# Unit Testing 101

A Practical Guide for Developers

# What is Unit Testing?

## Definition

Unit testing is the practice of testing small, isolated pieces of code—usually individual functions or classes—to ensure they behave exactly as expected.
It's about verifying the smallest "unit" of logic.

## The Benefits

- **Early Bug Detection:** Catch errors during development, not production.
- **Documentation:** Tests show exactly how your code is supposed to be used.
- **Confidence:** Refactor and optimize without breaking existing features.

# Case Study: The Calculator

We will be writing tests for a simple Expression Calculator. It has three main components:

- **Lexer:** Turns strings ("2+2") into tokens.
- **Parser:** Organizes tokens into an Abstract Syntax Tree (AST).
- **Evaluator:** Computes the result of the AST.

*Our focus today is testing the **Evaluator**.*

# The Anatomy of a Test (AAA)

Most unit tests follow the **Arrange-Act-Assert** pattern.

## 1. Arrange

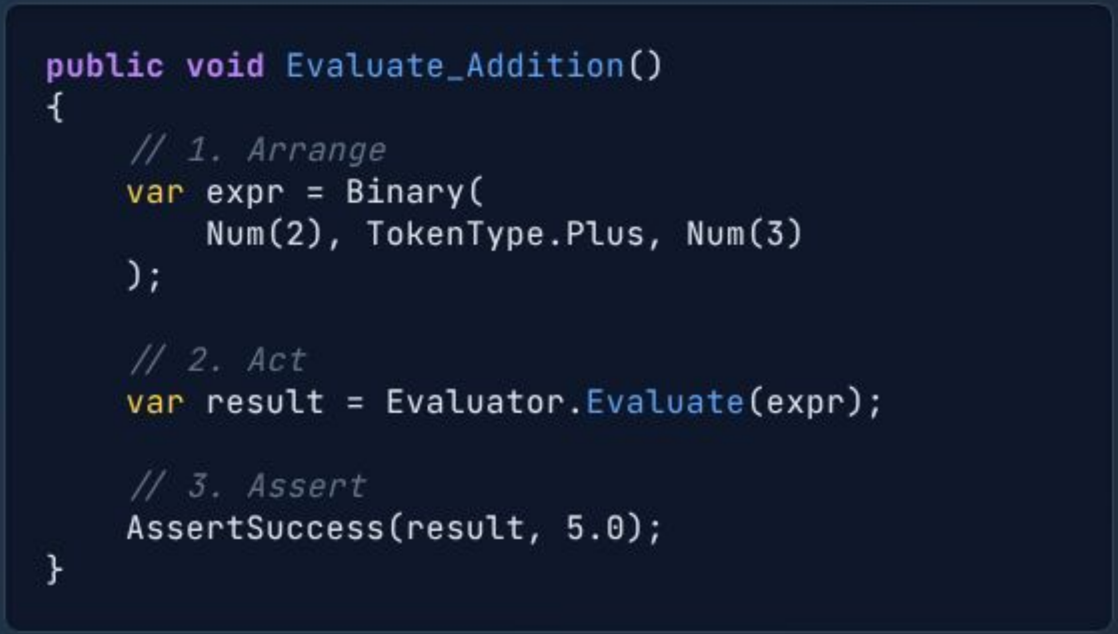
Set up the necessary objects and data.

## 2. Act

Invoke the method you want to test.

## 3. Assert

Verify that the result matches your expectations.

