# Module 2

farisbashatm@gmail.com

# Unit Testing

1. **Definition:** Unit testing involves testing individual program units in isolation.
    - Here, a program unit could be a function or method responsible for adding items to the shopping cart.
2. **Scope of Units:** Units can be functions, methods, or classes.
    - In our scenario, the "add item to cart" function can be considered a program unit.
3. **Testing in Isolation:** Units are tested alone to easily identify and fix errors.
    - For example, when testing the "add item to cart" function, you only focus on ensuring that it adds the item correctly without considering other parts of the application.
4. **Purpose:** To verify each unit's correctness before integration with others.
    - Before integrating the "add item to cart" function with the rest of the shopping cart application, you want to make sure that it correctly adds items.
5. **Execution Criteria:** Every line of code and predicate should be executed, and the unit should perform its intended function without known errors.
    - You ensure that when you call the "add item to cart" function with valid item details, it adds the item to the cart and updates the cart's total correctly.
6. **Limitations:** Not all errors can be detected in isolation; some will surface during integration and system testing.
    - Even if the "add item to cart" function works correctly on its own, there might be issues when it interacts with other parts of the shopping cart application.
7. **Responsibility:** The programmer who wrote the unit performs the testing to ensure quality and correctness.
    - The developer responsible for implementing the "add item to cart" function tests it to ensure it works correctly.
8. **Phases:** Unit testing is split into static and dynamic phases.
    - Static unit testing involves analyzing the code without executing it, like reviewing the logic of the "add item to cart" function. Dynamic unit testing involves actually running the code to verify its behavior, such as testing that calling the function adds items to the cart as expected.

# Static Unit Testing

Static unit testing is a non-execution-based examination of code, aimed at identifying potential issues through review rather than actual execution. This process involves

inspecting the code to find possible defects that could arise during runtime, ensuring the code meets its requirements and standards.

### Inspection vs. Walkthrough

- **Inspection:** A detailed, step-by-step peer review of the code against predefined criteria, introduced by Michael Fagan.
- **Walkthrough:** A less formal review led by the code author, presenting the code to the team using predefined scenarios, coined by Edward Yourdon.

## Roles in Review

Let's illustrate these roles with a scenario involving a code review meeting for a software development project:

**Scenario: Code Review Meeting for a New Feature Implementation**

1. **Moderator:**
   - *Role:* Chairs the meeting, ensuring fairness and guiding its pace. Ideally chosen from another team
   - *Example Scenario:* Sarah, a project manager from another team, is chosen as the moderator to oversee the code review impartially.
2. **Author:**
   - *Role:* The creator of the code under review.
   - *Example Scenario:* Alex, a software developer, has implemented a new feature and is the author of the code being reviewed.
3. **Presenter:**
   - *Role:* Presents the code being reviewed to ensure understanding and continuity.
   - *Example Scenario:* Emma, another developer familiar with Alex's work, presents the code in case Alex is absent or to provide additional clarity.
4. **Recordkeeper:**
   - *Role:* Documents identified issues and suggested actions during the review.
   - *Example Scenario:* David, a QA engineer, takes notes on identified issues and proposed solutions for later reference.
5. **Reviewers:**
   - *Role:* Experts in the code's subject area, focusing on its content.
   - *Example Scenario:* Claire, James, and Lisa, experienced developers with expertise in the project's technology stack, serve as reviewers.
6. **Observers:**
   - *Role:* Passive participants who attend to learn but don't actively engage in the review.
   - *Example Scenario:* Mark and Rachel, junior developers, attend the meeting to observe how code reviews are conducted and learn from experienced team

members.

In this scenario, each role contributes to a productive code review meeting by ensuring fairness, understanding, documentation, expertise, and learning opportunities.

