# Break It, Make It:

# Sync Development with Tests

Second Edition

**Preview Copy Notice**

This document is provided as a **preview copy** for review purposes only.

- All content is **subject to author approval** and may be revised prior to final publication.
- Any information contained herein may be **redacted or removed** in the event of misuse, unauthorized distribution, or modification.
- Redistribution, reproduction, or commercial use without prior written consent from the author is strictly prohibited.
- Feedback and constructive comments are welcome to help improve the final edition.

Thank you for respecting the integrity of this work.

# Preface

Welcome to Break It, Make It second edition. I as author hope to improve each version as reader feedback helps to nudge in right direction and tone. This publication is created with intention to create a package deal for young developers who are dipping toes to test waters.

This guide includes all starter information that are industry standard, easy to digest on practice and a small handbook to enable you in projects.

You will learn from small scale projects that are easy on mind to help budding minds focus. You will learn complex topics as you progress and solve growing complex combinations for terms, project code. Along with foundation theory, you are invited to learn another approach for writing tests for existing code.

Once you have build muscle and routine for tests, you are invited to test skills in real life projects.

## Prerequisites

In order to get most out of book:

1. Basic knowledge of C#
2. Ability to create simple console projects

## Good to have

You can accelerate from foundation to more responsive growth with:

1. Basic knowledge of S.O.L.I.D principles , Law of Demeter
2. Experience in creating projects.
3. A local working setup including IDE for C# development with .NET 8 or newer, version control

Understand diagrams to help visualize code and interactions. Keep this along with your daily work to help with new projects and legacy projects. Maintain your personal journal to monitor your growth.

# About Author

Abhishek Chaitanya Bhattacharya is a software engineer by profession with 12+ years experience.He has worked for clients in U.S. government contracts, e-commerce, finance , payment gateway along with small to medium size IT service companies. He is a a computer engineer graduate from Mumbai university.

He is currently working with technologies such as ASP.NET Core, C#, SQL Server, React, and WebAPI.

Abhishek carries a deep interest in software architecture and personal research in TDD application. Abhishek intends to create a book he never had as young developer while going through difficult projects. He intends to create a refined trustworthy resource for fellow developers. He collected notes over years for improving code over time.

Author believes Test driven development is an important milestone but not the destination to conclude. He has included chapters to help learn topics precise and expand over time. His approach treats automated tests as the bedrock of clean architecture. This publication discusses tools like NUnit, Moq, and CI/CD pipelines to validate ideas and accelerate feedback

Abhishek aims to create a thoughtful, test friendly culture. In personal time, he delves into creating documentation, UML diagrams for learning architecture.

# Copyright Notice

© 2026 Abhishek Chaitanya Bhattacharya. All rights reserved.

This publication is protected under international copyright laws. No part of this book may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the author, except for brief quotations in reviews, articles, or academic citations.

Prohibited actions include:

· Photocopying, recording, or other electronic or mechanical methods

· Distribution or transmission in any form

· Unauthorized commercial use, redistribution, or modification

The content, diagrams, and code samples within are the intellectual property of the author and intended for individual professional use. Unauthorized modifications are strictly prohibited.

This book is intended for educational and reference purposes only. While every effort has been made to ensure accuracy and best practices, the author assumes no liability for any direct or indirect outcomes resulting from the application of the content herein.

For permissions and licensing inquiries, contact: abhishekcbhattacharya@gmail.com.

# Acknowledgments

I own gratitude towards my colleagues and dedicated friends who have participated in process of providing crucial feedback on technical terminologies, developer biases that undermine knowledge gap. This feedback led to perceive interpretation mismatch between reader and author’s intent.

I have used multiple open source projects to create documentation , Visual Studio Code to execute source C# code , Graphviz used to create diagram for explaining concepts. There are Nuget packages like NUnit, Moq, and Playwright that help make TDD a practical approach

I acknowledge dedicated family members for showing their patience, understanding while undergoing demanding drafts and reviews.

I deeply acknowledge readers your contributions for nudging me as author to right direction and thought process.

# Errata & Reader Feedback

I as author have tried to best of available resources and time to provide knowledge helpful for reader in simple language and less technical jargon. All information is scrutinized for validity and useful nature for contributing to knowledge. As part of feedback process and available time, we are making sure to continue pushing out improvements.

Please share your critique as a reader about chapters, topics that are under discussed or complicated in nature that can be described in simple language.

While including your feedback, please include chapter title, topic and details what was expected to help address your concern

You can contribute by sharing:

1. · Typographical or grammatical corrections
2. · Your questions about specific examples or concepts
3. · Personal suggestions for clearer phrasing or alternative approaches
4. · Workable fixes or enhancements to code samples

Please send your submissions to email: **abhishekcbhattacharya@gmail.com**

# Chapter wise topics

Chapter 1: Test-Driven Development Essentials
Chapter 2: Framework v/s Libraries
Chapter 3: SOLID principle for clean code
Chapter 4: Testing Types and Strategies
Chapter 5: Tools for the Trade
References

# Chapter 1: Test-Driven Development Essentials

## Learning outcomes

1. Thought process for Red – Green – Refactor cycle and implications.
2. Psychology related to small test and refactoring.

## 1.1 Thoughts on Red – Green- Refactor cycle

Test driven development(TDD) has a core process to:

1. Start writing a failing test(red),
2. Making test pass (green)
3. Cleaning code to make it maintenance friendly(refactor).

The iteration process is followed to make code improvements and reliable (Beck, 2003).