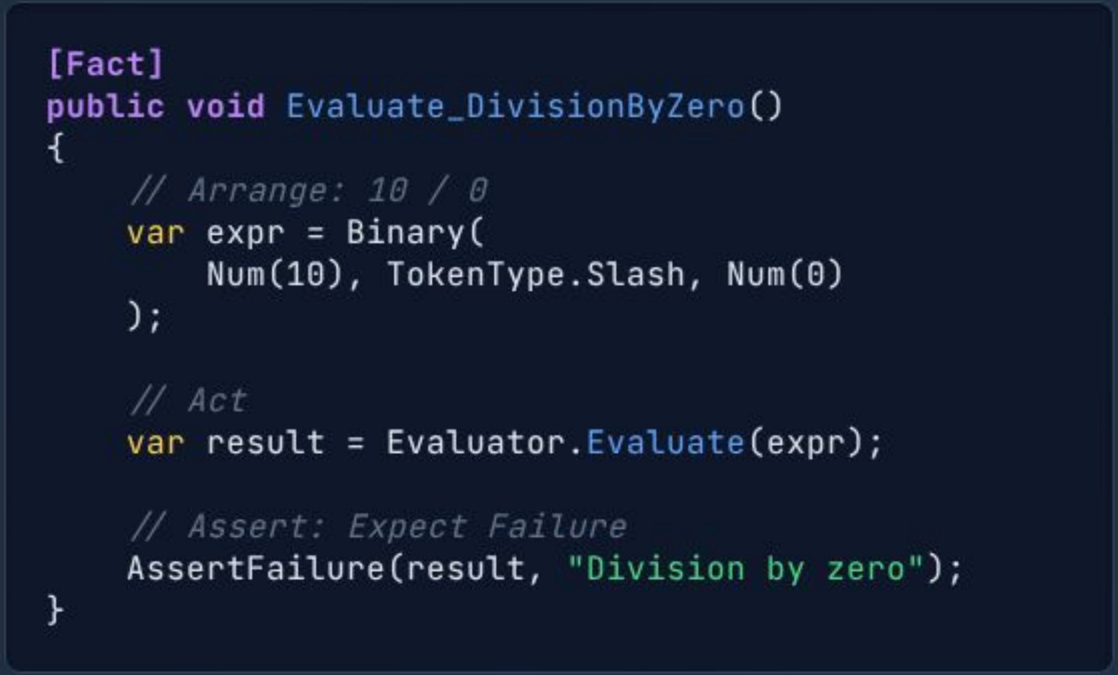


```
public void Evaluate_Addition()
{
    // 1. Arrange
    var expr = Binary(
        Num(2), TokenType.Plus, Num(3)
    );

    // 2. Act
    var result = Evaluator.Evaluate(expr);

    // 3. Assert
    AssertSuccess(result, 5.0);
}
```

# Handling Edge Cases



```
[Fact]
public void Evaluate_DivisionByZero()
{
    // Arrange: 10 / 0
    var expr = Binary(
        Num(10), TokenType.Slash, Num(0)
    );

    // Act
    var result = Evaluator.Evaluate(expr);

    // Assert: Expect Failure
    AssertFailure(result, "Division by zero");
}
```

## Expect the Unexpected

Tests aren't just for the "Happy Path." You must test invalid inputs and error states.

Here, we explicitly verify that dividing by zero returns a **Failure** result, preventing the app from crashing.

# Isolation is Key

## Test One Thing at a Time

Notice that in EvaluatorTests.cs, we manually build the **Expression** objects.

We **do not** use the Parser or Lexer in these tests.

> *Why?* If the Parser is broken, we don't want our Evaluator tests to fail. This helps us pinpoint bugs instantly.

# Writing Testable Code: Interfaces & DI

## The "Hard Dependency" Problem

If your code instantiates dependencies directly (e.g., new Database()), you cannot unit test it without a real database.

## The Solution: Dependency Injection

Ask for dependencies in the constructor via **Interfaces**. This allows you to pass in "Fake" or "Mock" objects during testing.

❌ Bad: Tightly Coupled

```
class Processor {
  private Logger _log = new FileLogger();

  void Process() {
    _log.Write("Done"); // Can't test this!
  }
}
```

✅ Good: Dependency Injection

```
class Processor {
  private ILogger _log;

  // Ask for the interface!
  public Processor(ILogger log) {
     _log = log;
  }
}
```

# Test Doubles: Know Your Tools

"Mock" is often used as a generic term, but there are differences.

### Stub

Returns hardcoded data. It has no logic.

*Example: A database stub that always returns "User Found".*

### Mock

Verifies behavior. It checks if a method was called.

*Example: Verifying that "SendEmail" was called exactly once.*

### Fake

A working implementation, but simplified.

