



Software Testing and Quality Assurance

Module-2

System Testing Techniques and Strategies

By Prof. Pallavi Mahajan

Contains

- **System testing**
- **Unit Testing**
- **Concept of Unit Testing**
- **Static Unit Testing**
- **Defect Prevention**
- **Dynamic Unit Testing**
- **Mutation Testing**
- **Debugging**
- **Unit Testing in eXtreme Programming**

System Testing

What is System Testing ?

System Testing is a type of software testing that is performed on a completely integrated system to evaluate the compliance of the system with the corresponding requirements. In system testing, integration testing passed components are taken as input.

What is System Testing ?

System Testing is carried out on the whole system in the context of either system requirement specifications or functional requirement specifications or the context of both. System testing tests the design and behavior of the system and also the expectations of the customer.

System Testing

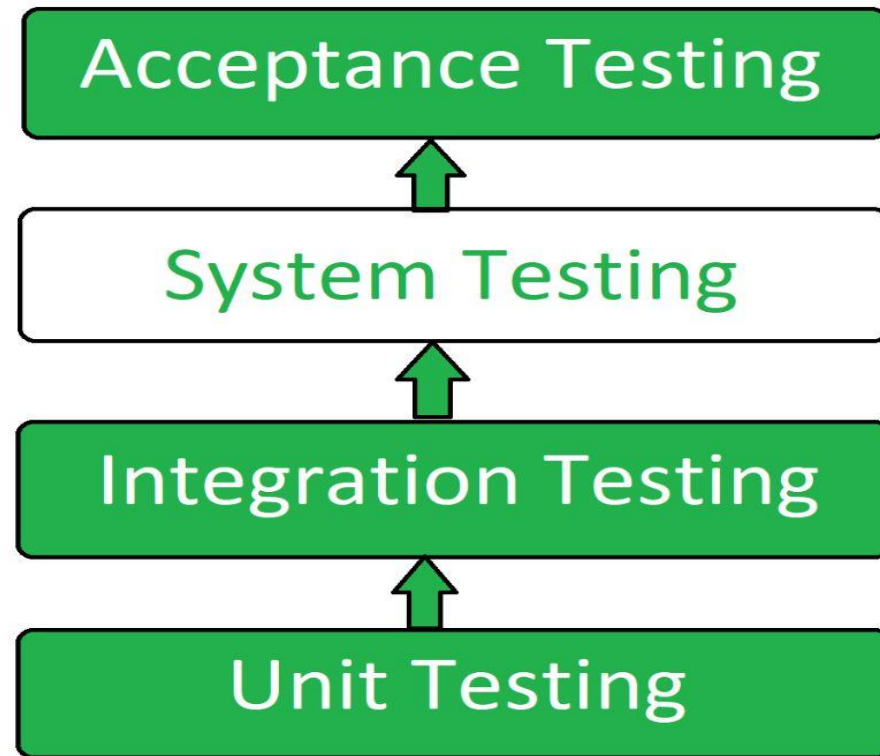
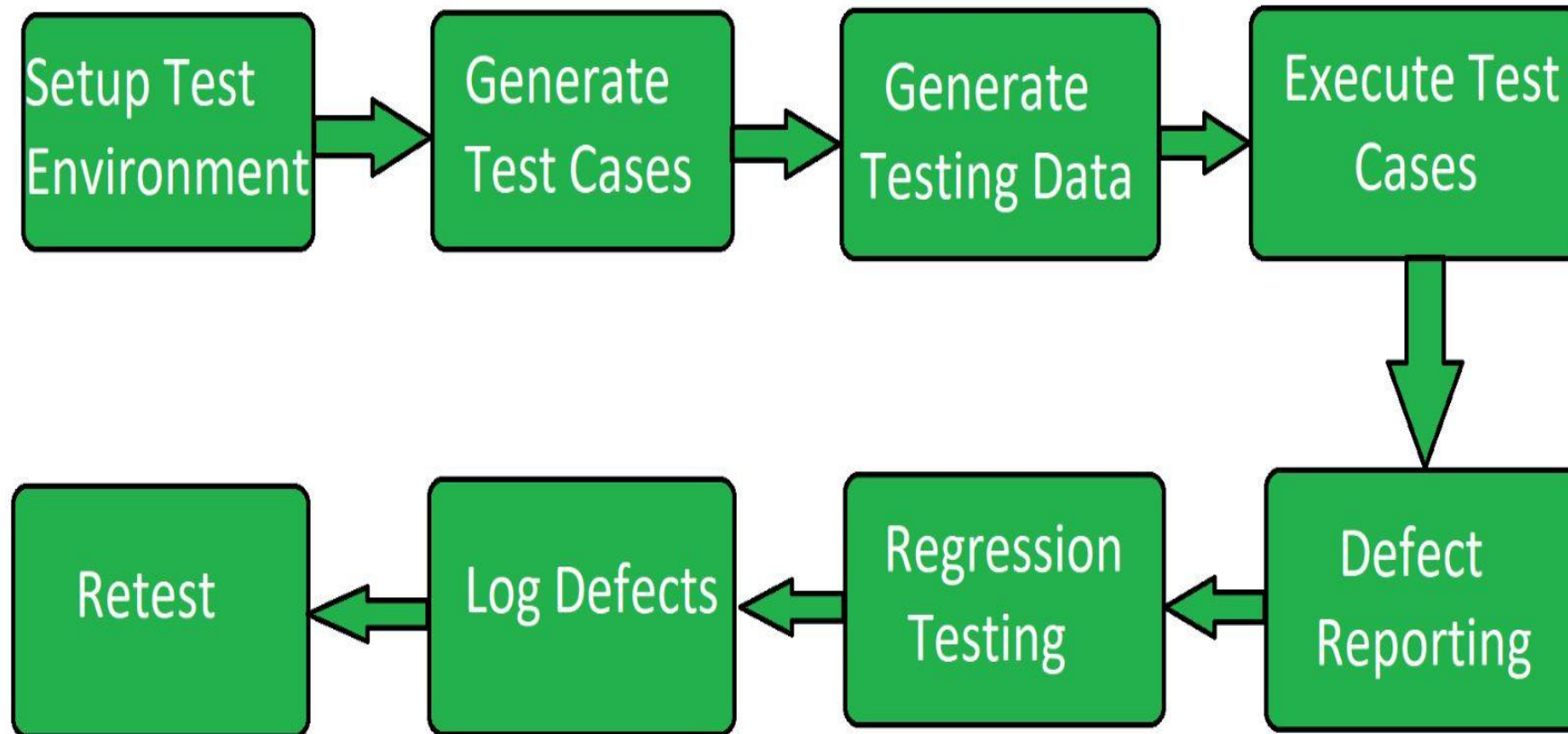