### 1.1.1 Type of tests

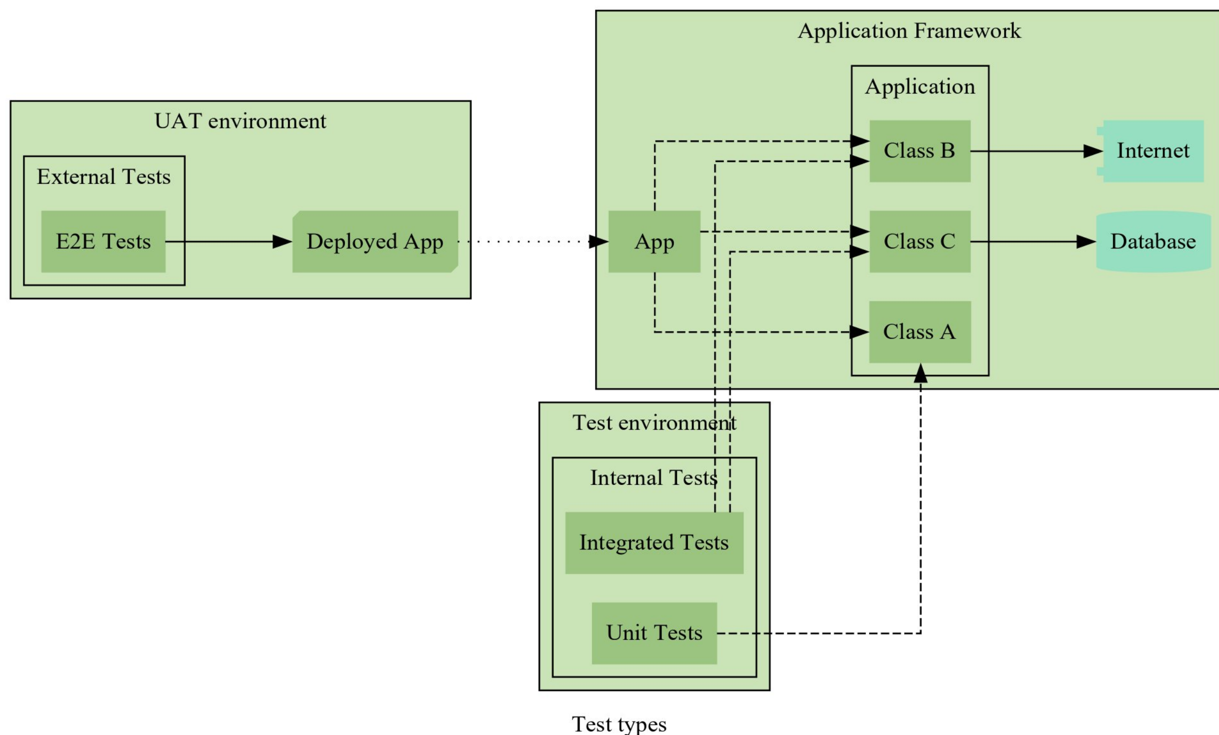



Fig 1.1.1.a – Types of tests

TDD cycle asks to create tests but does not force any test type recommendation. A new project will have unit tests, integration tests at early stages. E2E tests are created towards end of development. For Existing projects, E2E tests are insurance created for starting Tests on code.

1.1.1.1 Black box testing

Let assume you are testing a black-box under worst scenarios. A black-box means it does not share internal working details to infer information.  A good example is lock picking with number combination.

In case you are testing UI or front-end code, you will validate as user information available. You interact using inputs and receiving information or track changes for output based on testing outside in approach. These tests are classified as End to End tests. Tests are supported using Selenium or Playwright for recent times.

These E2E tests are brittle in comparison. It provides external checks against legacy projects that are requiring maintenance or upgrade where maintaining functionality to existing features are important.

1.1.1.2 Glass box testing

As a developer, you are having access to code, creating tests and executing tests  You are in easier position to compare information input, output, state changes or check interaction.

You are setting up internal details for executing internally and validating rules externally.

When you are doing tests only to validate implemented rules as business logic and no interacting components then you are conducting unit tests.

When you verify interaction with actual external components along with implemented rules then you are conducting integration tests.

## 1.1.2 Writing tests early

Given project deadlines, you are expected to release feature update for software code that does not have ideal guidelines to begin with.

You have an ice ball rolling down hill known as technical debt growing with time unattended.You create rules that are existing ideas.

The ability to run tests using automated tools requires to write tests first.

Once you are invested and create successful tests, you are able to have accelerated feedback in comparison to manual tests.

### 1.1.2.1 Define explicit rules

The stringent process of writing tests is to make sure you are converting all implicit assumptions to explicit inputs. When there are explicit rules available along with source control, you create a opportunity for code changes with less mental burden.

Hardware testing kits are easiest examples to make clear decision for faulty hardware.

#### 1.1.2.2 Whiteboard mapping ideas

A new idea you had thought will have underlying flow that has to converted to bare bone structure. Once you have base interpretation of idea to code. You have rule based system to evaluate success.

#### 1.1.2.3 Multiple attempts for feedback

In attempts to create tests before code, you end up evaluating rules to rely for future version. You will require iterations of creating rule and valid information, easy validation test to set failure. Each iteration gets easier than before.

You can stop iteration when outcomes start trimming out like a log chart. You can end up in gold platting territory when you overdo. Please manage it till it exceeds just a little above required.

### 1.1.3 Testing at team level

A developer owns task of writing tests for covering self created code ideally. This is not officially recognized job description but a part of professional work ethics. As individual developer add new tests to suite/ collection, tests are executed overall project for each individual. If you have common functionality, you will need to collectively understand common business rules.

## 1.2 Writing a failing test first (Red)

It is first fundamental step regarding test implementation. Writing tests are about converting implicit information to explicit documentation. When you write tests you have make clear intention for validating information.

When you are writing tests with no implementation or minimal implementation you are creating a clear distinction between assumption and actual scenario. The ability to measure gives ability to verify improvement. In case your tests are valid in rules with no or minimal implementation, you are doing fundamental mistakes.

## 1.3 Project timelines

A project using TDD and non TDD approach have different timelines.

For Non TDD projects, you are creating a dependency for knowledge on experienced members. New starting project members have to deal with knowledge biases and assumptions hard to quell.

You can find that delivery timelines start to falter, with more features to support.

You do not require TDD for quick and dirty interpretation for proof of concept projects.

You apply TDD in case of long term maintenance goal. Each iteration has cost of writing tests. You will require to reserve 2 to 4 weeks for automation setup, test goals.

## 1.4 Psychology related to small test

### 1.4.1 Minimum viable product approach

When you have focused on testing one feature at a time, it adds value while maintaining resources. You require to add rules coverage one by one. Create source code specific to your case.Organize by common origin

### 1.4.2 Heavy cost on top down approach

You are given a suggestion that is ambiguous enough to not describe or force validation approach.

A class includes encapsulated functionality and members to hold information. A class can include nested references of interface / class to indicate external interactions or common utilities.

When you create a dependency relation among class using and assigned implementation you can visualize a node tree structure.

Class at the bottom tier will be easier to test. As you write tests for higher levels, you have managed all required dependencies.

### 1.4.3 Making a test pass (Green)

**Identify outcome**

When you created a new test, you work on validation rules that require to check response and state changes. In racing analogy, you identify track for run.

**Create outcome**

Making a test pass is second step in iteration after failed test benchmark. You make sure to create rules based on whiteboard phase and execute. You are free to brainstorm implementation details. Once you have achieved success test benchmark.

In racing analogy, you complete race on track safely.Try adding multiple test data versions to enforce a legitimate rules implementation.

### 1.4.4 Cleaning code after success test (Refactor)

When you have created a tests, achieved success test while following TDD process. As you complete two steps , you execute refactor step as part of cleaning up activities.

**Need for separate cleanup step**

Developers vary from quick and dirty unreadable code to gold polish with unnecessary patterns. You should respect actions over theory.

**Looking for repetitions/patterns**

Identify duplicate code and replace with references. Apply commonly used design principles if repetition exceeds 3 count. When you separate refactoring after you have achieved results , you have avoided gold plating trap. Gold plating is a developer bias where they continue to be obsessed with complex patterns with no clear cost analysis.

**Refactoring**

Refactor is a common developer activity that looks for source code modification for better readability with no change in behavior. When you delay it, you can lose knowledge fast. If you do it normally, you integrate it the stride.

If refactoring is a step taken specially, you have created an obstacle for TDD cycle.

These steps apply for unit tests, integration tests when you have modification rights for tested source code.

S.O.L.I.D principle and Law of Demeter have proven success. Check your use case before apply.

## 1.5 Existing projects challenge

On paper, TDD has claimed to start projects with Tests. For real life scenarios, idea has to exist on whiteboard to brainstorm.

For legacy projects, creating a tests for existing code do not occur naturally.

