Environments

GUIs  Event-based testing sequences and automated capture/playback tools.	**Client-Server**  Network latency chaos, heavy load capacity, and transaction integrity.
Documentation 	**Real-Time Systems** Strict timing constraints, process synchronization, and hard deadline testing.

Testing Patterns and Final Thought

Working Smarter, Not Just Harder

Testing Patterns

By capturing and reusing proven solutions to common testing problems, we drastically improve both our team's efficiency and the overall quality of the tests.

> "Black-box testing is the user's advocate—it ensures the software does what it promises, regardless of how it's built. Combined with white-box testing, we achieve both structural integrity and functional correctness."