## Code Review Process

1. **Readiness:** The code must be complete, minimally functional, readable, and sufficiently complex to warrant a review. Relevant documents (requirements, design specs) must be available.
2. **Preparation:** Reviewers read the code beforehand, prepare questions, identify potential change requests (CRs), and suggest improvements.
3. **Examination:** During the meeting:
    - The author presents the code's logic.
    - The presenter (different from the author) reads the code line by line.
    - Reviewers raise questions and suggest changes.
    - The recordkeeper documents CRs and suggestions.
4. **Rework:** The author addresses the CRs, documenting changes and improvements within the agreed time frame.
5. **Validation:** An independent person (moderator or designated reviewer) validates the CRs, ensuring changes are correctly implemented.
6. **Exit:** The review is considered complete when all lines of code are inspected, critical issues are resolved, and all CRs are documented and validated.

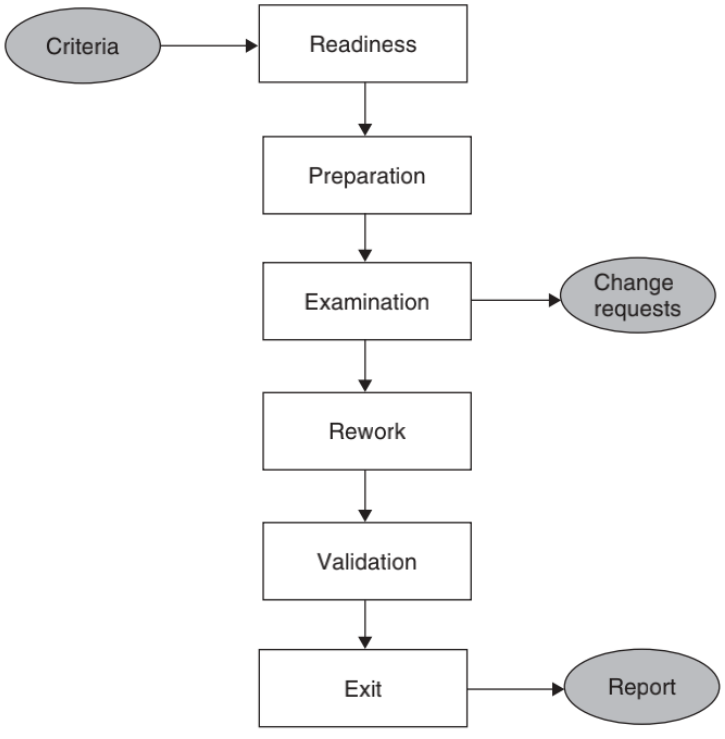



Figure 3.1 Steps in the code review process.

### Metrics for Code Review

Collecting data during code reviews helps evaluate and improve the process, facilitating better planning for future projects. Common metrics include:

- Number of lines of code reviewed per hour.
- Number of CRs per thousand lines of code.
- Number of CRs per hour.
- Total number of CRs and hours spent on code review per project.

Static unit testing, by combining thorough reviews and structured processes, enhances code quality and reduces the likelihood of defects in later stages of software development.

### Example Checklist

C Programming Code Review Checklist

1. Does the code meet the requirements specified in the design specification?
2. Does the code solve the problem as intended?
3. Does the module duplicate functionality available in existing modules?
4. Are the correct libraries and versions being used?
5. Does each function have a single entry and exit point?
6. Is the cyclomatic complexity of any function greater than 10?
7. Can each function be understood within 10-15 minutes?
8. Are naming conventions for identifiers followed consistently?
9. Is the code adequately commented?
10. Are all variables and constants correctly initialized?
11. Are global or shared variables carefully controlled?
12. Are there any hard-coded values that should be declared as variables?

# Dynamic Unit Testing

Dynamic unit testing is an execution-based testing method where a program unit is tested in isolation. Unlike ordinary execution, the unit is removed from its actual environment and tested within an emulated environment, created using test drivers and stubs. This approach allows for precise fault detection within the unit under test.

**Key Steps in Dynamic Unit Testing:**

1. **Isolation of Unit Under Test:** The unit is taken out of its actual execution environment.
2. **Emulation of Environment:** The actual environment is emulated using additional code, allowing the unit and this environment to be compiled together.
3. **Execution with Selected Inputs:** The compiled unit is executed with specific inputs, and outcomes are collected and compared to expected results to identify any discrepancies.