*Example: An In-Memory Database instead of SQL Server.*

# Unit vs. Integration Testing

## Unit Tests

**Scope:** Single Class (Evaluator).

**Dependencies:** Mocked/Stubbed.

**Speed:** Extremely fast (milliseconds).

**Purpose:** Verifies internal logic.

## Integration Tests

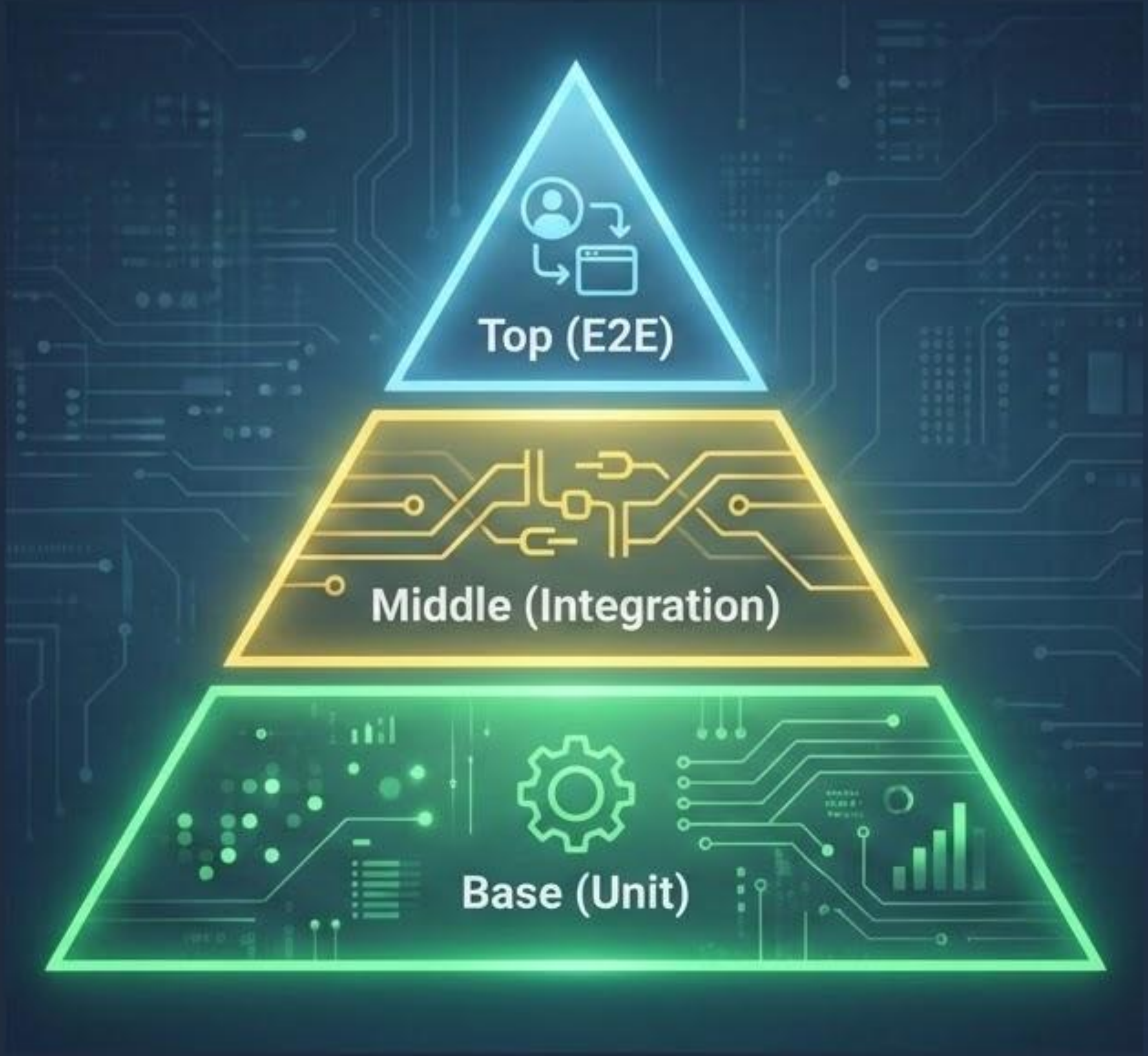
**Scope:** Full Pipeline (Lexer -> Parser -> Evaluator).

**Dependencies:** Real components.

**Speed:** Slower.

**Purpose:** Verifies wiring.

# The Test Pyramid



## Why this shape?

- 🟢 **Base (Unit):** Fast, cheap, and pinpoint bugs instantly. We want *lots* of these.
- 🟡 **Middle (Integration):** Verify connections. Slower to run.
- 🔵 **Top (E2E):** Slow and brittle. Use sparingly for critical user flows.

