Group 1: _underscore

Unit Test Coverage And Best Practice

Members

22120238 - Nguyen Minh Nguyen

22122048 - Nguyen Trong Nhan

22120256 - Ma Thanh Nhi

Lecturer

Tran Duy Thao

CONTENT

1. Unit Test Coverage
2. Minimum Coverage Level in Industry
3. Best Practices

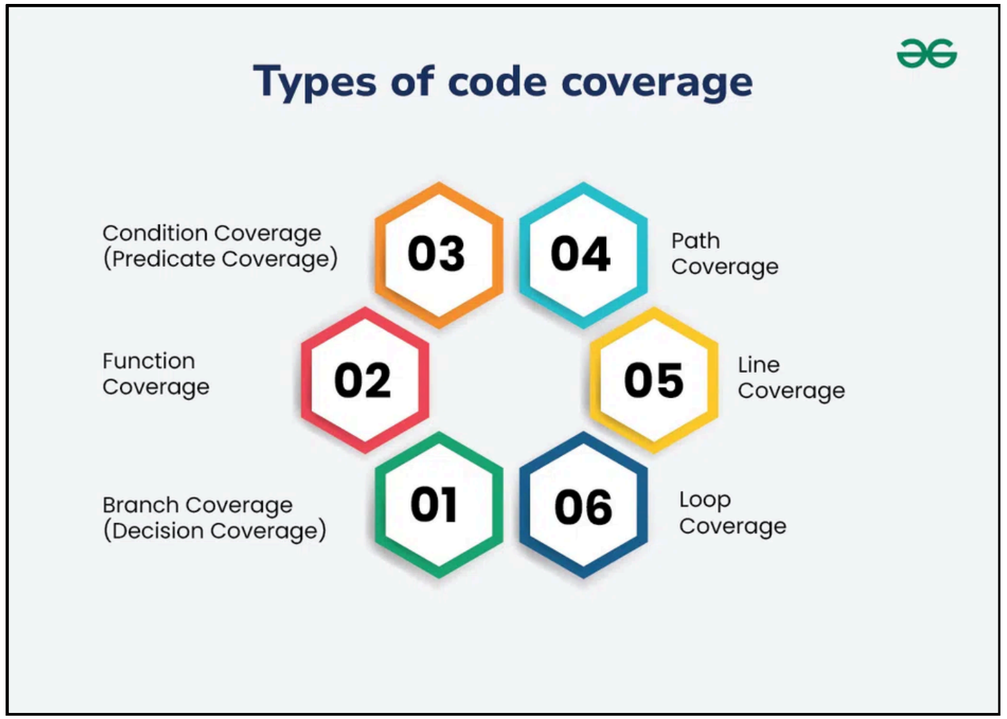
UNIT TEST COVERAGE

- ## What is Unit Test Coverage?

Unit Test Coverage (or Code Coverage) measures the percentage of source code executed by unit tests. It helps assess test quality, identify untested code, reduce defect risks, and improve software reliability. More than just a metric, it aids teams in ensuring comprehensive testing and detecting potential bugs.

Important Types of Code Coverage

1. **Line Coverage**
2. **Branch Coverage**
3. **Function Coverage**
4. **Path Coverage**



UNIT TEST COVERAGE

- ## Line Coverage

Define: Line Coverage measures the percentage of lines of code in a program that have been executed by test cases.

How it works:

- Coverage analysis tools track which lines of code are executed during test runs.
- Coverage ratio = (Number of lines executed / Total lines of code) × 100%.

Example:

```
public int calculateBonus(int salary, int performanceScore) {
    int bonus = 0;                          // Line 1
    if (performanceScore > 8) {             // Line 2
        bonus = salary * 0.2;               // Line 3
    } else if (performanceScore > 5) {      // Line 4
        bonus = salary * 0.1;               // Line 5
    }    return bonus;                      // Line 6
}
```

Total line: 6

Test case: performanceScore = 9

Line executed: 1, 2, 3 and 6

→ **Result: 4/6 = 66.67%.**

UNIT TEST COVERAGE

- **Line Coverage**

  Advantages:

  - Easy to understand and visualize
  - Suitable as a basic evaluation metric
  - Easily identifies untested portions of code

  Limitations:

  - Does not ensure all logical branches are tested
  - May miss exceptional cases

UNIT TEST COVERAGE

- **Branch Coverage**

Define: Branch Coverage measures the percentage of decision branches (such as if, else, switch, while, for) that have been executed by test cases.