Fig: System Testing

System Testing Process



System Testing Process

System Testing is performed in the following steps:

- **Test Environment Setup:** Create testing environment for the better quality testing.
- **Create Test Case:** Generate test case for the testing process.
- **Create Test Data:** Generate the data that is to be tested.
- **Execute Test Case:** After the generation of the test case and the test data, test cases are executed.
- **Defect Reporting:** Defects in the system are detected.
- **Regression Testing:** It is carried out to test the side effects of the testing process.
- **Log Defects:** Defects are fixed in this step.
- **Retest:** If the test is not successful then again test is performed.

System Testing Example - Flight Reservation System (Airline)

- **Scenario:** A new feature is introduced in an airline's reservation system that allows customers to choose their seats during the booking process.

System Testing Steps:

- **Functional Testing:** Ensure the seat selection process works correctly, with available seats being displayed and booked.
- **Regression Testing:** Check if this new feature impacts existing functionality, like booking flights, processing payments, or sending booking confirmation emails.
- **Performance Testing:** Test the system's ability to handle simultaneous seat selection requests, especially during peak booking periods.
- **Security Testing:** Validate that sensitive data like passport information or credit card details is stored and transmitted securely.
- **Usability Testing:** Ensure the seat selection interface is intuitive for customers, and they can easily choose or change seats.

Unit Testing

Unit Testing

- is a level of the software testing process where individual units/components of a software/system are tested.
- The purpose is to validate that each unit of the software performs as designed.

Unit Testing

- a method by which individual units of source code are tested to determine if they are fit for use
- concerned with functional correctness and completeness of individual program units
- typically written and run by software developers to ensure that code meets its design and behaves as intended.
- Its goal is to isolate each part of the program and show that the individual parts are correct.

What is Unit Testing?

Concerned with

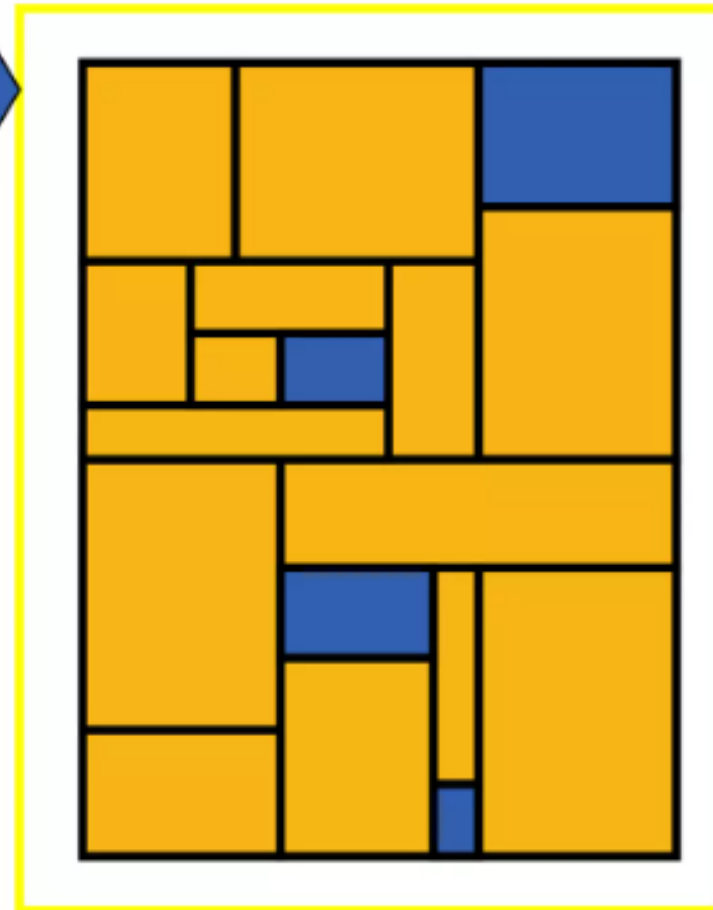
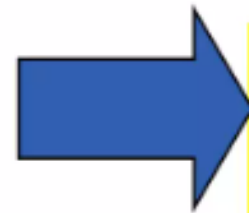
- ❑ Functional correctness and completeness
- ❑ Error handling
- ❑ Checking input values (parameter)
- ❑ Correctness of output data (return values)
- ❑ Optimizing algorithm and performance

What is Unit Testing?