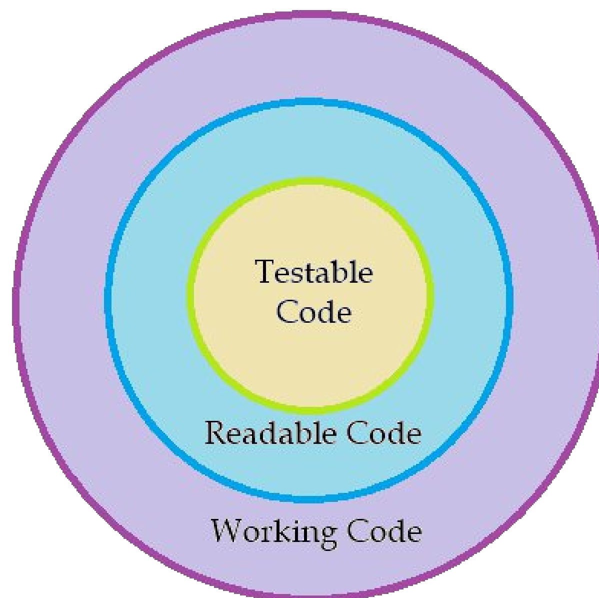



Fig 1.5.a – Venn diagram highlighting coding characteristics

# 1.6 TDD example

Problem : Create a class Calculator which performs addition of two numbers. Use TDD method for implementation.

Statement: Given you are accepting two numbers, when you perform add operation then it should return sum of inputs.

Create test before any existing implementation of class.

## 1.6.1 Create a failing tests - Red

```
public class TDD_Beginner
{
    private Calculator _calculator;
    [SetUp]
    public void Setup()
    {
        _calculator = new Calculator();
    }

    [TestCase(1, 2, 3)]
    [TestCase(3, 4, 7)]
    public void AddSumOfTwoNumbers(int a, int b, int expectedSum)
    {
        //Arrange - setup
        //Act
        int actualSum = _calculator.Sum(a, b);
        //Assert
        Assert.That(actualSum, Is.EqualTo(expectedSum));
    }
}
```

Fig 1.6.1.a – A common example of N Unit 3 test

You are validating rules externally. This code snippet highlights common structure to tests.

[SetUp] attribute executes before every [Test] / [TestCase] attribute marked test method. [TestFixture] attribute helps detect class for test runner.

## 1.6.2 Implement class under test – Green

```
public class Calculator
{
    public int Sum(int a, int b)
    {
        throw new NotImplementedException();
    }
}
```

Fig 1.6.2.a – Class under test

Create simplest implementation to achieve response successful in rule validation. Adding is a operation with response based on formula.

### 1.6.3 Improve code implemented - Refactor

```
public class Calculator
{
    public int Sum(int a, int b)
    {
        return a + b;
    }
}
```

Fig 1.6.3.a – Class implemented under test

Once a implementation is complete, do a refactor for improvements. Apply S.O.L.I.D principle along with Law of Demeter to help restructure code. One side effect of refactoring is it allows to find new dependencies deeply nested.

## 1.7 E2E Tests

These are tests considered as outside in approach. Tests ask you to measure behavior. As you execute playwright test against a system going though UI, API calls. You have to check details that do not maintain in long term. A small change in UI can break assertion details.

In code example , [Setup] [TearDown] attributes are highlighting common tests code.

```
public class LoginE2ETests
{
    private IBrowser _browser;
    private IPage _page;
    private IPlaywright _playwright;

    [SetUp]
    public async Task Setup()
    {
        _playwright = await Playwright.CreateAsync();
        _browser = await _playwright.Chromium.LaunchAsync(new BrowserTypeLaunchOptions
        {
            Headless = false // Set to true for headless mode
        });

        var context = await _browser.NewContextAsync();
        _page = await context.NewPageAsync();
    }
```

```csharp
    [Test]
    public async Task Login_Should_Succeed()
    {
        //Arrange - SetUp
        //Act - multiple steps
        // Navigate to login page
        await _page.GotoAsync("https://example.com/login");

        // Fill in credentials
        await _page.FillAsync("#username", "yourUsername");
        await _page.FillAsync("#password", "yourPassword");

        // Click login
        await _page.ClickAsync("#loginButton");

        // Wait for dashboard URL
        await _page.WaitForURLAsync("https://example.com/dashboard");

        // Assert login success by checking welcome message
        var welcomeText = await _page.InnerTextAsync("#welcomeMessage");
        Assert.IsTrue(welcomeText.Contains("Welcome"), "Login failed: Welcome message not found.");
    }

    [TearDown]
    public async Task Teardown()
    {
        await _browser.CloseAsync();
        _playwright.Dispose();
    }
}
```

Fig 1.7.a – E2E test using Playwright

In Test, you are executing as user interacts with system. Create multiple criteria for validate page information. Ensure dependencies are installed

1. Microsoft.Playwright
2. NUnit
3. NUnit3TestAdapter

## 1.8 Exercises

Each exercise level allows to test a fundamental concept

### GitHub Repository

Repo URL: https://github.com/abhishekcbhattacharya-svg/BreakItMakeIt_Exercises_v2

Folder: *BreakItMakeIt_Exercises/Chapter_01_TDDFundamentals*

**Steps**

git clone https://github.com/abhishekcbhattacharya-svg/BreakItMakeIt_Exercises_v2.git

cd BreakItMakeIt_Exercises_v2

Use VS Code or Visual Studio 2022 C# IDE

**Level 1: TDD_Beginner**

Topics:

1. Write your first unit test,
2. Focus on Red-Green-Refactor cycle

Challenge: Create method implementation with help of provided test information.

**Level 2: TDD_Intermediate**

Topics:

1. Handle edge case
2. Create logic to handle range of input and combination

Challenge: Create method implementation with help of provided test information and helper class as real life scenario.

**Level 3: TDD_Advanced**

Topics:

1. Implement Playwright code
2. Interact with website input
3. Check interaction response for validation

Challenge: Create E2E tests for existing website to assess behavior validation.

## 1.9 MCQ Questions

1. In the Red–Green–Refactor cycle, what does the “Red” phase represent?

A. Writing minimal production code
B. Writing a failing test before implementation
C. Cleaning up duplicate code
D. Reviewing test coverage

Answer: B Explanation: The Red phase is about writing a failing test to capture requirements before coding.

2. Which type of test validates business logic without external dependencies?

A. Unit test
B. Integration test
C. End-to-End test
D. Smoke test

Answer: A Explanation: Unit tests focus on isolated business logic without external components.

3. Why is writing tests early emphasized in TDD?

A. To reduce compilation errors
B. To convert implicit assumptions into explicit rules
C. To avoid using external libraries
D. To eliminate the need for refactoring

Answer: B Explanation: Writing tests early ensures assumptions are documented as explicit rules, reducing ambiguity.

4. What is the main risk of over-refactoring in TDD?

A. Increased compilation time
B. Entering the “gold plating” trap with unnecessary complexity
C. Reduced test coverage
D. Loss of framework compatibility

Answer: B Explanation: Over-refactoring can lead to gold plating, where developers obsess over patterns without cost analysis.

