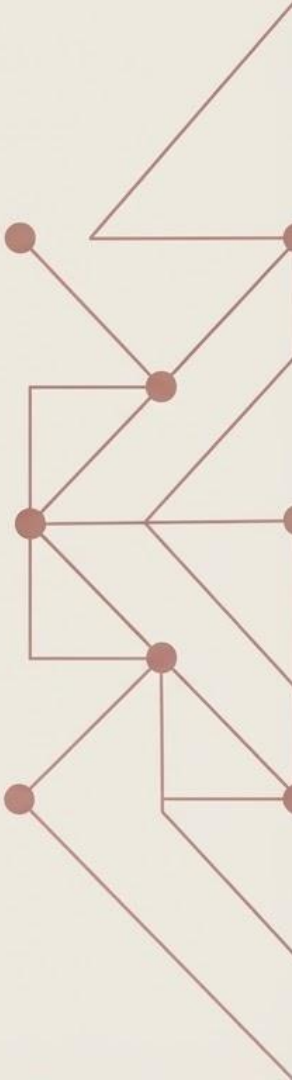
# Chapter 17: Software Testing Strategies (Part 1)

Subject:
Software Engineering

Program:
BTech Computer Science and Engineering

Duration:
1 Hour



Slide 1: Introduction to Software Testing Strategies

# Does It Actually Work?

## The Transition from Static Quality to Dynamic Verification

Lecture Introduction & Objectives

Slide 2: The Opening Hook

# Testing is Not Random Button-Pushing

**The Journey So Far**: We have designed the architecture, subjected the code to peer reviews, and built the formal quality assurance plans.

**The Big Question**: Now we must ask the most critical question of all: Does the software actually work?

**Dynamic Verification**: Testing is the active, dynamic verification that the software behaves exactly as expected under real-world conditions.

**The Strategic Mindset**: Testing is not just clicking buttons until something breaks. It forces us to answer complex strategic questions:

- How do we properly organize a testing phase?
- How do we test individual pieces and safely put them together?
- How do we mathematically know when we can stop testing?

# Lecture Objectives

What We Will Cover Today
By the end of this lecture, you will be able to:

- Define the Terms: Define both verification and validation, and clearly distinguish between the two concepts.
- Build the Strategy: Describe a formal, strategic approach to software testing, including how to organize teams and define strict completion criteria.
- Identify the Roadblocks: Identify the key strategic issues and pitfalls that heavily influence the success or failure of a testing phase.
- Execute the Tactics: Explain the core mechanics of unit testing and integration testing strategies for conventional software development.

# The Blueprint for Verification

## Organizing and Executing a Software Testing Strategy

Part 1 - Moving from ad-hoc debugging to a formal, strategic testing lifecycle.

# Verification vs. Validation (V&V)

## Understanding the Difference

Before we test, we must understand what we are actually measuring.

### Verification 

**The Core Question**
"Are we building the product right?"

**Focus & Activities**
Ensures the software correctly implements specified functions (consistency, completeness).
**Activities**: Reviews, walkthroughs, static analysis.

### Validation 

**The Core Question**
"Are we building the right product?"

**Focus & Activities**
Ensures the software actually meets customer expectations and real-world needs.
**Activities**: Acceptance testing, prototyping, user feedback.

**The Relationship**
Verification ensures the product is built correctly according to the blueprint; validation ensures that blueprint is actually what the customer wanted. Both are essential.

# Organizing for Software Testing

## The Independence Spectrum

Who should actually write and execute the tests?

### Level 1
(Lowest Independence)


Tests designed by the **person who wrote the code**.

- **Pro:** Knows internals deeply.
- **Con:** Blind to their own mistakes.

### Level 2


Tests designed by a **peer** (another developer on the same team).

### Level 3


Tests designed by a **dedicated, internal testing/QA team**.

### Level 4
(Highest Independence)


Tests designed by an **external organization** or **independent consultants**.

**The Principle:** Greater independence leads to more objective, unbiased testing, but may sacrifice deep, code-level technical knowledge.

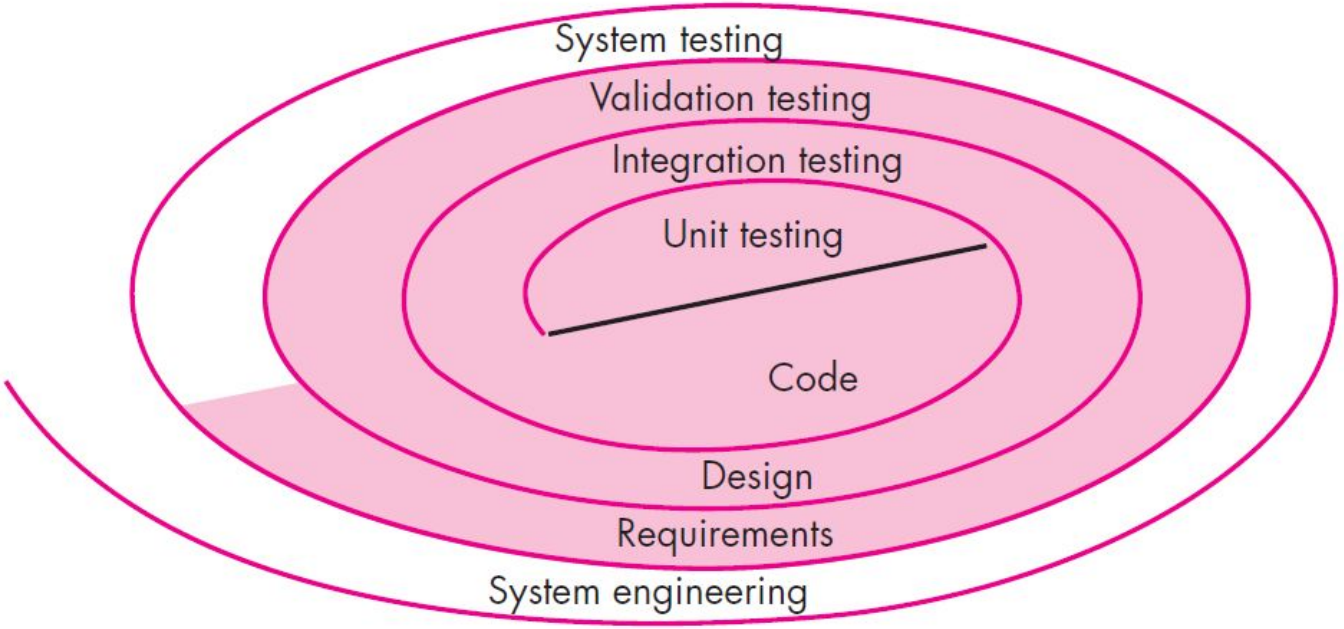

**Typical Structure**

Developers → Unit Testing

Separate QA Team → Integration, System, and Acceptance Testing.