**Components of a Dynamic Unit Test Environment:**

1. **Test Driver:**
   - **Function:** Invokes the unit under test, provides input, and compares the actual outcome with the expected outcome.
   - **Role:** Acts as the main program during testing, facilitating compilation and input provision.
2. **Stubs:**
   - **Definition:** Dummy subprograms replacing units called by the unit under test.
   - **Tasks:** Show evidence of being called (e.g., printing a message) and return precomputed values to the caller to allow continued execution.

**Example**

```
# Function to divide 2 number , ie this function is our unit to test
def divide(a, b):
    try:
            return a / b
        except:
                send_error_notification_stub() # this is our stub in this
test

# Test Drive for calling the unit ( ie, divide() in this case )
def test_drive():
    divide(4, 2)
    divide(5, 0)

# Run the test
test_drive()
```

In this example:

- **Unit Under Test**: The `divide` function is the unit under test. It's responsible for dividing two numbers (`a` by `b`). However, it includes error handling to handle division by zero.
- **Stub**: The `send_error_notification_stub` function is a stub. It's used to simulate the behavior of sending an error notification. In this case, when an error occurs (i.e., division by zero), the stub is invoked to notify about the error.
- **Test Drive**: The `test_drive` function serves as the test driver. It's responsible for calling the unit under test, which is the `divide` function, with different sets of inputs. In this case, it tests two scenarios: dividing 4 by 2 (which should succeed) and dividing 5 by 0 (which should raise an error).

**Scaffolding:**

- **Definition:** The combination of test drivers and stubs used to create the test environment.
- **Reusability and Maintenance:** Test drivers and stubs are reused in regression testing and must be maintainable and version-controlled.

**Best Practices for Test Drivers and Stubs:**

- **Dedicated Test Drivers:** Each unit should have its own test driver and necessary stubs.
- **Automated Success/Failure Checks:** Drivers should automatically determine the success or failure of tests.
- **Memory and Resource Checks:** Drivers should check for memory leaks and proper file handling.
- **Version Control:** Test drivers and stubs should be reviewed, cross-referenced with the unit, and checked into version control systems.

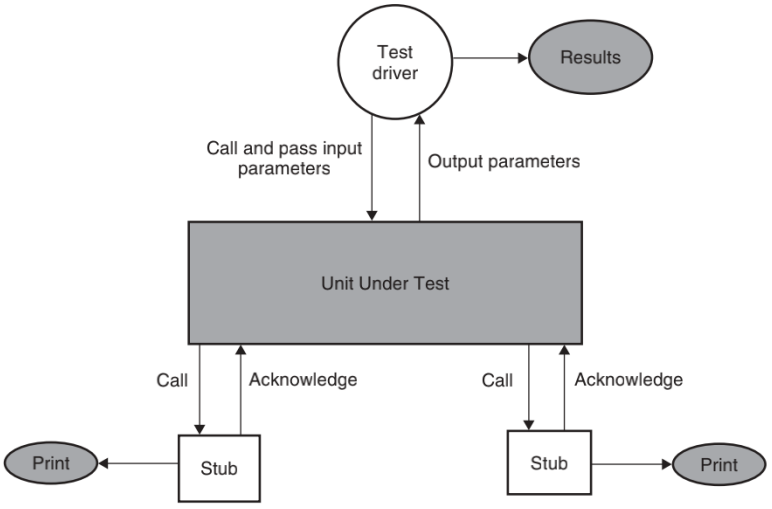


Figure 3.2 Dynamic unit test environment.

## Techniques for Selecting Input Test Data:

### 1. Control Flow Testing:

- **Draw Control Flow Graph:** Nodes for statements/blocks, edges for control paths.
- **Select Testing Criteria:**
  - *Statement Coverage:* Test every statement.
  - *Branch Coverage:* Test every branch direction (true/false).
  - *Path Coverage:* Test all possible paths.
- **Identify Paths:** Determine paths to test based on criteria.
- **Generate Test Cases:** Find inputs to execute specific paths.

### 2. Data Flow Testing:

- **Draw Data Flow Graph:** Highlight variable definitions and uses.