5. In team-level TDD, what is expected of each developer?

A. To only test framework code
B. To write tests covering their own code contributions
C. To avoid adding new tests to the suite
D. To rely solely on senior developers for test writing

Answer: B Explanation: Each developer is responsible for writing tests for their own code, contributing to the collective suite.

6. Which project type benefits most from TDD?

A. Quick proof-of-concept projects
B. Long-term maintenance projects
C. One-time demo projects
D. Throwaway prototypes

Answer: B Explanation: TDD is most effective for projects with long-term maintenance goals, where automated tests provide lasting value.

7. In the calculator example, what is the first step before implementing the class?

A. Write a passing test
B. Write a failing test for the add operation
C. Refactor existing code
D. Create integration tests

Answer: B Explanation: TDD requires writing a failing test first to define expected behavior before implementation.

8. Which testing approach validates behavior from a user’s perspective using tools like Playwright?

A. Unit testing
B. Integration testing
C. End-to-End testing
D. Regression testing

Answer: C Explanation: End-to-End tests simulate user interactions and validate system behavior externally.

9. What psychological benefit comes from writing small tests in TDD?

A. Faster compilation
B. Reduced mental burden and easier iterations
C. Avoiding dependency injection
D. Eliminating framework usage

Answer: B Explanation: Small tests reduce cognitive load and make iterations easier to manage.

10. Why is refactoring considered a separate step in the TDD cycle?

A. To add new features
B. To improve readability without changing behavior
C. To remove all tests
D. To speed up compilation

Answer: B Explanation: Refactoring improves design and readability while preserving behavior, ensuring clean architecture.

# Chapter 2: Framework v/s Libraries

## Learning outcomes

1. Compare framework and library differences.
2. Examples in implementation for framework v/s library approach.

## 2.1 Framework

A framework is a system that allows to host functionality providing information and handles interaction in lieu of trust,documented rules

This is a foundation on which code is executed. User interaction with framework requires to accept rules.A framework exposes certain code to implement your functionality.

Ex: A house provides sockets for power consumption

### Hollywood principle

This is common text stated as "*Don't call us, we'll call you*." used for framework. You write code but do not know time of execution, hence you delegate execution to framework.

You have an user Interface having a button. Button does not anything on its own. You will register a delegation to execute your code on event.

1. **Publisher**: UI Framework will know an event, but does not know code to execute.
2. **Subscriber**: Code will not know origin of information, but execute business rules.

### Opinionated structure

A framework is created with a template that dictates or limits execution within a process.You will always have debates on flexibility and structure. There is no perfect agreement.

If you have uncontrolled flexibility, you have lost control and can have undocumented behavior. If you choose over structured, you will end up creating large boilerplate code for small behavior

Projects have naming conventions, folder structure and system to use common template  They have to be designed for beginner friendly and professional expertise.

You can compare frontend framework like Angular, React or Vue.

1. Angular: Enforces strict architecture, conventions, and tools
2. React: Leaves most decisions to developers, only UI-focused
3. Vue: Provides defaults but allows flexibility

A successful framework makes sure to communicate rules and lead to focused work while holding ability to use additional library for required task.

## 2.2 Library

A library is system that provides functionality as utility, can require information to perform and used in plug and play manner. You are in control of executing library based functionalities. It is focused on specific task.

A library successfully integrates to a project without any limitations.A framework can include multiple libraries and work independently

A library is comparable to shareable, reusable equipment that is working within range of infrastructure.

Ex: An electric appliance is connecting to socket for power input.

A library operates with consistent behavior irrespective of environment. A test framework executes code on your behalf, you are required to mimic framework behavior

**Library examples**

1. JSON serializers(System.Text.Json)
2. HTTP clients(HttpClient)
3. Utility collections(Linq, Math)

All development code executes within one designated framework. We develop a library that is executed within framework. One framework supports one or more libraries.

In C# , you implement class library project for creating class, interface

## 2.3 What to focus own in tests?

A developer intends to write tests for their developed code. You are to test implemented rules or logic. When you test tools itself, it is loss of direction. The term tools is used for framework specially, library are used for individual use case.

You are assuming 3 types of interactions –

1. business rules / logic
2. external component interactions - database / email / Otp,
3. framework code - navigation , model binding

For testing, we use Testing framework to execute test code based on Arrange, Act and Assert. A user developed code library is executed within test cases.

You may call code directly for framework specific code. Please note to execute application framework code, you may require additional code to mimic framework behavior in test environment.

As a developer , you are never testing framework code during unit, integrated testing. A developer is responsible to test behavior.

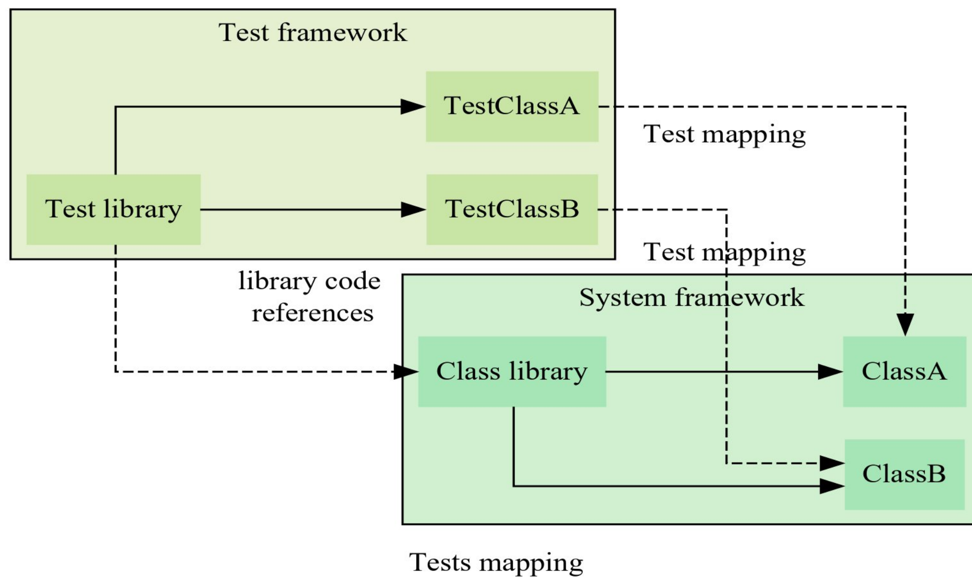



Fig 2.3.a – Mapping between Test framework and library

A class library holds ability to interact with variety of frameworks. In Fig 2.3.b you can view a class library implementation interacting with possible .Net frameworks.

A class library holds class, interface for describing structure and implementation. For TDD approach, you have a test ready before implementation for class.

In Fig 2.3.a you can see mapping that starts to appear for test class for testing class

As author, I suggest to think logic implementation separate as class library project.

Image describes frameworks like ASP.NET Web Form , .Net Core  MVC, .Net Core WebAPI and Test framework.

Older framework like ASP.NET WebForm is based on MVP pattern, UI is event driven. You are stuck with writing a dedicated class library for execution.

Newer framework like .Net Core MVC, .Net Core WebAPI have better support for unit testing. Create Controller class with DI setup. Model binding, validation is simpler.