**FIGURE 17.1**

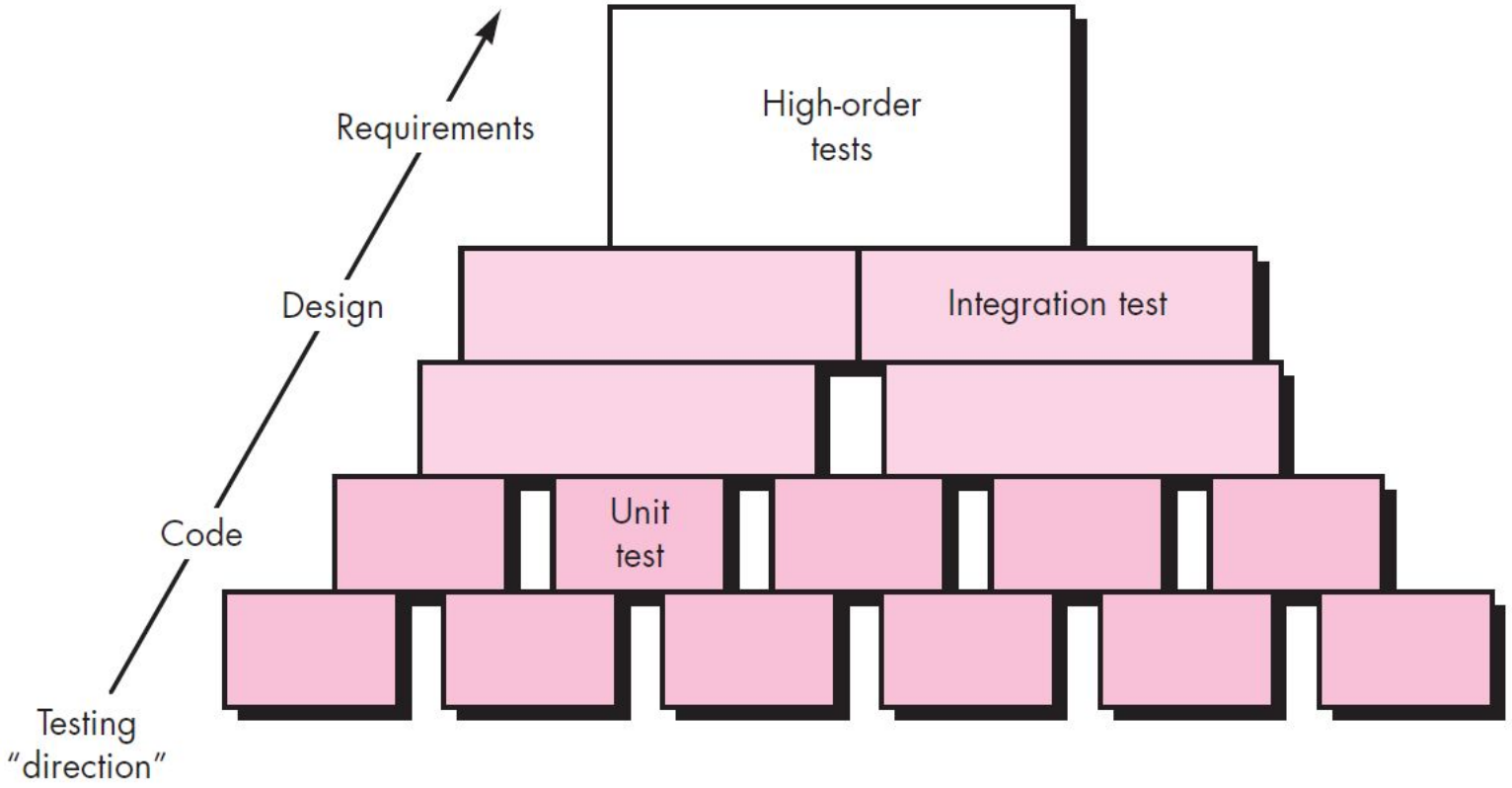
Testing strategy



A strategy for software testing may also be viewed in the context of the spiral (Figure 17.1). *Unit testing* begins at the vortex of the spiral and concentrates on each unit (e.g., component, class, or WebApp content object) of the software as implemented in source code. Testing progresses by moving outward along the spiral to *integration testing,* where the focus is on design and the construction of the software architecture. Taking another turn outward on the spiral, you encounter *validation testing,* where requirements established as part of requirements modeling are validated against the software that has been constructed. Finally, you arrive at *system testing,* where the software and other system elements are tested as a whole. To test computer software, you spiral out in a clockwise direction along streamlines that broaden the scope of testing with each turn.