- **Select Testing Criteria:**
  - *Def-Use Pairs:* Test all definition-use pairs.
  - *All-Uses Coverage:* Test all uses of each variable.
- **Identify Paths:** Focus on paths involving def-use pairs.
- **Generate Test Cases:** Create inputs to test variable interactions.

**3. Domain Testing:**

- **Define Subdomains:** Divide input domain into subdomains.
- **Select Test Cases:** Choose representative values from each subdomain to catch domain-specific errors.

**4. Functional Program Testing:**

- Identifies input and output domains, selects special values, computes expected outcomes, and tests combinations of input values.

**Major Differences Between Control Flow Testing and Data Flow Testing**

| Aspect | Control Flow Testing | Data Flow Testing |
|---|---|---|
| **Focus** | Execution paths and control structures | Variable definitions and their usage |
| **Goal** | Ensure all paths and branches are executed | Track data through the code to ensure proper usage |
| **Techniques** | Path testing, branch testing, loop testing | Definition-use chains (DU), definition-clear paths |
| **Metrics** | Statement coverage, branch coverage, path coverage | DU-pairs coverage, def-clear paths coverage |
| **Example** | Verify all if-else branches are tested | Ensure every variable definition is used correctly |
| **Use Case** | Identifying missing paths, unreachable code | Detecting incorrect variable usage or data flows |

# Static vs. Dynamic Unit Testing

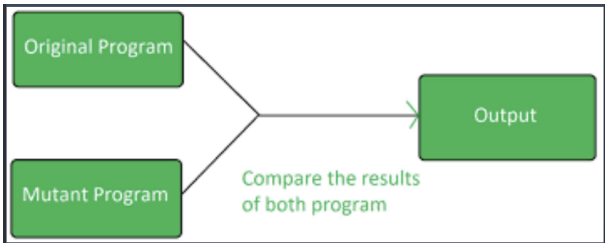
| Static Testing | Dynamic Testing |
|---|---|
| Non-execution-based examination | Execution-based testing |
| Reviews code without running it | Tests code by running it |
| Identifies potential issues through review | Detects faults by executing code |
| Techniques include inspection and walkthroughs | Techniques include unit testing with test drivers and stubs |

| Static Testing | Dynamic Testing |
| --- | --- |
| Conducted by code reviewers and authors | Conducted by developers using test frameworks |
| Example: Code review, checklist | Example: Running unit tests, using mock objects |

==========

# Mutation Testing

Mutation testing is a technique used to evaluate the effectiveness of a test suite. By introducing small changes, or mutations, into a program and observing if the test suite detects the errors caused by these changes, we can assess how well the test suite performs. If the test suite fails to catch these errors, it indicates areas where the test suite could be improved.




## Key Concepts in Mutation Testing

### Mutation and Mutants

- **Mutation**: The process of making a small change to the original program.
- **Mutant**: A version of the program that has been altered by a mutation. The mutant should still be syntactically correct and compile without errors.

### Mutation Operator:

A rule that specifies syntactic variations of strings generated from a grammar.

### Ground String

A **ground string** is the original, unmodified version of a program or piece of code used as a baseline in mutation testing. It serves as the reference against which all mutants (modified versions) are compared to assess the effectiveness of a test suite.

### Types of Mutants

- **First Order Mutants**: Created by making a single change to the original program.
- **Higher Order Mutants**: Created by making multiple changes to the original program (second order mutants have two changes, third order mutants have three changes, and so on).

Let's modify the example program from finding the largest of three numbers to calculating the sum of three numbers. We'll follow a similar approach to create mutants, including first-order and higher-order mutants.