# The TDD Cycle

Test-Driven Development (TDD) follows a simple rhythm:

### 1. Red

Write a test that fails. This confirms the test is valid and the logic is missing.

### 2. Green

Write just enough code to make the test pass. Do not worry about perfection yet.

### 3. Refactor

Clean up the code. The test ensures you don't break anything while improving it.

# Common Pitfalls

🚫

### Logic in Tests

Avoid `if` statements or `loops` in tests. If your test has logic, who tests the test?

🚫

### Overspecification

Don't test *how* the code works (implementation), test *what* it does (behavior). Avoid testing private methods.

🚫

### The Mockery

Don't mock everything! If you mock every single class, you aren't testing the real system anymore.

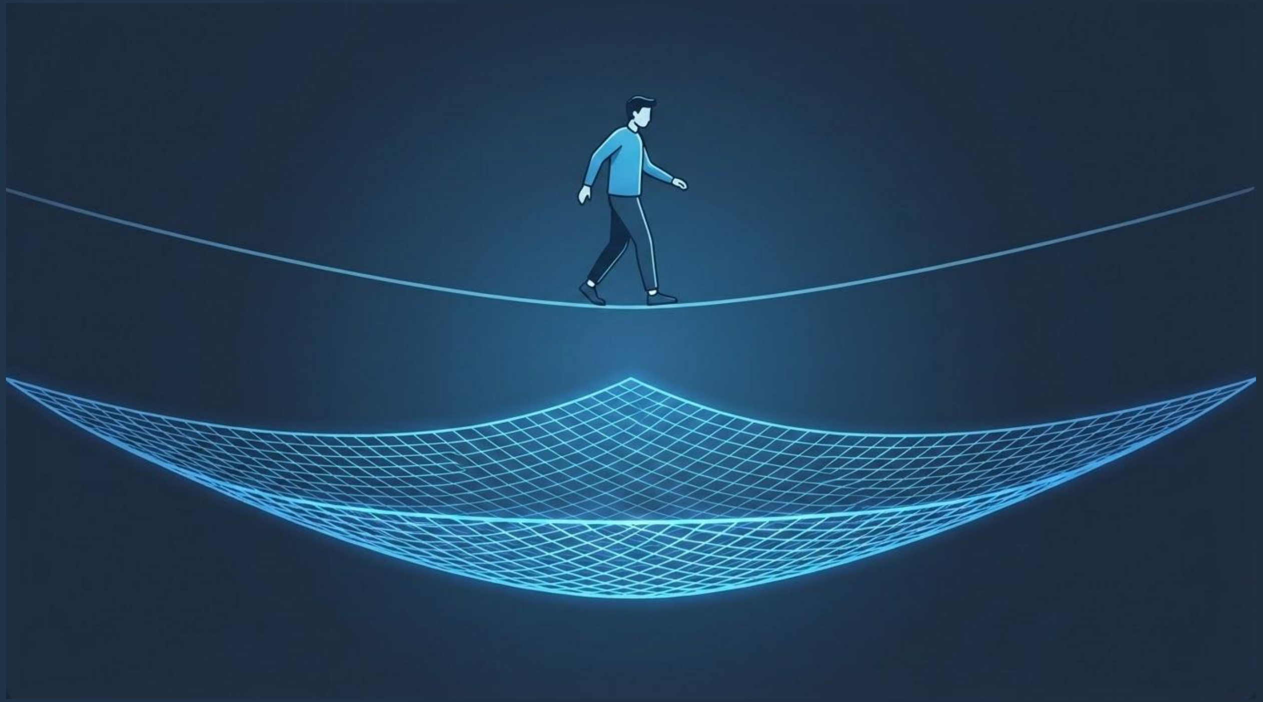
# Your Safety Net

## Refactor with Confidence

Imagine you want to optimize how EvaluateBinary works.

Without tests, you're crossing a tightrope without a net. One mistake crashes the app.

**With tests:** You make the change, run the tests, and if they pass, you know you haven't broken anything. Tests allow code to evolve.



# Best Practices (F.I.R.S.T.)

### Fast

Tests should run quickly so you can run them often.

### Isolated

Tests should not depend on each other or external systems.

### Repeatable

They should produce the same result every time.