**FIGURE 17.2**

**Software testing steps**

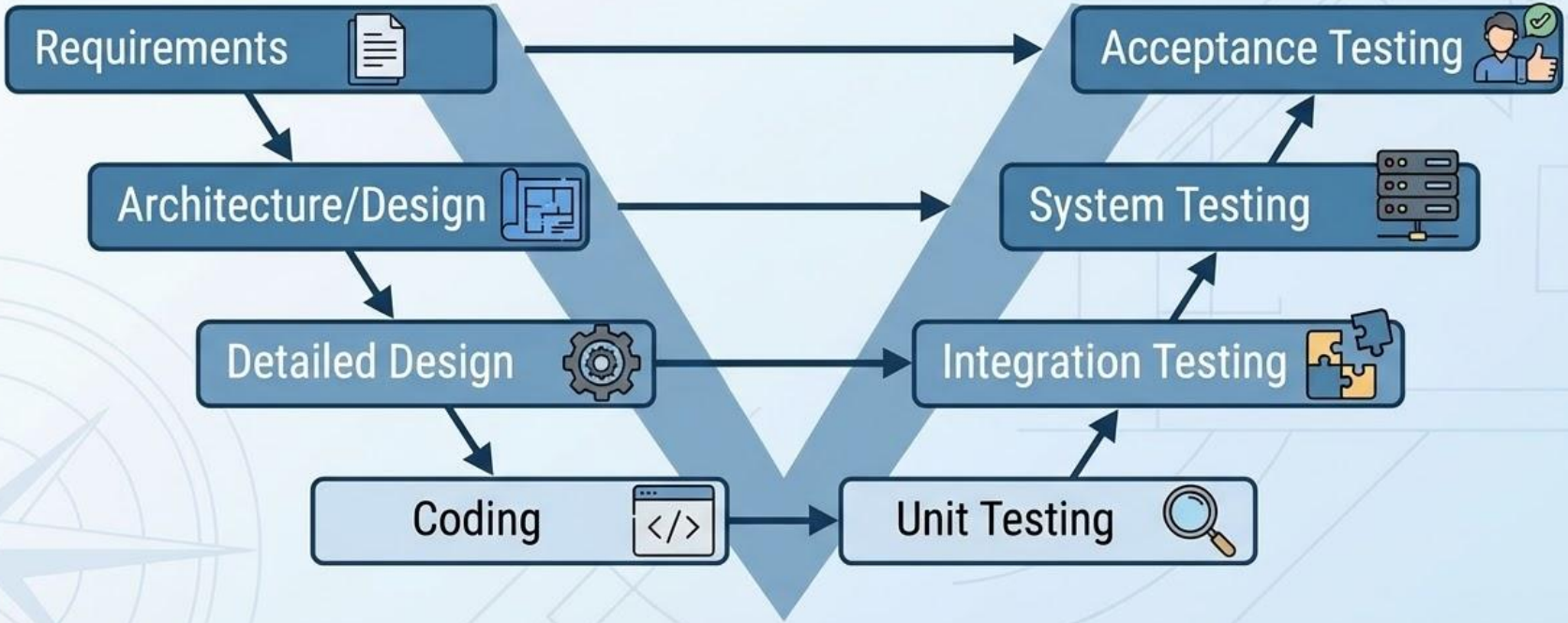


component individually, ensuring that it functions properly as a unit. Hence, the name *unit testing*. Unit testing makes heavy use of testing techniques that exercise specific paths in a component's control structure to ensure complete coverage and maximum error detection. Next, components must be assembled or integrated to form the complete software package. *Integration testing* addresses the issues associated with the dual problems of verification and program construction. Test case design techniques that focus on inputs and outputs are more prevalent during integration, although techniques that exercise specific program paths may be used to ensure coverage of major control paths. After the software has been integrated (constructed), a set of *high-order tests* is conducted. Validation criteria (established during requirements analysis) must be evaluated. *Validation testing* provides final assurance that software meets all informational, functional, behavioral, and performance requirements.

# The Big Picture: The ‘V’ Model

## Testing as a Parallel Process

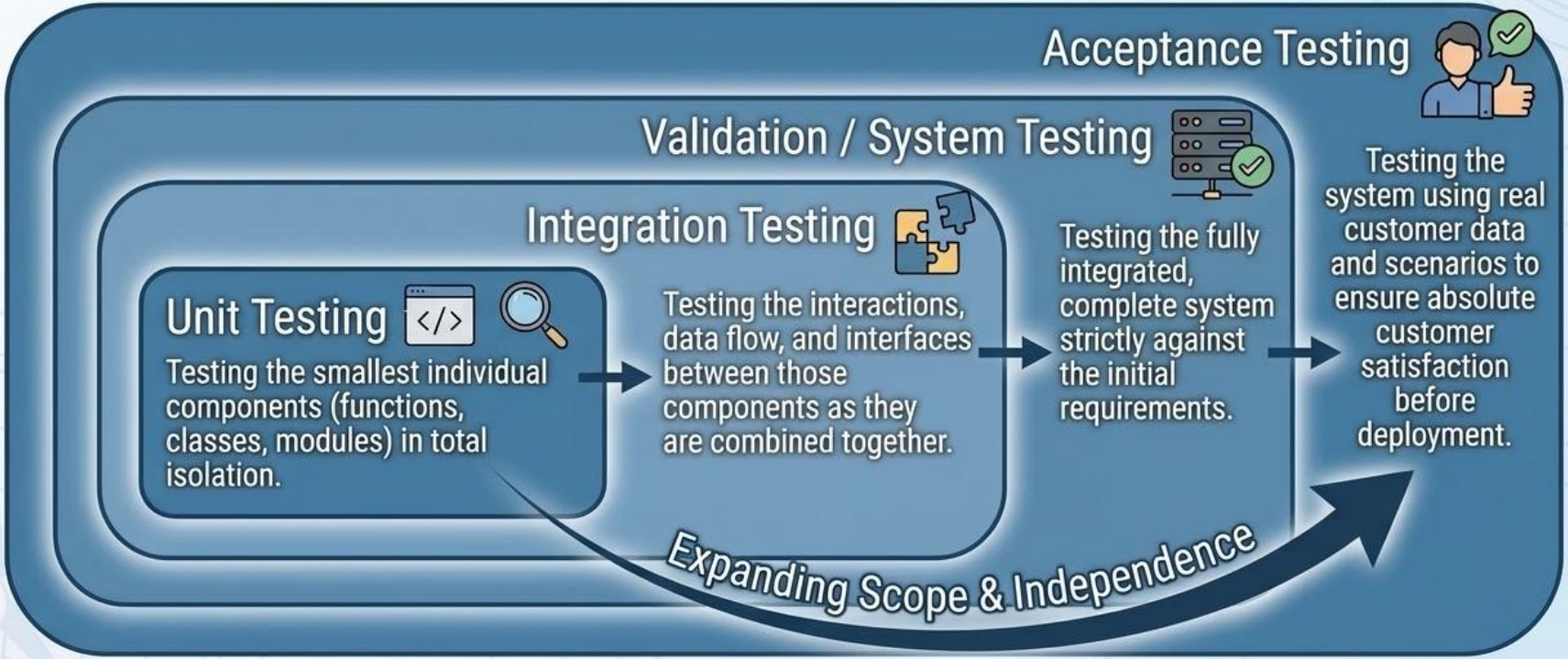
Testing is not a single phase tacked onto the end of a project. It is a series of steps that mirror the development lifecycle.



# The Strategic Flow of Testing

## Moving from the Inside Out

Following the V-Model, testing expands outward in a specific sequence:



# Criteria for Completion

## When Do We Stop Testing?

**The Dilemma:** Exhaustive testing is mathematically impossible (there are infinite input combinations). So, how do we know when we are done?

### Time-Based

Testing stops when the scheduled time or budget runs out. (Highly Risky!)