## Original Program

Here's a simple program to calculate the sum of three numbers:

```
# Original Program
def sum_of_three(a, b, c):
    return a + b + c
```

## Mutants of the Program

### First Order Mutants

**Mutant M1:** Change the addition operator to subtraction for one of the terms.

```
# Mutant M1
def sum_of_three(a, b, c):
    return a - b + c  # Operator '+' changed to '-'
```

**Mutant M2:** Change the addition operator to multiplication for one of the terms.

```
# Mutant M2
def sum_of_three(a, b, c):
    return a * b + c  # Operator '+' changed to '*'
```

### Higher Order Mutant

Combining two first-order mutations, we create a second-order mutant.
**Mutant M3:** Change the addition operator to both subtraction and multiplication.

```
# Second Order Mutant M3
def sum_of_three(a, b, c):
    return a - b * c  # Combination of '-' and '*'
```

## Steps for Mutation Testing

1. **Initialize**:
   - Start with program $P$ and a correct set of test cases $T$.
2. **Initial Testing**:
   - Run each test case in $T$ against $P$.
   - Modify $P$ and retest if any test case fails. If no failures, proceed.

3. **Create Mutants**:
    - Generate mutants $\{P_i\}$ by making simple, syntactically correct modifications to $P$.
4. **Test Mutants**:
    - Execute each test case in $T$ against each mutant $P_i$.
    - If $P_i$ produces different results from $P$, it is "killed".
    - If $P_i$ produces the same results as $P$:
        - $P$ and $P_i$ may be equivalent.
        - $P_i$ may be killable but the test cases are insufficient.
5. **Calculate Mutation Score**:
    - Mutation Score $= \frac{\text{Number of killed mutants}}{\text{Total number of mutants}}$.
6. **Enhance Test Suite**:
    - Create new test cases if mutation adequacy is low and return to Step 2.
    - Accept $T$ as a good test suite if adequacy is high and stop adding test cases.

## Mutation Operators

Mutation operators are rules used to create mutants by making changes to the original program. These changes should be syntactically correct. Mutation operators are usually applied to ground strings, but can also be applied to a grammar, or dynamically during a derivation. Here are some common mutation operators with simple program examples:

1. **ABS—Absolute Value Insertion**:
    - **Original**: `x = 3 * a`
    - **Mutant**: `x = abs(3 * a)`

```
# Original Program
def calculate(x):
    return 3 * x

# Mutant
def calculate(x):
    return abs(3 * x)
```

2. **AOR—Arithmetic Operator Replacement**:
    - **Original**: `x = a + b`
    - **Mutant**: `x = a - b`

```
# Original Program
def add(a, b):
    return a + b

# Mutant
```

```
def add(a, b):
    return a - b
```

3. **ROR—Relational Operator Replacement**:
   - **Original**: `if (m > n)`
   - **Mutant**: `if (m < n)`

```
# Original Program
def compare(m, n):
    return m > n

# Mutant
def compare(m, n):
    return m < n
```

4. **COR—Conditional Operator Replacement**:
   - **Original**: `if (a && b)`
   - **Mutant**: `if (a || b)`

```
# Original Program
def check(a, b):
    return a and b

# Mutant
def check(a, b):
    return a or b
```

5. **SOR—Shift Operator Replacement**:
   - **Original**: `x = m << a`
   - **Mutant**: `x = m >> a`

```
# Original Program
def shift_left(m, a):
    return m << a

# Mutant
def shift_left(m, a):
    return m >> a
```

6. **LOR—Logical Operator Replacement**:
   - **Original**: `x = m & n`
   - **Mutant**: `x = m | n`

```
# Original Program
def logical_and(m, n):
    return m & n

# Mutant
def logical_and(m, n):
    return m | n
```

7. **ASR—Assignment Operator Replacement**:
   - **Original**: `x += 3`
   - **Mutant**: `x -= 3`

```
# Original Program
def increment(x):
    x += 3
    return x

# Mutant
def increment(x):
    x -= 3
    return x
```

8. **UOI—Unary Operator Insertion**:
   - **Original**: `x = 3 * a`
   - **Mutant**: `x = -3 * a`

```
# Original Program
def multiply(a):
    return 3 * a

# Mutant
def multiply(a):
    return -3 * a
```

9. **UOD—Unary Operator Deletion**:
   - **Original**: `if !(a > -b)`
   - **Mutant**: `if (a > -b)`

```
# Original Program
def check(a, b):
    return not (a > -b)

# Mutant
```

```
def check(a, b):
    return a > -b
```