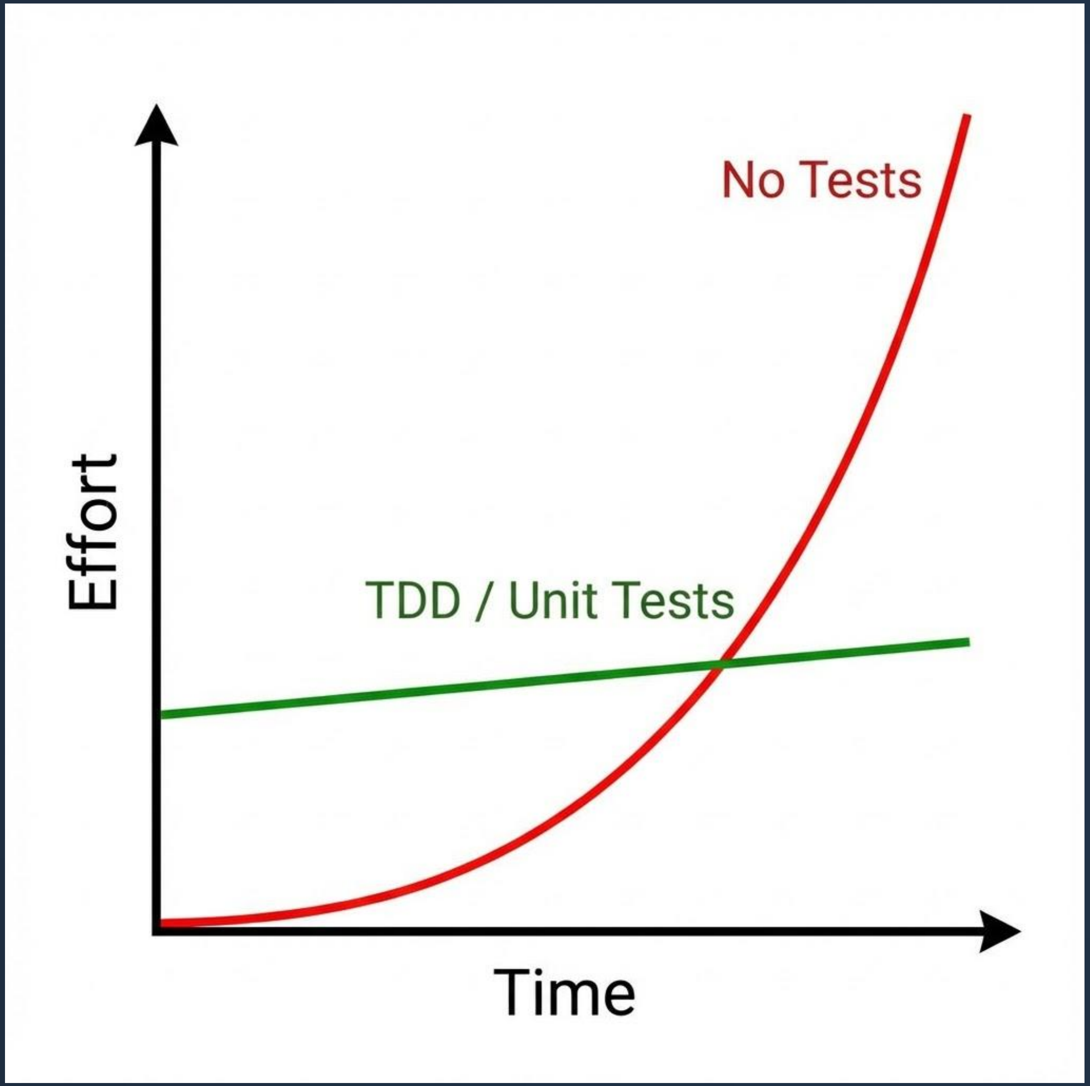
### Self-Validating

Pass or Fail. No manual checking of output.

### Timely

Write them just before or with the production code.

# The Cost of Quality



## Is it worth it?

Many developers think writing tests "slows them down."

### Without Tests

Initial speed is high, but as the codebase grows, **maintenance becomes a nightmare**. Bugs accumulate, and every change breaks something else.

### With Tests (TDD)

Higher initial investment to write the tests. However, development speed **remains constant** over time because you spend less time debugging and fixing regressions.

# 💬Questions?

Let's discuss Unit Testing!

Review the Code:

`Calculator/Tests/EvaluatorTests.cs`