### Coverage-Based

Testing stops when a defined metric is hit (e.g., 90% statement coverage, or 100% of critical functions functions tested).

### Defect-Based

Testing stops when the defect discovery rate falls below a specific threshold (e.g., fewer than 1 new defect found per week).

### Reliability-Based

Testing stops when calculated reliability metrics (like MTTF) meet the target goals.

### Risk-Based

Testing stops when the remaining known risks are deemed acceptable by management.

**Best Practice:** Use a strategic combination of these criteria, and always define them in the test plan before testing actually begins.

Part 2: Strategic Issues

# Setting the Stage for Success

## The Strategic Factors That Make or Break a Testing Phase

Context: Part 2 - Moving beyond the “how-to” and focusing on the overarching strategy of testing.

# The Foundation of the Strategy

## Clarity and Context

A successful testing strategy begins long before the first line of code is evaluated.

### Specify Quantifiable Requirements

Testing requires clear, testable, and strictly measurable requirements. Vague requirements inevitably lead to vague, ineffective tests.

### Specify Explicit Objectives

Clearly define exactly what each testing phase aims to achieve. (e.g., "Unit testing will verify each module strictly against its interface specification," rather than just "Unit testing will find bugs.")

### Understand the Users

Different users have entirely different usage patterns. Your testing scenarios must accurately reflect real-world usage profiles, not just perfect-world assumptions.

# Process and Architecture

## Building Testing into the DNA of the Project

How we build and iterate dictates how effectively we can test.

### Emphasize Rapid Cycle Testing

Do not wait for a single, massive test phase at the very end of the project. Implement iterative testing with rapid feedback loops—"test a little, code a little"—to catch errors immediately.

### Build Self-Testing Software

Design the architecture with built-in testing capabilities from day one. Include automated assertion checking, comprehensive error logging, and dedicated diagnostic interfaces to make the software easier to verify.

# Quality Checks and Evolution

## Reviewing the Reviewers

Testing is not the only quality gate; it is part of a larger ecosystem.

### Use Technical Reviews as a Filter

Formal peer reviews catch a massive percentage of defects before testing even begins, making the actual testing phase significantly cheaper and more efficient.

### Review the Tests Themselves

Conduct formal technical reviews of the test strategy and the individual test cases. A bad test is worse than no test at all.

### Continuous Improvement

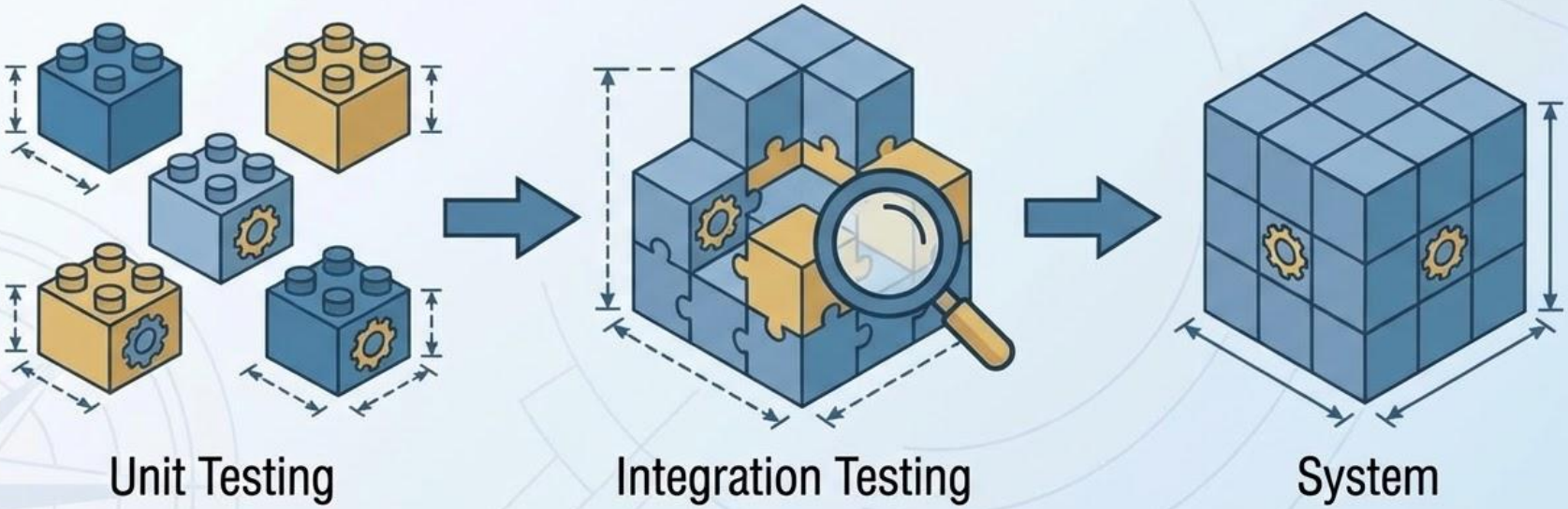
Treat testing as an evolving discipline. Actively measure your test effectiveness (e.g., tracking defect escape rates) and refine the process over time.

# Part 3: Test Strategies for Conventional Software

## Building from the Ground Up

### Mastering Unit and Integration Testing

Context: Part 3 - How we methodically test conventional (structured or object-oriented) software from the smallest component to the fully combined system.



# Unit Testing (The First Line of Defense)

## Isolating the Smallest Pieces

### Definition & “Unit”

The Definition: Testing individual software components in absolute isolation.

What is a “Unit”? It is the smallest testable piece of software (e.g., a single function, a procedure, a method, or a class).