In long term application, you can create library for logic / business rules. All nested external interactions are handled in separate implementation.

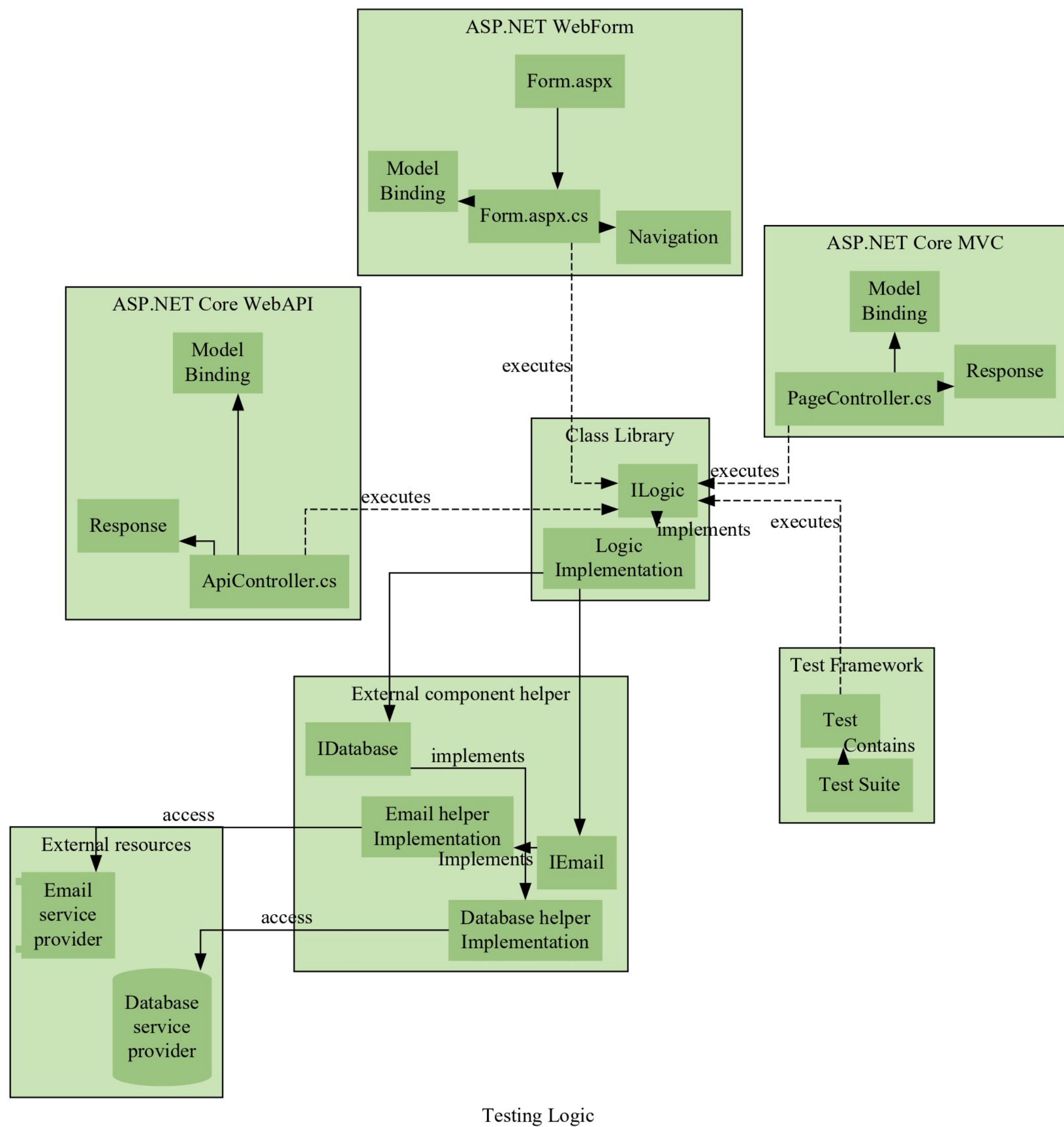



Fig 2.3.a – Interaction between Framework and library

## 2.4 Logic implementation

Model validation is a concept where we check information for object. If object is valid information it is processed further for implementation.

```
public class UserModel
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

Fig 2.4.a – class representing user information

**Framework**

MVC framework provides Data Annotations attributes for validation along with different combination.

```
public class UserModel
{
    [Required(ErrorMessage = "Name is required")]
    [StringLength(50, ErrorMessage = "Name must be less than 50 characters")]
    public string Name { get; set; }

    [Range(18, 60, ErrorMessage = "Age must be between 18 and 60")]
    public int Age { get; set; }
}
```

Fig 2.4.b – Class with properties annotated with attributes

Required attribute forces error if property is not assigned.

Range attribute forces error if property assigned is less than minimum and greater than maximum number specified.

```
public class UserController : Controller
{
    [HttpPost]
    public IActionResult Create(UserModel model)
    {
        if (!ModelState.IsValid)
            return BadRequest(ModelState);

        return Ok("User created");
    }
}
```

Fig 2.5.c – Controller and model validation

MVC framework will take care of class validation behind the scenes. This is a gap faced by code under test framework. Check ModelState.IsValid property to redirect or continue for operation

```
ModelState.AddModelError("", "Name is required");
```

Add error information during tests to mimic framework testing behavior under test framework like NUnit 3.

**Library**

FluentValidation is a nuget package that allows to decouple validation rules from class.

```
//FluentValidation
public class UserModelValidator : AbstractValidator<UserModel>
{
    public UserModelValidator()
    {
        RuleFor(x => x.Name).NotEmpty().WithMessage("Name is required");
        RuleFor(x => x.Age).InclusiveBetween(18, 60).WithMessage("Age must be between 18 and 60");
    }
}
```

Fig 2.5.c – Class validator with Fluent validation code

Create rule expression for property with rule and corresponding error message.

```
var validator = new UserModelValidator();
ValidationResult result = validator.Validate(model);
```

Fig 2.5.d – Code snippet for validation.

Check result object for overall valid , error messages