- **Definition:** Testing program units (functions, methods, classes) in isolation.
- **Unit:** Can be a function, method, procedure, or even a class in OOP.
- **Purpose:** Ensures the program unit performs its intended function before integration

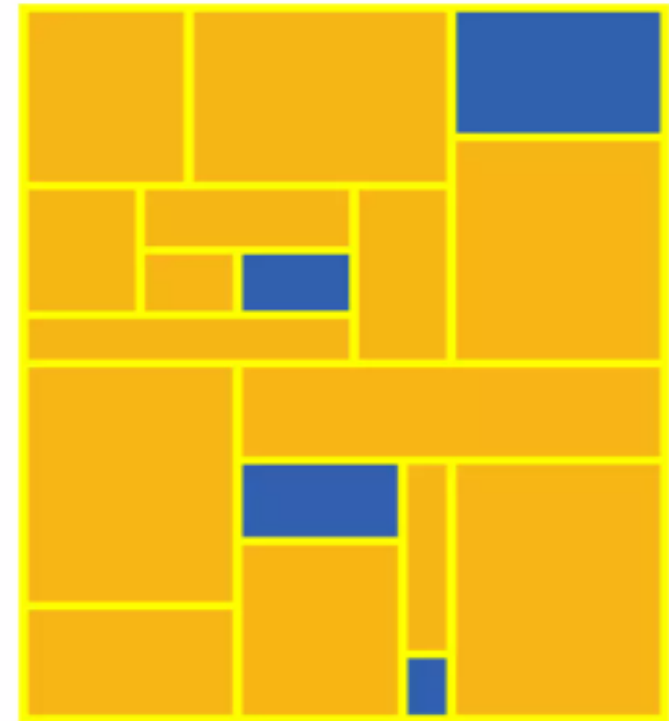
Traditional Testing Vs Unit Testing

- ❑ Test the system as a whole
- ❑ Individual components rarely tested
- ❑ Errors go undetected
- ❑ Isolation of errors difficult to track down



Traditional Testing Vs Unit Testing

- ❑ Each part tested individually
- ❑ All components tested at least once
- ❑ Errors picked up earlier
- ❑ Scope is smaller, easier to fix errors



Traditional Testing Vs Unit Testing

Unit Testing Ideals

- ❑ Isolatable
- ❑ Repeatable
- ❑ Automatable
- ❑ Easy to Write

Why Unit Test

- ❑ Faster Debugging
- ❑ Faster Development
- ❑ Better Design
- ❑ Excellent Regression Tool
- ❑ Reduce Future Cost

Why Unit Test

- **Isolated Testing:** Tests units in isolation to pinpoint errors easily.
- **Direct Access:** Programmer directly interacts with the input vector.
- **Distinct Execution Paths:** Verifies all possible paths within the unit.
- **Early Error Detection:** Catches errors early, reducing future costly fixes.

Why Unit Test

- ❑ Unit testing allows the programmer to refactor code at a later date, and make sure the module still works correctly.
- ❑ By testing the parts of a program first and then testing the sum of its parts, integration testing becomes much easier.
- ❑ Unit testing provides a sort of living documentation of the system.

Why Unit Test

Why are people writing unit tests?

Pay a little now
or
Pay a lot later

Objectives of Unit Testing

- **Execute every line of code:** Ensures no surprises in the future.
- **Test every predicate:** Evaluate predicates to both true and false values.
- **Functional Verification:** Ensure the unit performs as expected without errors.

Limitations of Unit Testing

- **Limited Scope:** Cannot guarantee system-wide functional correctness.
- **Integration Issues:** Errors might only show up in later phases like integration testing.
- **Cannot Test Everything:** Some issues only emerge after integrating with other units.

Who Conducts Unit Testing?

- **Programmers:** Conducted by the programmer who wrote the code.
- **Accountability:** Programmers are responsible for the quality of their own code.
- **Preventive Actions:** Focus on minimizing defects before integration.

Unit Testing Phases

1.Static Unit Testing:

- Code review and analysis without execution.
- Identify issues based on code structure and requirements.

2.Dynamic Unit Testing:

- Execute the code and observe outcomes.
- Identify runtime issues through actual execution.

Static Unit Testing

- **Non-Execution-Based Testing**
- **Code Review:** Ensures the code meets unit requirements without running it.
- **Example:** Handling program halting instructions like `abort()` vs `exit()` in C.
- **Benefit:** Less expensive, identifies issues early, reducing errors later.

Dynamic Unit Testing

- **Execution-Based Testing**
- **Program Execution:** Run the unit and check actual results.
- **Follow-up:** Static tests and dynamic tests go hand in hand. If static tests reveal new issues, dynamic testing must be repeated.

Static vs Dynamic Unit Testing

Static Unit Testing	Dynamic Unit Testing
Code review without execution	Code executed and outcomes observed
Identifies potential issues early	Detects runtime errors
Less expensive	More resource-intensive

Key Points

- **Importance of Unit Testing:** Critical for ensuring quality before integration.
- **Phases:** Static testing identifies issues early, while dynamic testing confirms functionality.
- **Collaboration:** Programmers are responsible for the quality of their own code.

Unit Test Example

Scenario: You have a function that calculates the area of a circle given its radius.

• **Test 1:** Verify that the function correctly calculates the area for a positive radius.

- Input: `radius = 3`

- Expected Output: `Area = 28.27` (approx.)

- Reason: This test ensures that the formula is working for typical cases.

• **Test 2:** Verify that the function returns zero for a radius of zero.

- Input: `radius = 0`

- Expected Output: `Area = 0`

- Reason: This tests the boundary condition for zero, which is a logical edge case.

• **Test 3:** Verify that the function handles negative values gracefully (if applicable).

- Input: `radius = -3`

- Expected Output: `Error / Exception`

- Reason: The test checks whether the function properly handles invalid input, ensuring it doesn't silently fail or return incorrect results.