### Who & When

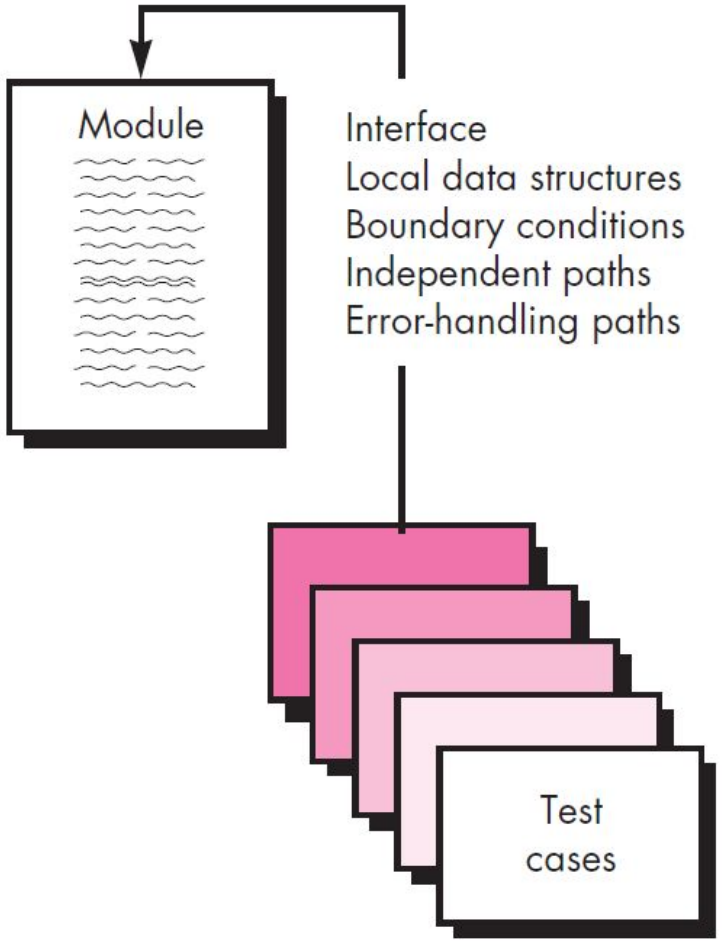
Typically performed by the specific developer who wrote the code, either during coding or immediately after (but before integration).

### What is Actually Tested?

- Module Interface: Does data flow correctly into and out of this specific unit?
- Local Data Structures: Does the unit maintain its internal data integrity?
- Boundary Conditions: Does it handle extreme or unexpected values correctly?
- Independent Paths: Are all logical paths through the code actually exercised?
- Error Handling: Does the unit handle errors gracefully without crashing?

**FIGURE 17.3**

Unit test



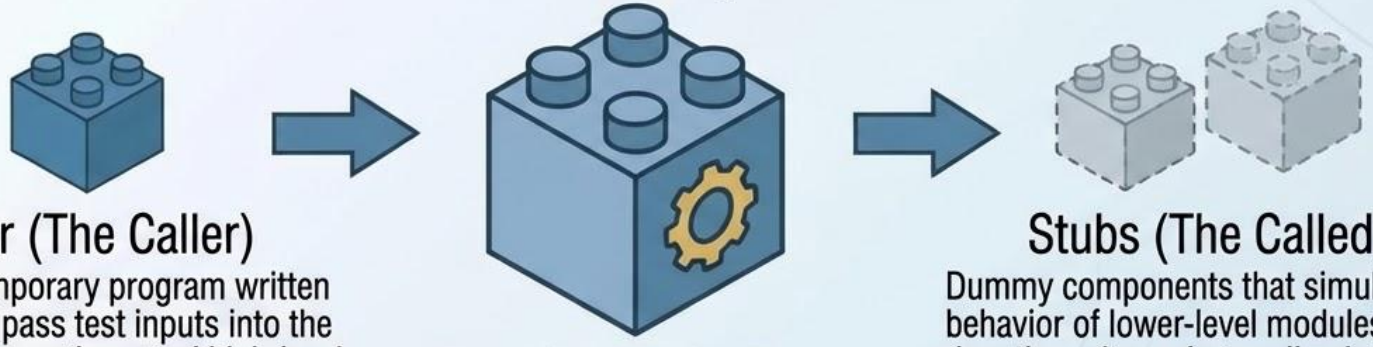
**Unit-test considerations.** Unit tests are illustrated schematically in Figure 17.3. The module interface is tested to ensure that information properly flows into and out of the program unit under test. Local data structures are examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution. All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once. Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing. And finally, all error-handling paths are tested.

# The Unit Testing Environment

## Simulating the Missing Pieces

To test a single unit in isolation, we have to fake the rest of the system using specialized test components.

### The Setup



**Driver (The Caller)**

A simple, temporary program written specifically to pass test inputs into the unit. Used because the actual high-level calling program isn't integrated yet.

**The Unit Under Test**

**Stubs (The Called)**

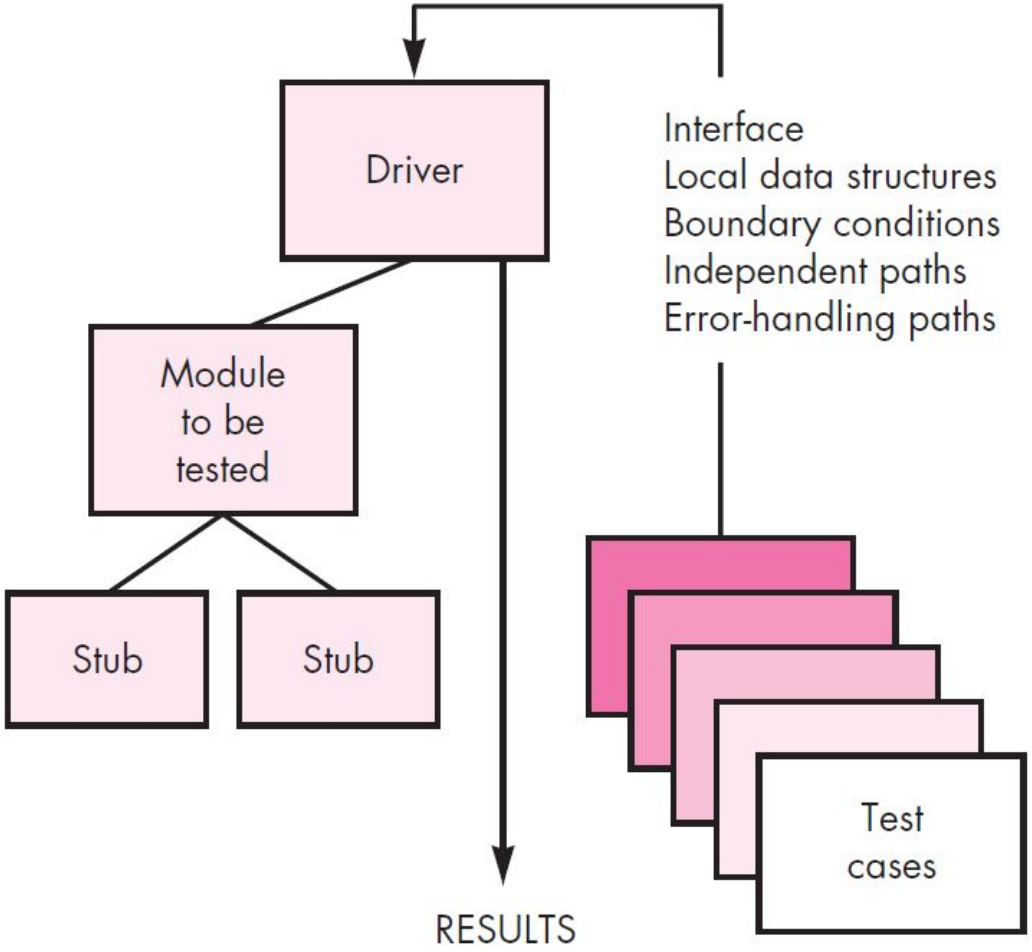
Dummy components that simulate the behavior of lower-level modules. Used when the unit needs to call a database or utility function that isn't available yet.