10. **SVR—Scalar Variable Replacement**:
    - **Original**: `x = a * b`
    - **Mutant**: `x = a * c`

```
# Original Program
def multiply(a, b):
    return a * b

# Mutant
def multiply(a, b, c):
    return a * c
```

11. **BSR—Bomb Statement Replacement**:
    - **Original**: `x = a * b`
    - **Mutant**: `Bomb()`

```
# Original Program
def multiply(a, b):
    return a * b

# Mutant
def multiply(a, b):
    raise Exception("Bomb")
```

## Mutation Score

The mutation score is a measure of how many mutants a test suite can "kill," or detect as faulty, out of the total number of mutants. It is calculated using the following formula:

$$\text{Mutation Score} = \frac{\text{Number of killed mutants}}{\text{Total number of mutants}}$$

- **Killed Mutant**: A mutant that is detected by at least one test case.
- **Live Mutant**: A mutant that is not detected by any test case.

**Example Calculation**:
Suppose we have the following test suite results:

- Total mutants: 10
- Killed mutants: 7
- Live mutants: 3

The mutation score would be:

$$\text{Mutation Score} = \frac{7}{10} = 0.7$$

A higher mutation score indicates a more effective test suite. If there are live mutants, it suggests that the test suite needs improvement. New test cases should be added to detect these live mutants, thus enhancing the test suite's capability.

## Enhancing the Test Suite

When live mutants are found, analyze why the test suite did not detect them:

1. **Identify Missing Coverage**: Determine if parts of the code are not covered by existing tests.
2. **Add New Test Cases**: Write new test cases specifically targeting the undetected changes.
3. **Enhance Original Test Suite**: Incorporate effective test cases that detect live mutants back into the original test suite.

By following these steps, mutation testing not only evaluates the test suite but also helps in making it more robust and comprehensive.

==========

# JUnit : Framework for Unit Testing

## Overview

JUnit is a widely-used unit testing framework for Java, created by Kent Beck and Erich Gamma. It has inspired similar frameworks for other languages, collectively known as xUnit (e.g., NUnit for C#, PyUnit for Python).

## Benefits

- **Automated Testing**: Executes tests automatically, saving time and reducing errors.
- **Early Bug Detection**: Identifies bugs early in the development process.
- **Regression Testing**: Ensures new changes do not break existing functionality.
- **Documentation**: Provides usage examples, serving as documentation.
- **Improved Design**: Encourages writing testable and well-designed code.
- **Tool Integration**: Works with build tools and CI/CD pipelines, improving workflow.
- **Strong Community**: Supported by a large community and extensive ecosystem.

## Basic Steps to Test a Method with JUnit

1. **Create an Object**:
    - Instantiate the class you want to test.

- Example: `Calculator calculator = new Calculator();`

2. **Select Input Values**:
   - Choose values for the method's input parameters.
   - Example: `int a = 2;` and `int b = 3;`
3. **Determine Expected Output**:
   - Know the expected result of the method with the given input.
   - Example: `int expectedSum = 5;`
4. **Execute Method**:
   - Call the method with the chosen inputs.
   - Example: `int actualSum = calculator.add(a, b);`
5. **Verify Results**:
   - Compare the actual output with the expected output.
   - Example:

```
if ( assertEquals(actualSum, expectedSum) )
    System.out.println("Test passed");
else
    System.out.println("Test failed.");
```

This setup ensures that the `add` method works correctly for the given inputs.

## JUnit Framework Features

- **Assertions**:
  - `assertTrue(Boolean condition)`: Passes if the condition is true.
  - `assertEquals(Object expected, Object actual)`: Passes if the expected and actual values are equal.
  - `assertEquals(int expected, int actual)`: Passes if the expected and actual int values are equal.
  - `assertEquals(double expected, double actual, double tolerance)`: Passes if the difference between expected and actual is within tolerance.
  - `assertSame(Object expected, Object actual)`: Passes if the expected and actual refer to the same object.
  - `assertNull(Object testObject)`: Passes if the testObject is null.
  - `assertFalse(Boolean condition)`: Passes if the condition is false.

JUnit simplifies unit testing by providing a structured framework to write, execute, and verify tests, enhancing code quality and reliability.