Static Unit Testing

What is Static Unit Testing

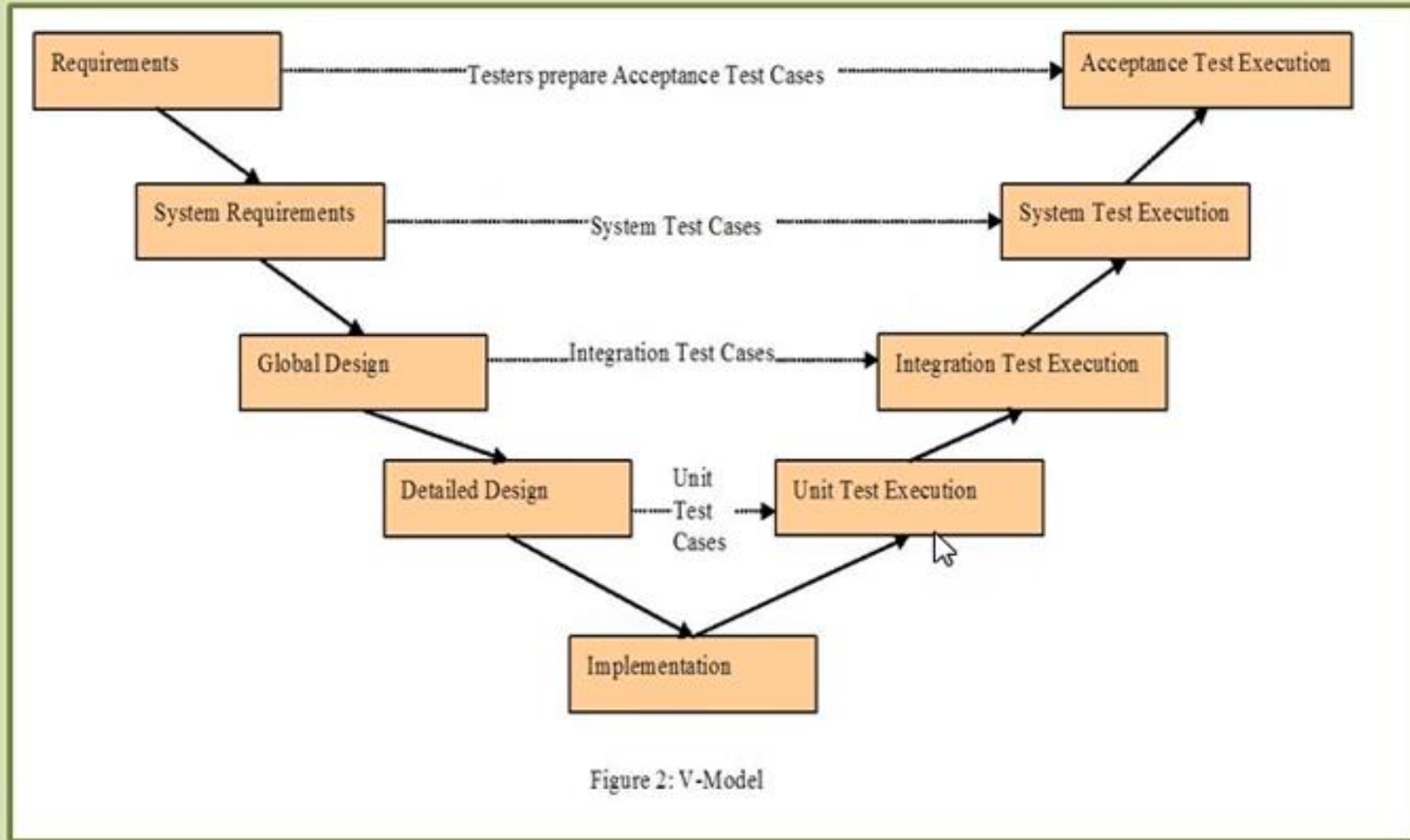
- **Part of software development philosophy for continuous inspection and correction**
- **Focuses on reviewing code before execution**
- **Aims to identify defects as early as possible in the process**
- **Review techniques include inspection and walkthrough**

Key Review Techniques

- **Inspection**
- **Peer group review, step-by-step examination**
- **Checked against predetermined criteria**
- **Walkthrough**
- **Author leads team through manual or simulated code execution**
- **Uses predefined scenarios**

Static Testing Vs Dynamic Testing

Static Testing



Dynamic Testing

Figure 2: V-Model

Benefits of Static Unit Testing

- **Detects defects early in the development process**
- **Reduces costs by identifying issues before testing**
- **Enhances software quality by ensuring fewer defects in the product**
- **Effective even in incomplete products, post-coding**