How it works:

- Ensures that each decision point in the code is evaluated for both true and false cases.
- Coverage ratio = (Number of branches executed / Total number of branches) × 100%.

Example:

```java
public int calculateBonus(int salary, int performanceScore) {
  int bonus = 0;
  if (performanceScore > 8) {           // branch 1
    bonus = salary * 0.2;
  } else if (performanceScore > 5) {    // branch 2
    bonus = salary * 0.1;
  }
  return bonus;
}
```

Total branch: 2

- performanceScore > 8 (true/false)
- performanceScore > 5 (true/false)

Test case: performanceScore = 9

Branch executed: 1

→ **Result: 1/2 = 50 %.**

UNIT TEST COVERAGE

- **Branch Coverage**

  Advantages:

  - Detects more errors than line coverage
  - Ensures that all logical decisions are fully tested
  - Suitable for code with many conditions

  Limitations:

  - Does not ensure all complex condition combinations are tested
  - Does not consider all logical paths

UNIT TEST COVERAGE

- **Function Coverage**

Define: Function Coverage measures the percentage of functions/methods in the code that have been called at least once during test execution.

How it works:

- Checks whether each function in the code is called when running the test suite.
- Coverage ratio = (Number of functions called / Total number of functions) × 100%.

Example:

```
public class StudentService {
    public Student getStudent(int id) { }

    public void addStudent(Student student) { }

    public void updateStudent(Student student) { }
}
```

Total function: 3

- getStudent(int id)
- addStudent(Student student)
- updateStudent(Student student)

Test case: getStudent() and addStudent()

→ **Result: 2/3 = 66.67 %.**

UNIT TEST COVERAGE

- **Function Coverage**

  Advantages:

  - Simple and easy to achieve
  - Quickly identifies untested functions
  - Suitable for the early stages of test development

  Limitations:

  - Does not ensure the logic inside functions is fully tested
  - May create a false sense of security if relying solely on this metric

UNIT TEST COVERAGE

- **Path Coverage**

Define: Path Coverage measures the percentage of possible logical paths through a program that have been executed by test cases.

How it works:

- Identifies all possible paths from the beginning to the end of a code segment or function.
- Coverage ratio = (Number of paths executed / Total number of possible paths) × 100%.

Example:

```
public int calculateBonus(int salary, int performanceScore) {
    int bonus = 0;                          // Line 1
    if (performanceScore > 8) {             // Line 2
        bonus = salary * 0.2;               // Line 3
    } else if (performanceScore > 5) {      // Line 4
        bonus = salary * 0.1;               // Line 5
    }    return bonus;                      // Line 6
}
```

Total function: 3

- performanceScore > 8: line 1 → 2 → 3 → 6
- 5 < performanceScore ≤ 8: line 1 → 2 → 4 → 5 → 6
- performanceScore ≤ 5: line 1 → 2 → 4 → 6

Test case: performanceScore = 9

Path executed: line 1

→ **Result: 1/3 = 33.33 %.**

UNIT TEST COVERAGE

- **Path Coverage**

  Advantages:

  - Most comprehensive level of testing
  - Detects complex logical errors
  - Ensures all processing flows are tested

  Limitations:

  - Difficult to achieve high coverage in complex systems
  - The number of paths can increase exponentially
  - Time-consuming and effort-intensive to design tests

MINIMUM COVERAGE LEVEL IN INDUSTRY

- **Principle**
  - Code coverage **doesn't need to be excessively high,** it should focus on **testing all critical logic** instead

- **Rule of thumb**
  - **Google** considers **60% “acceptable”**, **75% “commendable”** and **90% “exemplary”**
  - **Industry average code coverage** ranges from **74-76%**
  - **Coverage of 80-90%** is widely regarded as **strong**

- **Need 100% ?**
  - Require **enormous** effort
  - **More feasible for:** New code patches, core business logic & safety-critical modules

```
$ coverage report -m
Name                    Stmts   Miss  Cover   Missing
-----------------------------------------------------
my_program.py              20      4    80%   33-35, 39
my_other_module.py         56      6    89%   17-23
-----------------------------------------------------
TOTAL                      76     10    87%
```

Good

MINIMUM COVERAGE LEVEL IN INDUSTRY

- **Best practices**

Horses for courses

There is no “ideal code coverage number” that universally applies to all products.