```
Assert.IsFalse(result.IsValid);
Assert.That(result.Errors[0].ErrorMessage, Is.EqualTo("Name is required"));
```

Fig 2.5.e – Assertion code snippet

# 2.5 Exercises

Identify whether each class is part of the framework or a standalone library.

1. System.Text.StringBuilder — Efficient string manipulation.

2. System.IO.File — File operations like reading/writing files.

3. System.Net.Http.HttpClient — Making HTTP requests.

4. System.Linq.Enumerable — LINQ extension methods for collections.

5. System.Text.Json.JsonSerializer — JSON serialization/deserialization.

6. Controller in ASP.NET MVC — You inherit from this and the framework routes HTTP requests to your methods.

7. Startup in ASP.NET Core — Defines how the app is configured; the framework calls `ConfigureServices` and `Configure`.

8. DbContext` in Entity Framework — You define your data model, and EF Core handles the life-cycle and queries.

9. `PageModel` in Razor Pages — Used in ASP.NET Core for page logic; the framework binds requests to these.

10. `BackgroundService` in.NET Worker Services — You implement background tasks, and the host framework manages execution.

Answers

Framework: 1 - 5

Library: 6 - 10

## GitHub Repository

Repo URL: https://github.com/abhishekcbhattacharya-svg/BreakItMakeIt_Exercises_v2

Folder: *BreakItMakeIt_Exercises/Chapter_02_Framework_vs_Library*

**Steps**

git clone https://github.com/abhishekcbhattacharya-svg/BreakItMakeIt_Exercises_v2.git

cd BreakItMakeIt_Exercises_v2

Use VS Code or Visual Studio 2022 C# IDE

Create a validation implementation for UserModel class

**Level 1: Framework Validation Tests**

Topics:

1. Write your model validation using DataAnnotations,
2. Write code to mimic validation behavior

Challenge: Create validation method implementation for MVC framework.

**Level 2: Library Validation Tests**

Topics:

1. Write your model validation using Fluent Validation nuget package,
2. Write validator class to help validation behavior

Challenge: Create validation method implementation for FluentValidation nuget package.

# 2.6 MCQ

1. Which analogy best explains the relationship between frameworks and libraries?

A. Frameworks are appliances, libraries are sockets
B. Frameworks are sockets, libraries are appliances
C. Frameworks are utilities, libraries are rules
D. Frameworks are houses, libraries are furniture

Answer: B Explanation: Frameworks provide the structure (sockets), while libraries are plug-in utilities (appliances).

2. What does the “Hollywood Principle” signify in frameworks?

A. Developers call the framework directly
B. Frameworks dictate naming conventions
C. “Don’t call us, we’ll call you” — framework controls execution flow
D. Frameworks are always opinionated

Answer: C Explanation: Frameworks control the flow and invoke developer code when needed.

3. Which of the following is an example of a library in C#?

A. ASP.NET Core MVC Controller
B. Entity Framework DbContext
C. System.Text.Json.JsonSerializer
D. Razor Pages PageModel

Answer: C Explanation: JsonSerializer is a standalone library for JSON serialization, unlike framework-managed classes.

4. What is the main difference between frameworks and libraries?

A. Libraries enforce strict architecture, frameworks are flexible
B. Frameworks control execution flow, libraries are invoked directly by developers
C. Frameworks are plug-and-play, libraries dictate project structure
D. Libraries cannot be reused across projects

Answer: B Explanation: Frameworks manage control flow, while libraries are utilities explicitly called by developers.

5. Why are frameworks often described as “opinionated”?

A. They enforce strict rules and templates for structure
B. They allow unlimited flexibility without guidance
C. They only support one library at a time
D. They prevent developers from writing business logic

Answer: A Explanation: Opinionated frameworks enforce conventions and templates, reducing flexibility but ensuring consistency.

6. Which testing focus is recommended for developers when working with frameworks and libraries?

A. Testing framework code directly
B. Testing business rules and logic they implement
C. Testing framework navigation and lifecycle
D. Testing compiler optimizations

Answer: B Explanation: Developers should test their own business logic, not the framework internals.

7. Which frontend framework is known for enforcing strict architecture and conventions?

A. React
B. Angular
C. Vue
D. Express

Answer: B Explanation: Angular is highly opinionated, enforcing strict architecture and conventions.

8. Which frontend tool provides defaults but allows flexibility, balancing framework and library characteristics?

A. Angular
B. React
C. Vue
D. jQuery

Answer: C Explanation: Vue offers defaults but allows flexibility, making it less opinionated than Angular.

9. In testing workflows, how do frameworks and libraries interact?

A. Frameworks execute test code, libraries provide reusable utilities
B. Libraries execute test code, frameworks provide utilities
C. Frameworks and libraries are tested independently
D. Libraries cannot be tested within frameworks

Answer: A Explanation: Frameworks (like NUnit) execute test code, while libraries provide reusable logic within tests.

10. Which principle explains why developers don't control when framework code executes?

A. Dependency Inversion Principle
B. Hollywood Principle

C. Law of Demeter
D. Open/Closed Principle

Answer: B Explanation: The Hollywood Principle means frameworks control execution timing, not developers.

# Chapter 3: SOLID principle for clean code

## Learning outcomes

1. S.O.L.I.D principles for cleaner code tests.
2. Law of Demeter for handling chain code.

## Definitions for principles:

**S.O.L.I.D** — An acronym for set of design principles that help to create software easy to maintain, extend and refactor.

1. **Single Responsibility Principle (SRP):** Each class or module should focus on one responsibility, so it has only one reason to change.

   *Reference:* Martin, R. C. (2002). *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall.

2. **Open/Closed Principle (OCP):** Code should be designed so that new functionality can be added without altering existing, tested code.