The Code Review Process

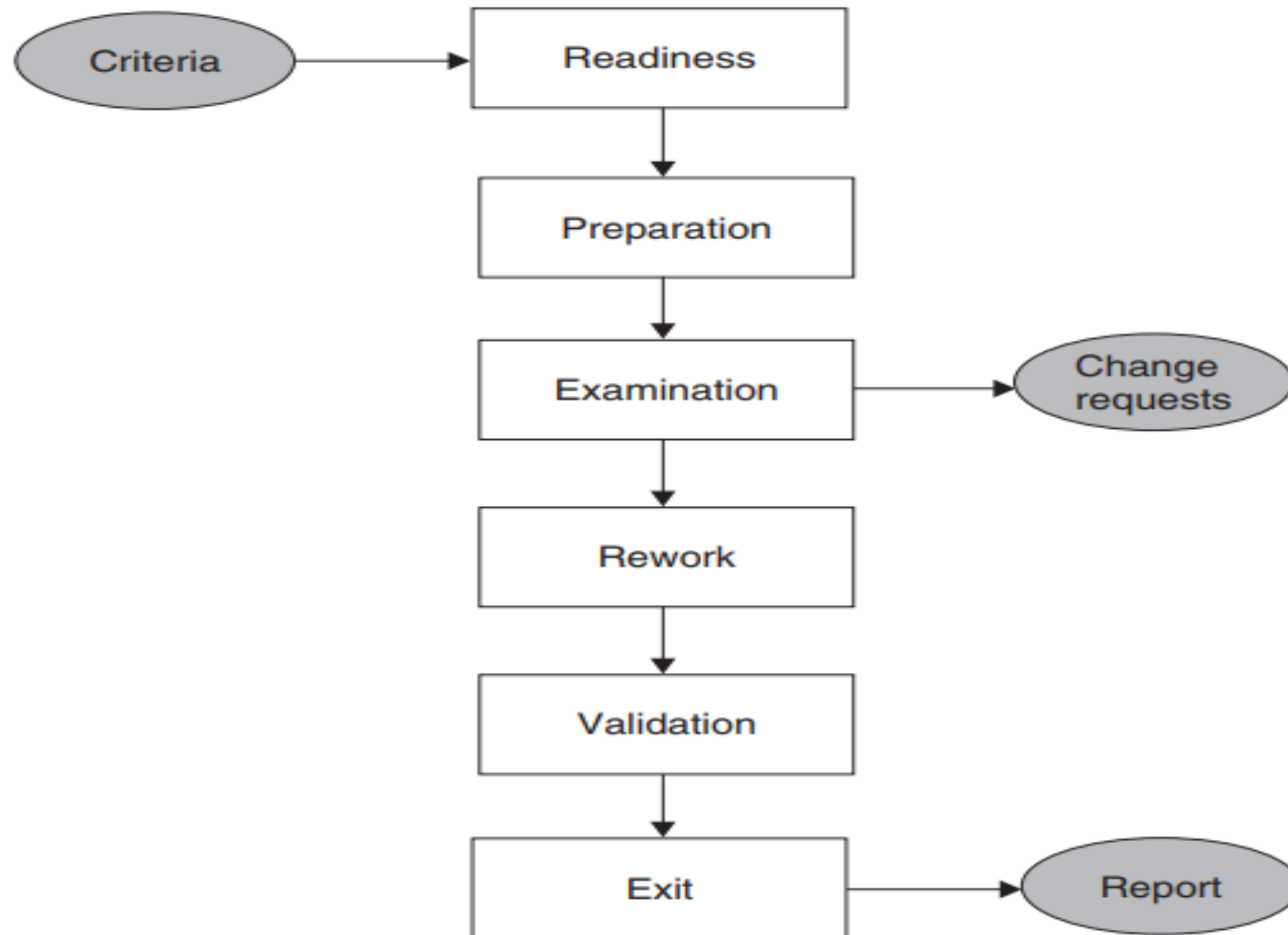


Figure 3.1 Steps in the code review process.

The Code Review Process

The six main steps:

- Readiness
- Preparation
- Examination
- Rework
- Validation
- Exit

Step 1: Readiness

Ensures the unit under test meets key criteria:

- **Completeness**
- **Minimal Functionality**
- **Readability**
- **Complexity**
- **Availability of design documents**

Step 2: Preparation

Before the meeting, each reviewer prepares:

- **A list of questions for clarification**
- **Change requests (CR) for improvements**
- **Suggested improvement opportunities**

Step 3: Examination

- **The author presents code logic and dependencies**
- **Reviewers identify potential defects (not resolved in the meeting)**
- **Change requests are documented**
- **Moderator ensures progress and focus**

Step 4: Rework

After the meeting:

- Author works on addressing change requests
- Documentation of the changes and improvements made
- Meeting summary shared with the review group

Step 5: Validation

- CRs validated by the moderator or designated person
- Ensures improvements are applied correctly
- Final version distributed to the group

Step 6: Exit

- The review is complete when:
- Every line of code is inspected
- Consensus reached on defect-free code
- CRs documented and validated
- Review meeting summary distributed

Code Review Metrics

- **Lines of Code (LOC) reviewed per hour**
- **Change Requests (CRs) per thousand lines of code**
- **CRs generated per hour**
- **Total number of CRs and hours spent per project**

Static Unit Testing Example - Static Review in a Healthcare Management System

Scenario: A hospital management system has a feature that handles appointment scheduling and availability checks.

Static Unit Testing Example - Static Review in a Healthcare Management System

Static Unit Testing Steps:

Code Walkthrough: Developers manually inspect the function that checks doctor availability to ensure it correctly queries the database and handles available time slots.

Boundary Conditions: The function is reviewed to ensure that it accounts for appointment limits (e.g., doctors can't schedule more than a certain number of appointments per day).

Data Validation: The code review ensures that patient data (such as name, contact info) is correctly validated before being saved in the system to avoid errors.

Optimization Review: The reviewer checks for inefficient database queries or excessive API calls that could slow down the system, suggesting optimizations if necessary.

Key Points

- **Static unit testing is an essential part of the software development cycle**
- **Helps detect defects early and reduces overall development costs**
- **A systematic, well-organized review process can significantly enhance software quality**

Defect Prevention

Introduction to Defect Prevention

Definition:

- **The goal of defect prevention is to reduce the number of change requests (CRs) generated during code review, as CRs indicate potential issues in code.**

Introduction to Defect Prevention

- **Why It Matters:** Addressing CRs means spending more resources and potentially delaying the project.
- **Objective:** Adopt defect prevention practices to reduce code errors and improve overall product quality.

Internal Diagnostic Tools:

- Build diagnostic tools (instrumentation code) to track internal states of units.
- Passive role in dynamic testing: Observing and recording without actively testing.

Guidelines for Error Detection

Standard Controls for Error Conditions:

- Standard Controls for Error Conditions:
- Detect issues like divide by zero and array index out of bounds.
- Handle invalid return values.
- Prevent buffer overflow/underflow errors.

Error Handling and Assertions

Error Messages & Help Text:

- Provide consistent messages from a common source.
- Root cause identification for users.

Assertions:

- Use assertions for detecting impossible conditions or undesirable behavior.
- Examples: Preconditions, postconditions, and invariants.

Further Defect Prevention Measures

Leave Assertions in Code:

- Deactivate assertions in the release version for performance improvement.