### The Method

Unit testing is highly White-Box intensive (focused on internal code structure), but it also includes basic black-box testing of the module's inputs and outputs.

**FIGURE 17.4**

Unit-test environment



Because a component is not a stand-alone program, driver and/or stub software must often be developed for each unit test. The unit test environment is illustrated in Figure 17.4. In most applications a *driver* is nothing more than a "main program" that accepts test case data, passes such data to the component (to be tested), and prints relevant results. *Stubs* serve to replace modules that are subordinate (invoked by) the component to be tested. A stub or "dummy subprogram" uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.

# Integration Testing

## The Chaos of Combination

**The Definition:**
Testing the interaction, data flow, and interfaces between newly integrated components.

**The Core Problem:**
Why do units that pass 100% of their unit tests suddenly fail when put together?

- Interface Mismatches: Component A sends an integer, but Component B expects a string.

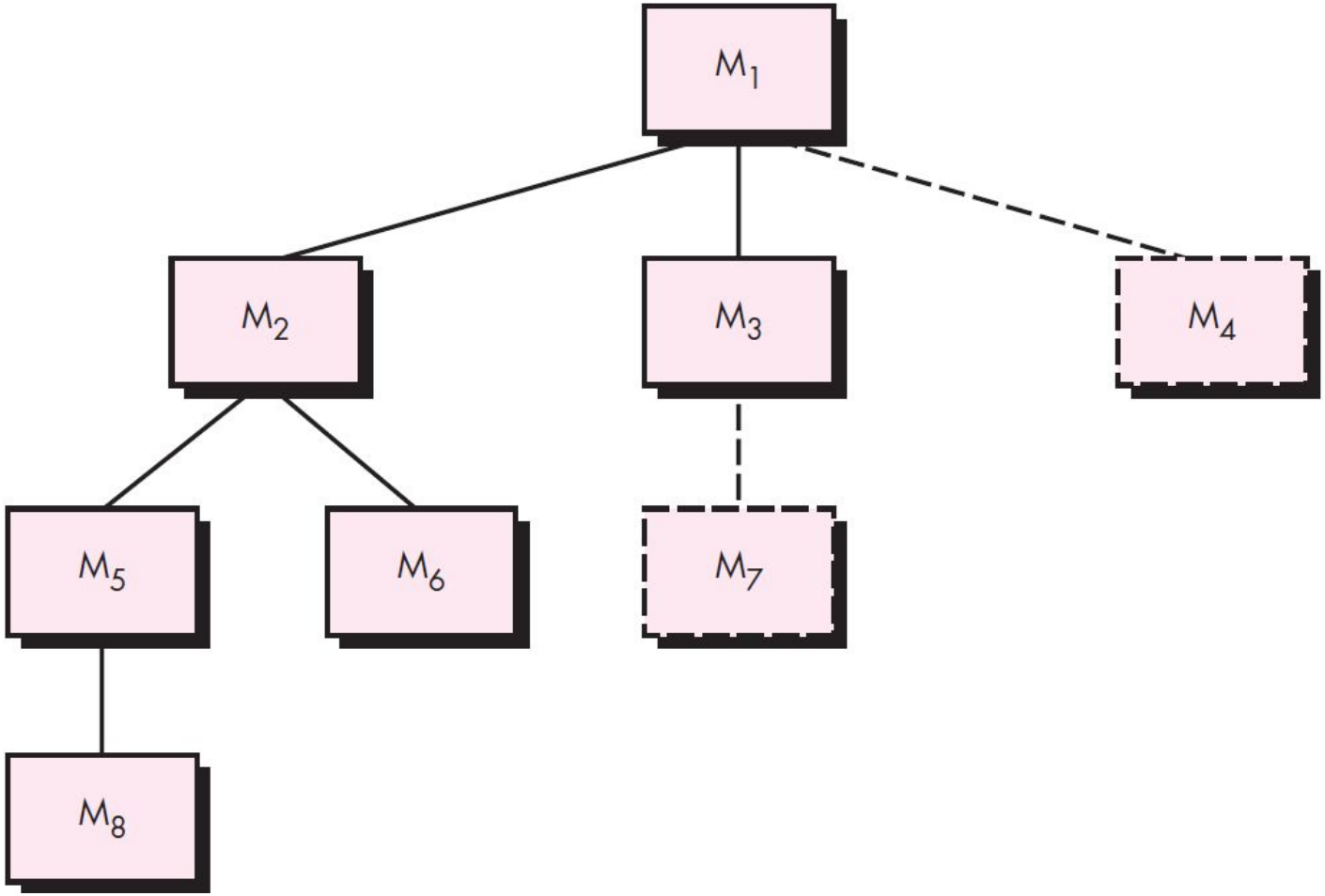
- Data Corruption: Global data is accidentally altered across module boundaries.

- Unexpected Side Effects: Fixing a bug in one module breaks another.

- Timing/Sequencing: Component B tries to run before Component A finishes fetching the data.