   *Reference:* Meyer, B. (1988). *Object-Oriented Software Construction*. Prentice Hall.

3. **Liskov Substitution Principle (LSP):** Subclasses must be usable in place of their parent classes without breaking program correctness.

   *Reference:* Liskov, B., & Wing, J. M. (1994). *A Behavioral Notion of Subtyping*. ACM Transactions on Programming Languages and Systems, 16(6), 1811–1841.

4. **Interface Segregation Principle (ISP):** Interfaces should be small and specific, so clients only depend on methods they actually use.

   *Reference:* Martin, R. C. (2002). *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall.

5. **Dependency Inversion Principle (DIP):** High-level modules should rely on abstractions rather than concrete implementations, enabling flexibility and easier testing.

   *Reference:* Martin, R. C. (2002). *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall.

**Law of Demeter (LoD)**

1. **Core Idea:** An object should only communicate with its immediate collaborators (its own fields, parameters, or locally created objects), not with the internal details of distant objects.

*Reference:* Holland, I. (1987). *Law of Demeter: A Structural Guideline for Designing Object-Oriented Systems*. Northeastern University, Technical Report.

2. **Design Heuristic:** By limiting knowledge of other classes, LoD reduces coupling and makes systems easier to test and maintain.

   *Reference:* Riel, A. J. (1996). *Object-Oriented Design Heuristics*. Addison-Wesley.

3. **Modern Application:** Avoids long chains of method calls (e.g., a.getB().getC().doSomething()), which complicate unit testing and increase fragility.

   *Reference:* Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall.

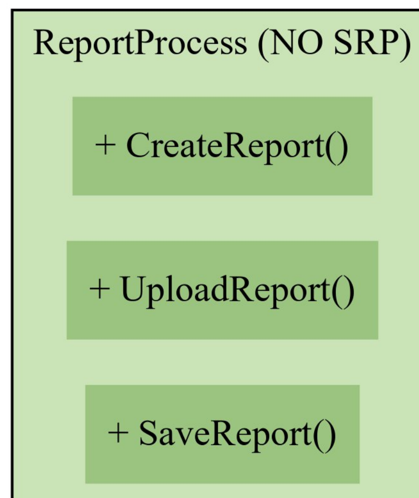
**Note**: Principles are time proven guidelines along with their complexities. When you are executing code as a playground exercise, it is not an expected rule. You follow principles after factoring cost benefit analysis when you have a need to implement a long term maintenance.

## 3.1 Single Responsibility Principle

A class with multiple responsibilities will stretch to different directions as change requests occur. This is one important principle to start with tests then other principles add up for additional benefits for architecture

Create a class based on steps for clean separation.

1. Create separate classes for each responsibility.
2. Finally create a class only to manage inter working among components.




SRP Class seperation

Fig 3.1.a – A class handling reports

A report handling class here involves getting information, formatting information and saving information to retrieve later.

1. There are two file operations related to save, upload respectively.
2. There are operation based for creating report that implies logical handling.

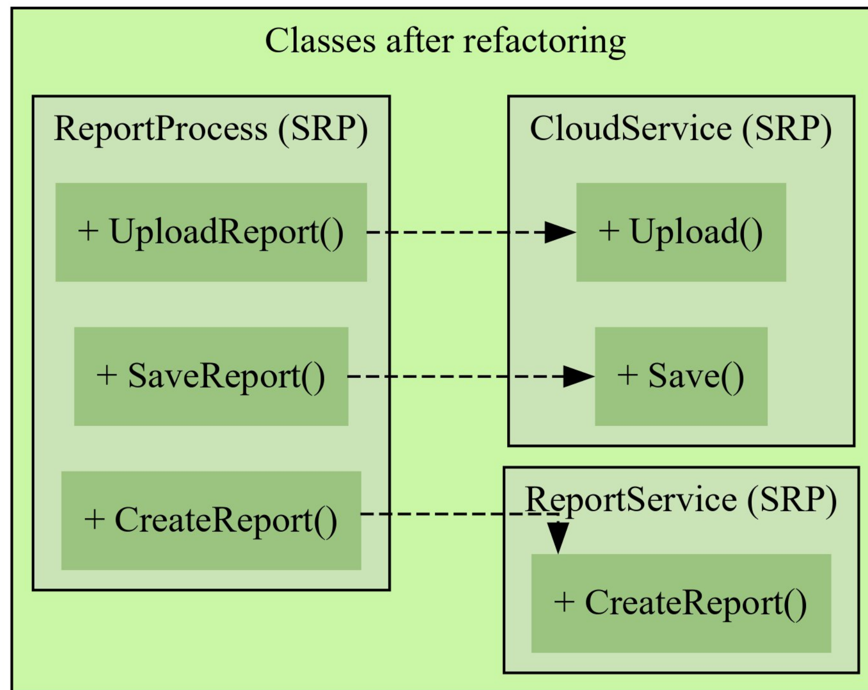



Fig 3.1.b – A class handling reports based on Single Responsibility Principle

**Note**:This is fundamental principle required for test friendly class design. Bad class design will create bad tests.

In a thought experiment , if you are asked to replace cloud service vendor code to another provider. Find out how many classes are affected even though they are not related in technical implementation. If there is any, then its a violation for SRP principle – one reason to change.

Once you create separate classes to handle file operation, create report respectively. You can easily incorporate them as a dependency.

### 3.1.1 Domain of context

There is domain of context which requires some research before you conclude your changes. All changes should exist as one cohesive manner.Information such as Class type definition, namespace can highlight implementation specific or agnostic behavior.

There is a merit to describe it as its behavior. Contradiction to act in opposition will create error for details.

For example, A class with rules for validation, storing information to database and interacting with email, SMS communication will change.

There are four different directions for change

1. Business rule validation
2. Database interaction
3. Email communication
4. SMS communication

You are to create a high level class to manage logic and indirectly interact with components. There is one business rule component and 3 external component – database, email, sms.

A top down approach requires to create very clear definitions and thought.

For bottom up approach, Start with working code and continue refactoring based on SRP to create method. Once all methods created, create a class to be included as dependency.

**Note**:Please refer 8.3 Separation of vectors – external components from class code for steps for refactoring with intent of Single responsibility per class.

Successful implementation cases

1. Class have achieved low coupling, requiring lesser mocking dependencies.
2. Implementations have localized changes.
3. It makes easier to write isolated unit tests

## 3.1.2 Exercises

Each exercise contains instructions, Please make honest attempts before reviewing solution

## GitHub Repository

Repo URL: https://github.com/abhishekcbhattacharya-svg/BreakItMakeIt_Exercises_v2

Folder: *BreakItMakeIt_Exercises/Chapter_03_SOLID/SRP*

**Steps**

git clone https://github.com/abhishekcbhattacharya-svg/BreakItMakeIt_Exercises_v2.git

cd BreakItMakeIt_Exercises_v2

Use VS Code or Visual Studio 2022 C# IDE

## 3.2 Open Closed Principle

The principle is commonly denoted as class, modules, functions being open for extension , closed for modification.

### 3.2.1 Methods/Functions

Class methods / functions have encapsulated logic. In order to manage a method ideally accepts parameters that allows for configuration. If you are required to implement a calculation based on formula. Formula is consistent and values that you will apply will vary. All varying inputs are parameters for method.

Calculations are easiest to explain incorporating changes without changing behavior

1. Area of circle = 3.14159 * radius^2
2. Radius is a variable information, it will be counted as method parameter.
3. 3.14159 is a constant denoted by pi, it is shared information to avoid magic numbers.