Documentation:

- Fully document unclear assertions for better understanding.

Advanced Techniques for Preventing Defects

Reverse-Compute Inputs After Computation:

- For example, after computing the square root, square the output and check it against the input value.

Message Passing & Buffer Management:

- Monitor buffer availability to prevent running out of space.

Advanced Techniques for Preventing Defects

- **Timer Routine:**

- Prevent infinite loops with a timer that expires and invokes an exception handler.

- **Loop Counter:**

- Monitor loop iterations to prevent irregular executions.

Decision Logic for Preventing Defects

Decision Logic Variable:

- Use a variable to track decision logic branches and ensure no fall-through conditions exist.

Example of Defect Prevention: Online Ticketing System - Preventing Performance Issues

- **Scenario:** An online ticket booking platform faces performance bottlenecks during peak times, leading to slow response times or system crashes when users try to book tickets for popular events.

Example of Defect Prevention: Online Ticketing System - Preventing Performance Issues

Defect Prevention Activities:

- **Load Testing:** Conducting load tests early in development to simulate heavy traffic during peak times and identify potential bottlenecks in the system.
- **Database Indexing:** Ensuring that critical database queries related to ticket availability and booking are optimized with proper indexing to speed up response times.
- **Code Refactoring:** Continuously refactoring code to improve efficiency, removing unnecessary loops and redundant logic that could degrade system performance under high load.
- **Caching Mechanisms:** Implementing caching strategies for frequently accessed data, like available tickets, to reduce database load and improve performance during high demand.

Key Points

Online Ticketing System - Preventing Performance Issues

- **Scenario:** An online ticket booking platform faces performance bottlenecks during peak times, leading to slow response times or system crashes when users try to book tickets for popular events.

Defect Prevention Activities:

- **Load Testing:** Conducting load tests early in development to simulate heavy traffic during peak times and identify potential bottlenecks in the system.
- **Database Indexing:** Ensuring that critical database queries related to ticket availability and booking are optimized with proper indexing to speed up response times.

Key Points

- Preventing defects during development is essential for avoiding costly CRs and improving code quality.
- Implementing proactive measures like assertions, diagnostics, and error handling can significantly reduce defects.

Dynamic Unit Testing

Introduction to Dynamic Unit Testing

- **Definition:** Execution-based unit testing is also called dynamic unit testing, where a program unit is executed in isolation.
- **Key Concept:** Unlike normal execution, the unit is tested in an emulated environment to simulate its actual execution context.

The Dynamic Unit Testing Process

- **Isolation:** The unit is taken out of its real execution environment.
- **Emulated Environment:** Code is written to emulate the actual execution environment.
- **Execution:** The compiled unit and emulated environment are executed with selected inputs.
- **Outcome Comparison:** The result is compared with the expected outcome. Any differences indicate faults in the code.

Test Environment Creation

Test Environment Structure:

- **Caller Unit:** Known as the test driver.
- **Units Called by the Unit Under Test:** These are emulated by stubs.

Purpose: To ensure that any fault can be traced back only to the unit under test.

Test Driver & Stubs

Test Driver:

- Invokes the unit under test and compares the actual outcome with the expected outcome.
- Facilitates compilation and provides input data in the expected format.

Stubs:

- "Dummy subprograms" that replace units called by the unit under test.
- Provide evidence that they were called and return precomputed values.

Reusability of Test Driver & Stubs

Reusability: Test driver and stubs are reused for regression testing and throughout the life cycle of the unit.

Modification Considerations:

- If a unit is modified, the corresponding test driver and stubs may need updates.
- Drivers should be independent of external data files, with their own segregated test data.

Automated Success/Failure Detection

Automated Checks:

- The test driver should automatically determine if the unit passed or failed the test.
- It should check for issues like memory leaks and file handling (open/close state).
- The driver can also check internal variables that are not accessible in other testing levels.

Maintaining Test Drivers and Stubs

- **Continuous Maintenance:** Test drivers and stubs should be maintained and updated whenever a new fault is detected.
- **Version Control:** The test driver and stubs should be stored in a version control system alongside the unit they test.

Selecting Input Test Data

Techniques for Selecting Test Data:

- **Control Flow Testing:** Involves creating a control flow graph, selecting criteria, and generating input values.
- **Data Flow Testing:** Involves a data flow graph, selecting criteria, and deriving input values to test specific paths.
- **Domain Testing:** Focuses on detecting domain errors by selecting test data to identify such faults.
- **Functional Program Testing:** Selects specific input values for a given domain and computes the expected outcome.

Control Flow Testing

Steps in Control Flow Testing:

- Draw a control flow graph.
- Select testing criteria.
- Identify paths in the graph.
- Derive path predicate expressions.
- Generate test case input values based on the predicates.

Data Flow Testing

- Steps in Data Flow Testing:
- Draw a data flow graph.
- Select data flow testing criteria.
- Identify paths in the graph.
- Derive path predicate expressions.
- Generate test case input values to exercise the corresponding path.

Domain Testing

Domain Testing Approach:

- Focus on domain errors, which are faults specific to input and output domains.
- Test data are selected to detect domain-specific faults.

Functional Program Testing

Steps in Functional Program Testing:

- Identify input and output domains.
- Select special values within those domains.
- Compute the expected outcome for those values.
- Consider combinations of input values to test various conditions.

Example of Dynamic Unit Testing: Bank Loan Approval System - Loan Eligibility

Scenario: Testing a function that calculates whether a customer is eligible for a loan based on income, credit score, and loan amount.

Example of Dynamic Unit Testing: Bank Loan Approval System - Loan Eligibility

Dynamic Unit Testing Steps:

Test Case 1: Test eligibility with a high income and good credit score.

Input: Income = \$100,000, Credit Score = 750, Loan Amount = \$50,000.