**FIGURE 17.5**

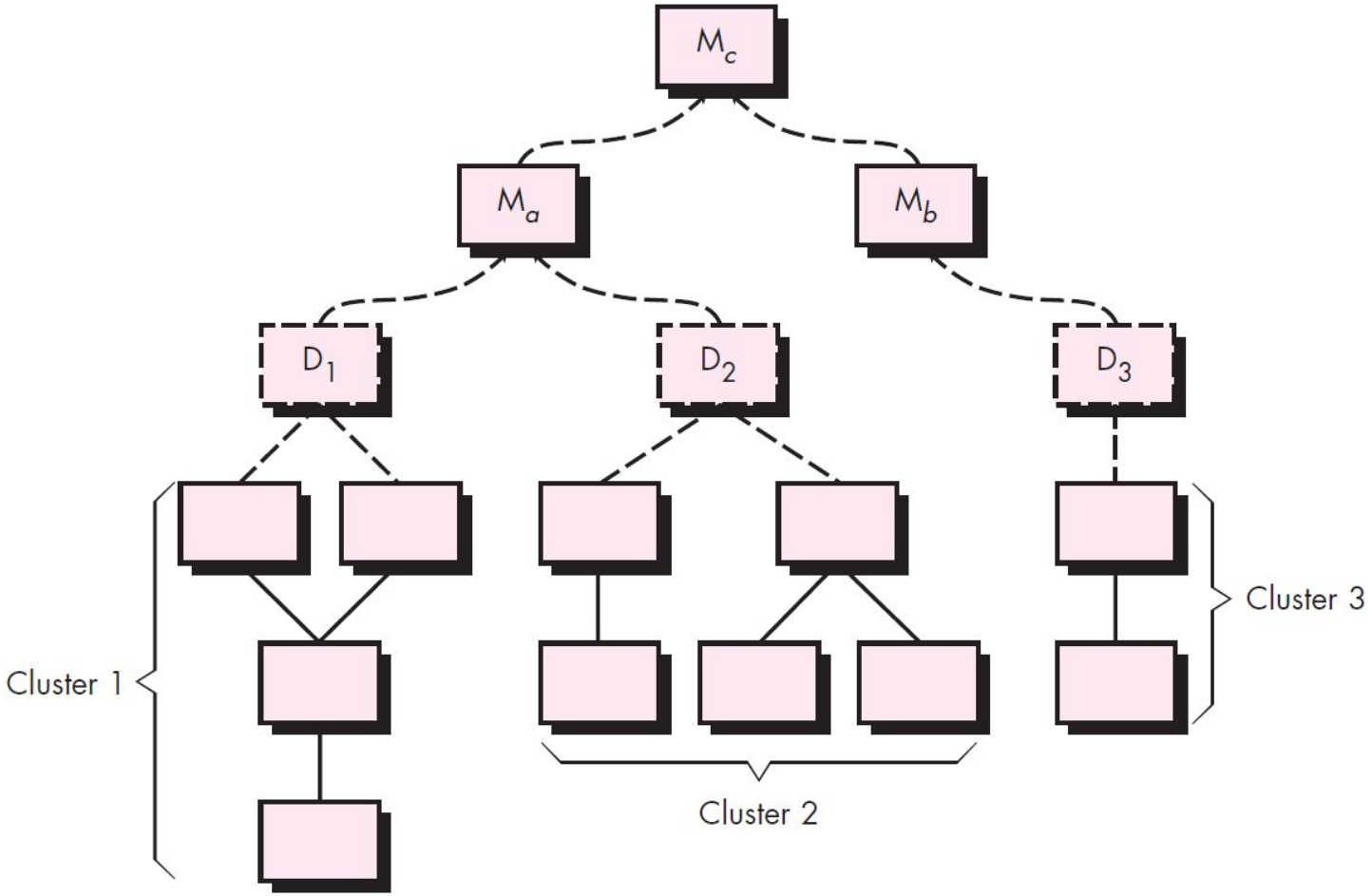
Top-down integration



Referring to Figure 17.5, *depth-first integration* integrates all components on a major control path of the program structure. Selection of a major path is somewhat arbitrary and depends on application-specific characteristics. For example, selecting the left-hand path, components $M_1$, $M_2$ , $M_5$ would be integrated first. Next, $M_8$ or (if necessary for proper functioning of $M_2$) $M_6$ would be integrated. Then, the central and right-hand control paths are built. *Breadth-first integration* incorporates all components directly subordinate at each level, moving across the structure horizontally. From the figure, components $M_2$, $M_3$, and $M_4$ would be integrated first. The next con-

**FIGURE 17.6**

Bottom-up integration



Integration follows the pattern illustrated in Figure 17.6. Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a driver (shown as a dashed block). Components in clusters 1 and 2 are subordinate to $M_a$. Drivers $D_1$ and $D_2$ are removed and the clusters are interfaced directly to $M_a$. Similarly, driver $D_3$ for cluster 3 is removed prior to integration with module $M_b$. Both $M_a$ and $M_b$ will ultimately be integrated with component $M_c$, and so forth.

# Integration Approaches

## Big Bang vs. Incremental

How exactly do we combine the pieces?

### Big Bang Integration

#### The Approach

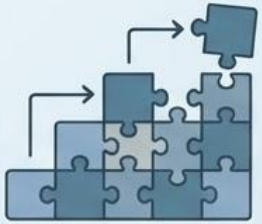
Throw all components together at once and test the complete system simultaneously.

#### The Trade-offs

- ✓ **Pros:** Simple; no need to write stubs or drivers.
- ✗ **Cons:** When it inevitably crashes, it is nearly impossible to isolate the defect. High risk and poor for debugging.

VS

### Incremental Integration



#### The Approach

Integrate and test components one at a time, in small, methodical steps.

#### The Trade-offs

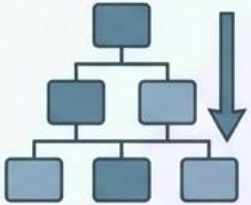
- ✓ **Pros:** Extremely easy to isolate defects (if it breaks, the newly added component is the culprit).
- ✗ **Cons:** Requires writing complex stubs and drivers.

# Directional Incremental Strategies

## Top-Down, Bottom-Up, and Sandwich

When using the Incremental approach, which direction do we build?

### Top-Down Integration



Start with the main control module (the top) and add subordinates one by one, using stubs for the lower levels.

- ✓ **Advantage:** Validates high-level control logic early.
- ✗ **Disadvantage:** Stubs can be highly complex to write.

### Bottom-Up Integration

Start with low-level utility modules and combine them into clusters, using drivers to invoke them.

- ✓ **Advantage:** Tests core functionality early; drivers are easier to write than stubs.
- ✗ **Disadvantage:** High-level system logic isn't tested until the very end.

### Sandwich (Hybrid) Integration

Combine Top-Down for high-level modules and Bottom-Up for low-level modules, meeting in the middle.

Balances the benefits of both, but is much more complex to manage.

# The Strategy of Quality

## Summary of Software Testing Strategies

Part 5 - Wrapping up the core concepts of dynamic verification.