“When you have reached more than 90%, you are doing well”

100% sounds impressive but is only justified when absolutely necessary.

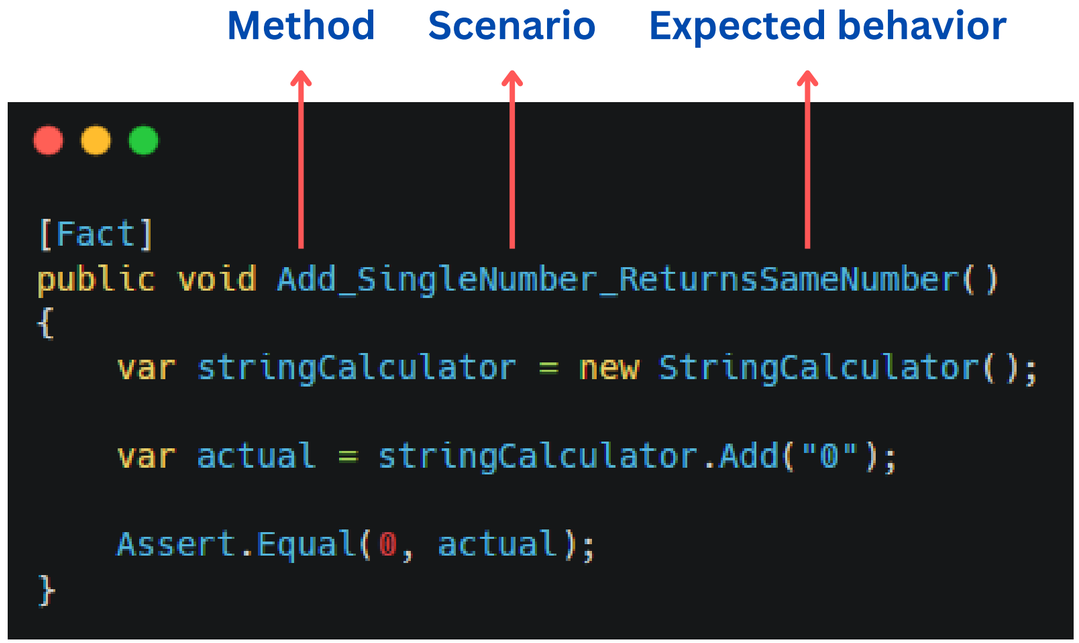
Human judgment matters more than percentages

Expert assessment of uncovered code and behaviors is far more valuable than chasing arbitrary coverage metrics.

BEST PRACTICES

- **Guidelines for Effective Testing**
  - **Avoid Infrastructure Dependencies:** Keep unit tests independent by separating them from integration tests. Avoid relying on databases or persistent storage; use mock dependencies instead for faster, more reliable testing.
  - **Follow Test Naming Standards:** Include method, scenario, and expected behavior in the name.