Expected Output: Eligible for loan.

Test Case 2: Test eligibility with a low credit score.

Input: Income = \$50,000, Credit Score = 500, Loan Amount = \$30,000.

Expected Output: Not eligible for loan due to low credit score.

Test Case 3: Test eligibility with a high loan amount that exceeds the eligible loan limit.

Input: Income = \$60,000, Credit Score = 700, Loan Amount = \$150,000 (loan limit is \$100,000).

Expected Output: Not eligible for loan due to excessive loan amount.

Outcome: The loan eligibility function is executed with various inputs to validate that the system correctly approves or denies loans based on the given criteria.

Key Points

- **Summary:** Dynamic unit testing isolates units to ensure faults are identified specifically within the unit under test.
- **Reusability:** Test drivers and stubs should be maintained throughout the unit's life cycle.
- **Test Data:** Using control flow, data flow, domain, and functional testing helps in thorough testing of units.

Mutation Testing

Introduction to Mutation Testing

- **Definition:** Mutation Testing is a technique to measure the adequacy of test data (test cases) by introducing small changes (mutations) to a program.
- **Purpose:** Exposes weaknesses in test cases and helps improve their quality.
- **Origin:** Proposed by Dick Lipton in the late 1970s, with seminal work by DeMillo, Lipton, and Sayward.

What is Mutation Testing?

- **Mutation:** A modification made by introducing small, legal syntactic changes to the code.
- **Mutant:** A modified version of the original program.
- **Goal:** To test how well a set of test cases can detect faults in the mutants.

Mutants and Their Types

- **Equivalent Mutants:** Mutants that produce the same output as the original program.
- **Faulty Mutants:** Mutants that produce incorrect results and are killed by test cases.
- **Killable Mutants:** Mutants that are not killed by the current test suite, indicating inadequate test coverage.

Mutation Score

➤ Mutation Score: The percentage of non-equivalent mutants killed by the test suite.

➤ Formula:

$$\text{Mutation Score} = (100 \times D) / (N - E)$$

Where:

D = Dead (killed) mutants

N = Total mutants

E = Equivalent mutants

➤ Ideal Mutation Score: 100% (test suite kills all non-equivalent mutants)

Mutation Testing Example

```
main(argc, argv)
{
    int argc, r, i;
    char *argv[];
    {
        r = 1;
        for(i = 2; i <= 3; i++)
            if (atoi(argv[i]) > atoi(argv[r])) r = i;
        printf("Value of the rank is %d \n", r);
        exit(0);
    }
}
```

- **Test Suite:**

- Test Case 1: Input: 1 2 3 → Output: Rank = 3
- Test Case 2: Input: 1 2 1 → Output: Rank = 2
- Test Case 3: Input: 3 1 2 → Output: Rank = 1

Mutation Testing Example

Mutants:

- ❑ **Mutant 1:** Change loop to for `i = 1 to 3` do
- ❑ **Mutant 2:** Modify condition to `if (i > atoi(argv[r]))`
- ❑ **Mutant 3:** Modify condition to `if (atoi(argv[i]) >= atoi(argv[r]))`
- ❑ **Mutant 4:** Modify condition to `if (atoi(argv[r]) > atoi(argv[r]))`

Mutation Testing Results

- Mutant 1 and 3: Pass the test suite (not killed).
- Mutant 2: Fails test case 2 (killed).
- Mutant 4: Fails test case 1 and 2 (killed).
- Mutation Score Calculation:
 - Mutants created = 4
 - Mutants killed = 2
 - Mutation Score = 50%

Improving Mutation Score

- **Analyzing Mutant 1:** Identified as equivalent mutant (no impact on results).
- **Enhancing Test Suite:** Add a fourth test case to kill Mutant 3.
 - **Test Case 4:** Input: 2 2 1 → Kills Mutant 3.
 - **New Mutation Score:** 100%.

Steps in Mutation Testing

- Start with Program P and Test Cases T.
- Run test cases against P.
 - Modify the program if any test case fails.
- Create mutants (P_i) with syntactic changes to P.
- Run test cases against each mutant:
 - a. If P_i 's output differs from P's output, P_i is killed.
 - b. If P_i 's output matches P's output, check if P_i is equivalent or killable.
- Calculate Mutation Score.
- If score is not high enough, create new test cases and repeat.

Assumptions in Mutation Testing

- Competent Programmer Hypothesis:
- Assumes programmers create programs with few errors, and mutations should be small and realistic.
- Coupling Effect:
- Assumes that detecting simple faults in a program will also detect more complex faults.
- Supported by empirical studies (Offutt, Wah).

Assumptions in Mutation Testing

➤ **Competent Programmer Hypothesis:**

- Assumes programmers create programs with few errors, and mutations should be small and realistic.

➤ **Coupling Effect:**

- Assumes that detecting simple faults in a program will also detect more complex faults.

- Supported by empirical studies (Offutt, Wah).

Advantages and Challenges

Advantages:

- ✓ Helps identify weaknesses in test cases.
- ✓ Can lead to a highly robust test suite.
- ✓ Powerful as a white-box testing technique.

Challenges:

- ✓ **Time-consuming:** Requires generating and testing many mutants.
- ✓ Identifying equivalent mutants can be difficult.
- ✓ Requires significant computational resources.

Automated Tools for Mutation Testing

Mothra: A tool used to automate mutation testing and speed up the process.

Recent Trends: Mutation testing has seen a resurgence with advancements in computing power and automated tools.



Example of Mutation Testing: E-commerce Website - Discount Calculation

Scenario: An e-commerce platform calculates a discount for a customer based on the items in their cart. The function checks if the user is eligible for a discount and then applies the correct discount percentage.