# The Dual Mandate of Testing

## Verification vs. Validation

Testing exists to answer two fundamentally different, but equally important, questions:

### Verification (Process-Focused)

**“Are we building the product right?”**

Ensures the software strictly adheres to its design and technical specifications.

### Validation (Customer-Focused)

**“Are we building the right product?”**

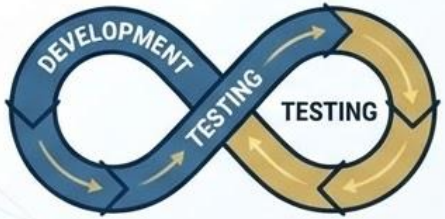
Ensures the software actually solves the user's problem and meets their real-world expectations.

# The Testing Lifecycle

## A Multi-Level Strategy

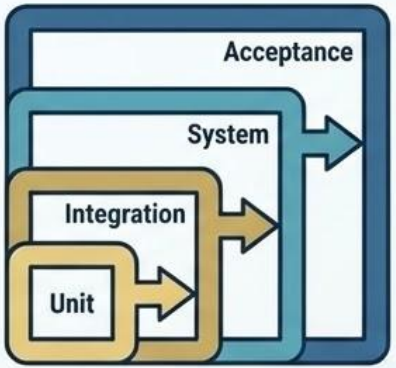
### Not a Single Phase:

Testing is a strategic, continuous activity that runs parallel to development.




Testing is a strategic, continuous activity that runs parallel to development.

### The Progression:




It must proceed methodically from the inside out: Unit → Integration → System → Acceptance.

### Knowing When to Stop:


You cannot test endlessly. Strict completion criteria (such as test coverage percentages, acceptable defect rates, or reliability metrics) must be defined before testing even begins.

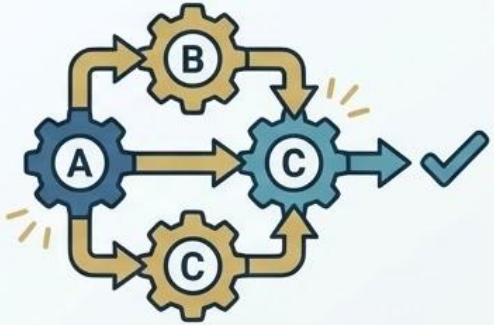
# The Core Mechanics

## From Units to Systems

### Unit Testing


Verifies individual components (like functions or classes) in total isolation, heavily relying on custom drivers and stubs to simulate the missing parts of the system.

### Integration Testing




Verifies the complex interactions and data **hand-offs** between those components.

### The Golden Rule of Integration

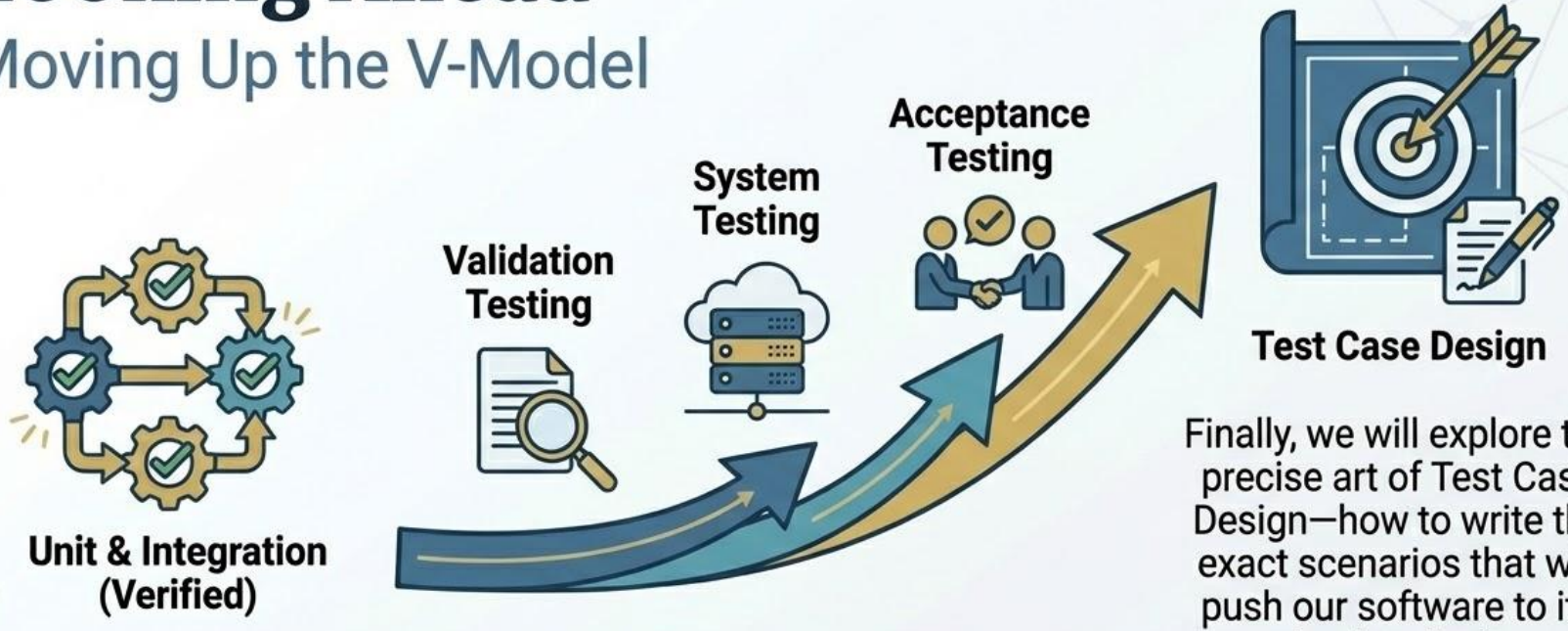
**Big Bang**

**Incremental**
(Top-Down, Bottom-Up, Sandwich)

Incremental strategies (Top-Down, Bottom-Up, or Sandwich) are vastly superior to the "Big Bang" approach because they allow for precise defect isolation.

# Looking Ahead

## Moving Up the V-Model




**Unit & Integration (Verified)**

We have successfully built and verified the individual pieces, and we have proven they can integrate without crashing.

Next, we'll continue our journey into higher-level testing strategies: Validation Testing, System Testing, and Acceptance Testing.

**Test Case Design**

Finally, we will explore the precise art of Test Case Design—how to write the exact scenarios that will push our software to its absolute limits.