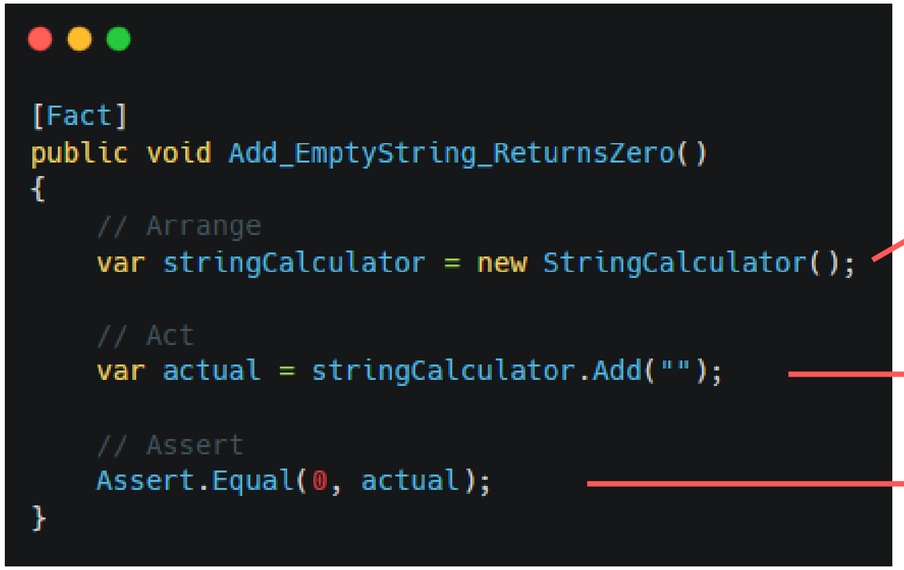
Example



BEST PRACTICES

- ## Guidelines for Effective Testing
  - **Keep Inputs Simple:** Use the simplest necessary information for verification.
  - **Avoid "Magic Strings":** Use constants instead of hard-coded values.
  - **Follow the AAA Pattern (Arrange, Act, Assert):** Arrange sets up the context, Act performs the action, and Assert verifies the outcome. Avoid multiple Act tasks within a single test.

Example



```
[Fact]
public void Add_EmptyString_ReturnsZero()
{
    // Arrange
    var stringCalculator = new StringCalculator();

    // Act
    var actual = stringCalculator.Add("");

    // Assert
    Assert.Equal(0, actual);
}
```

Arrange: Create and configure the StringCalculator instance.

Act: Call the Add method with an empty string as input.

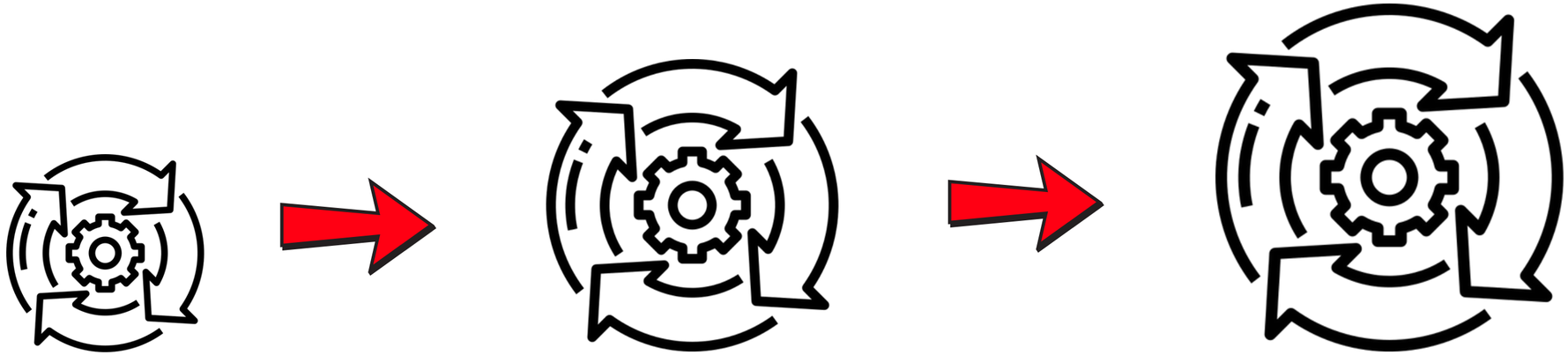
Assert: Verify that the method returns 0, as expected.

BEST PRACTICES

- **Additional Guidelines for Effective Testing**
  - **Avoid Duplicate Tests and Over-Dependence on Implementation Details:** Avoid writing repetitive tests or tightly coupling them to specific implementation details.
  - **Avoid coding logic in unit tests:** Avoid manual string concatenation, logical conditions, such as “if”, “while”, “for”, and “switch”, and other conditions. If logic is unavoidable, split tests into smaller.
  - **Use Helper Methods:** Utilize helper methods for similar objects or states.
  - **Validate Private Methods with Public Methods:** Test public methods that call private methods instead of testing private methods directly.
  - **Test Edge Cases, Error Conditions, and Happy Cases:** Ensure your tests cover edge cases like empty inputs, null values, and boundary conditions, as well as happy cases to verify expected success scenarios.

BEST PRACTICES

- **Benefits of Best Practices**
  - **Improved Clarity:** Well-structured tests are easier to read and understand.
  - **Enhanced Maintainability:** Clear tests make it easier to identify and fix issues.
  - **Increased Reliability:** Comprehensive testing ensures that code behaves correctly under various conditions.
  - **Faster Debugging:** Isolating failures becomes simpler, speeding up the development process.



REFERENCES

- GeeksForGeeks: Code Coverage Testing in Software Testing
- LaunchDarkly: On Code Coverage in Software Testing
- Google: Code Coverage Best Practices
- Microsoft: Unit testing best practices for .NET

Thank You