Example of Mutation Testing: E-commerce Website - Discount Calculation

- **Original Code:**



python

 Copy  Edit

```
if cart_value > 100:  
    discount = cart_value * 0.1 # 10% discount  
else:  
    discount = 0
```

- **Mutant 1:** Change the comparison operator to `>=`:



python

 Copy 

```
if cart_value >= 100:  
    discount = cart_value * 0.1 # 10% discount
```

- **Mutant 2:** Change the discount percentage to 5%:

python

 Copy 

```
if cart_value > 100:  
    discount = cart_value * 0.05 # 5% discount
```

Example of Mutation Testing: E-commerce Website - Discount Calculation

- **Mutant 3:** Always apply a discount of 10%, regardless of the cart value:

python

 Copy  Edit

```
discount = cart_value * 0.1 # Discount always applied
```

Mutation Testing Objective:

- The goal is to test if the current test suite can detect these mutations. If a test case exists that verifies the discount is applied only when the cart value is above 100, and that the correct discount percentage is applied, the mutants will be "killed." If not, the test suite needs to be improved.

Key Points

- Mutation Testing is a valuable technique for improving test suite quality.
- It helps in identifying the adequacy of existing test cases. Ensures that test cases detect various types of faults in the program.

Debugging

Introduction to Debugging

Definition:

Debugging is the process of identifying and fixing faults after a program failure, triggered by a test revealing an issue.

Goal:

To isolate and fix the root cause of a failure.

Approaches to Debugging

1. Brute Force Approach

Philosophy: "Let the computer find the error."

Method: Use print statements scattered throughout the code for tracing errors.

Tools:

- Dynamic debuggers for stepping through code, inspecting variable changes.
- Instrumentation code to log values and detect problems.
- Memory dumps after failures for understanding the final state.

Approaches to Debugging (2/3)

2. Cause Elimination Approach

Induction Process:

- Collect data and organize it to find patterns.
- Form a hypothesis about the cause.
- Conduct tests to prove or disprove the hypothesis.

Deduction Process:

- List all possible causes in order of likelihood.
- Eliminate causes one by one by running tests.

Approaches to Debugging (3/3)

3. Backtracking Approach

- **Method:** Start from the point where the failure occurred and trace backward through the code.
- **Challenge:** As programs grow in size, tracing backward becomes complex due to multiple possible paths.

Handling New Faults During Debugging

Problem: New, undetected faults may be noticed while debugging.

Solution:

- **Create a Change Request (CR):** Discuss with team members and architects.
- **Defect Tracking:** Document the new issue and proceed with fixing the original problem.
- **Note:** This process interrupts debugging but ensures a more structured approach.

Debugging Heuristic - Step 1

Step 1: Reproduce the Symptoms

- **Methods:**
 - Read troubleshooting guides and logs.
 - Turn on diagnostic code to gather more information.
 - Use causal analysis to pinpoint the root cause.

Debugging Heuristic - Step 2 and 3

Step 2: Formulate Hypotheses

Develop likely hypotheses based on the symptoms and collected data.

Step 3: Develop Test Scenarios

- Test Design:** Create test cases to prove or disprove each hypothesis.

- Test Types:** Static (code reviews, documentation) and Dynamic (execution tests).

Debugging Heuristic - Step 4 and 5

Step 4: Prioritize Test Cases

1. Execute the most likely and least expensive test cases first.
2. **Cost Factor:** Start with low-cost tests, followed by expensive ones.

Step 5: Execute Tests

1. Execute test cases, gather evidence, and refine data if necessary.

Debugging Heuristic - Step 6 and 7

Step 6: Fix the Problem

- **Simple Fix:** May involve changing code or adding a new line.
- **Complex Fix:** May involve code redesign.
- **Retest:** Confirm the failure does not reoccur after applying fixes.

Step 7: Document the Changes

- **Documentation:**
 - Update source code and system docs.
 - Update dynamic test cases.
 - Log defects in the defect tracking system.

Key Points

- ✓ Debugging is an essential process to ensure software correctness.
- ✓ A systematic approach helps isolate faults efficiently.
- ✓ Tools and structured techniques, like the debugging heuristic, streamline the process.

Unit Testing in eXtreme Programming

Introduction to XP and TDD

Title: Introduction to Extreme Programming and TDD

Content:

- TDD (Test-Driven Development) is key to XP methodology.
- In TDD, tests are written before production code to ensure functionality.
- Unit testing in XP focuses on writing small, simple tests, followed by incremental code development.

The Test-First Approach in XP

Content:

TDD focuses on writing tests first, then creating code to pass them.

Steps:

1. Pick a requirement (story).
2. Write a test for the requirement.
3. Write code to pass the test.
4. Run all tests.
5. Refactor code until all tests pass.

XP Test-First Process

Content:

- Illustration of the cycle from the provided Figure 3.3.
- Visual representation of the Test-First process.
- **Cycle:**
 1. Add test.
 2. Write code to pass test.
 3. Rework and execute all tests.
 4. Repeat until the story is complete.

The Three Laws of TDD

Content:

- **Law 1:** Write a failing test first before production code.
- **Law 2:** Write just enough test code to fail.
- **Law 3:** Write just enough production code to pass the test.
- These laws ensure minimal and efficient development.

Benefits of Unit Testing in XP

Content:

- Provides a clear focus on requirements (user stories).
- Code and unit tests are released together.
- Helps prevent incomplete or untested code from being released.
- Acts as a safety net for regression testing and refactoring.

Pair Programming in XP

Content:

- **Two programmers work side by side.**
- **One writes code, the other reviews it.**
- **Ensures quality and keeps the user story in mind.**
- **Similar to the two-person inspection strategy.**

Key Points

- **Unit testing in XP ensures quality, helps focus on requirements, and encourages collaboration through pair programming.**
- **Following the TDD process and the three laws ensures efficient development cycles and prevents defects.**



Thank You..