```
public class Circle
{
    private const decimal _PI = 3.14159M;
    public decimal Area(decimal radius)
    {
        //Calculation
        return _PI * radius * radius;
    }
}
```

Fig 3.2.1.a – Calculate Area for circle

### 3.2.2 Class

Implementation for open closed principle vary from polymorphism to using dependency injection, design patterns like template method to decouple high level logic and low level implementation.

```
public class Square
{
    public decimal Area(decimal side)
    {
        return side * side;
    }
}
```

Fig 3.2.2.a – Calculate Area for circle

In comparison for Fig 3.2.1.a and Fig 3.2.2.a, you will see pattern for abstraction/inheritance.

```
public abstract Shape
{
    public abstract decimal Area();
}
```

Fig 3.2.2.b – Abstraction class for shape

**Open for extension**

Circle, square are simpler shapes to begin with. There are other shapes like triangle, rhombus have individual formulae calculation. A new shape can be handled just by inheriting from Shape abstract class.

**Closed for modification**

An abstract class creates implementation agnostic code. Abstract class is used for advanced scenarios like to handle common shared code.

Steps

1. Create a implementation detailed class for grouping input for method / function.
2. Create a interface/ abstract class for handling defined classes.

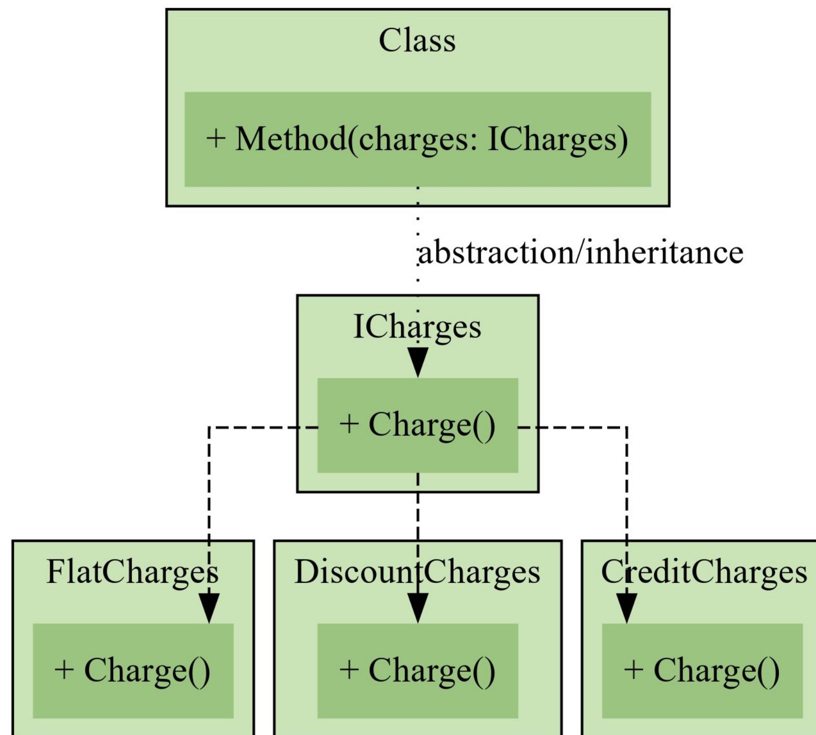



Fig 3.2.2.2.a – Abstraction example class for transaction charges

### 3.2.3 Exercises

Each exercise contains instructions, Please make honest attempts before reviewing solution

### GitHub Repository

Repo URL: https://github.com/abhishekcbhattacharya-svg/BreakItMakeIt_Exercises_v2

Folder: *BreakItMakeIt_Exercises/Chapter_03_SOLID/OCP*

**Steps**

git clone https://github.com/abhishekcbhattacharya-svg/BreakItMakeIt_Exercises_v2.git

cd BreakItMakeIt_Exercises_v2

Use VS Code or Visual Studio 2022 C# IDE

## 3.3 Liskov Substitution principle

This principles dictates any subclass can replace by parent class without breaking behavior.

When you derive class from parent class, make sure no functionality of class should reduce capability in comparison of parent class.

Violation Examples:

1. You can derive ReadOnlyList from List, which does not allow any modification.
2. Dot is a shape that does not contribute to area. Area is an invalid method for Dot class.

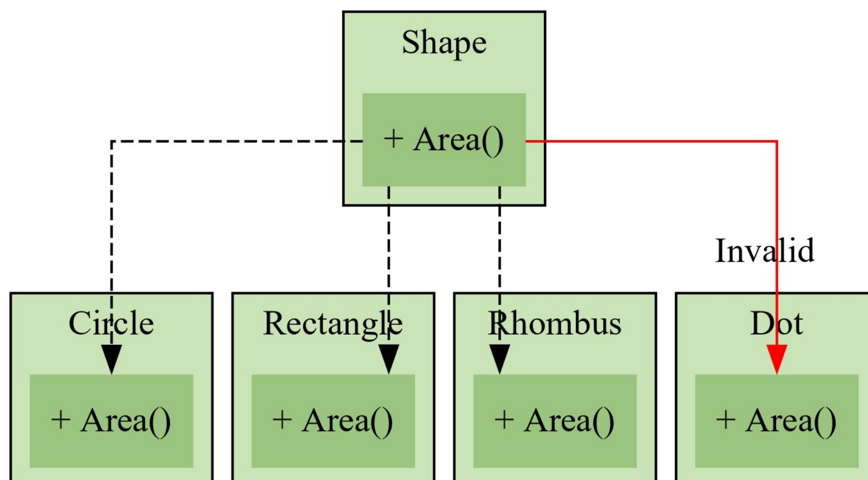



Fig 3.3.a – Violation for Liskov substitution principle

The derived class implementation should not decrease abilities available before inheritance and should increase abilities after inheritance.

It is importance of preserving capabilities that should maintain code function for predictable behavior in use cases.

Inheritance is concept that allows to add layer of new functionality for new class that depends on fundamental base class. Given substitution is mentioned of any new class with base class.

The only common implementation among new derived is parent class. Thus make sure to use declared functionality of base class and derived classes can vary all definitions as per purpose of class

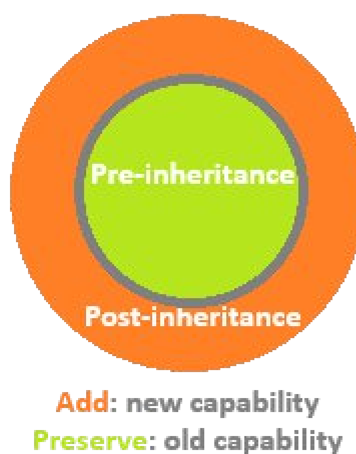



Fig 3.3.b – Venn diagram for pre and post changes

**Observations**:

You will face substitution related issues for top down related changes. Base implementation will be forced commitment.

**Top Down approach**

Inheritance is a process confirming full parent inclusion and individual characteristics using top down process. Forcing a template that is not supported means to split information that doesn't comply.

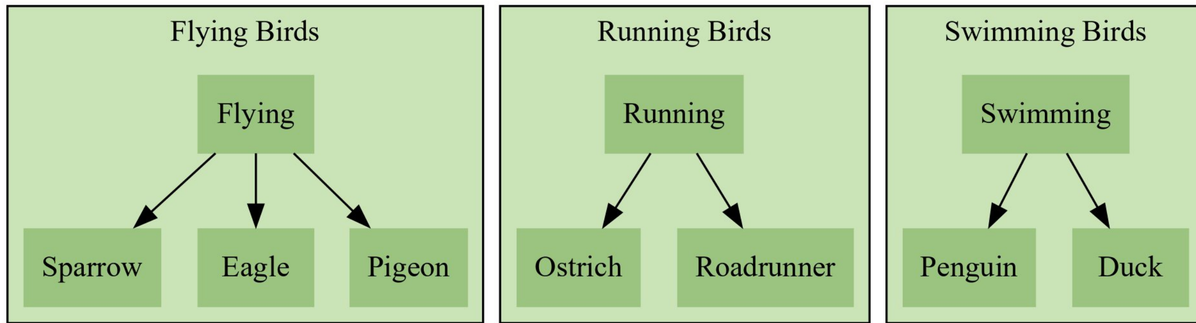



Fig 3.3.c – Arranging by common origin

The violations of Liskov Substitution Principle highlight importance of using composition over inheritance.

Liskov substitution principle highlights problem with inheritance breaking behavior. A flying bird has different method of locomotion in comparison to a running bird.

## 3.3.1 Exercises

Each exercise contains instructions, Please make honest attempts before reviewing solution

## GitHub Repository

Repo URL: https://github.com/abhishekcbhattacharya-svg/BreakItMakeIt_Exercises_v2

Folder: *BreakItMakeIt_Exercises/Chapter_03_SOLID/LSP*

**Steps**

git clone https://github.com/abhishekcbhattacharya-svg/BreakItMakeIt_Exercises_v2.git

cd BreakItMakeIt_Exercises_v2

Use VS Code or Visual Studio 2022 C# IDE

# 3.4 Interface Segregation principle

Clients need to implement only required functionality without forcing to depend upon interfaces they do not use.

This principles works closely with Liskov substitution principle. If an interface forces working behavior to all implementations suggesting  top down approach. Interface segregation principle is a feedback mechanism from implementations to interface, suggesting bottom up approach

This principle clearly promotes composition over inheritance.

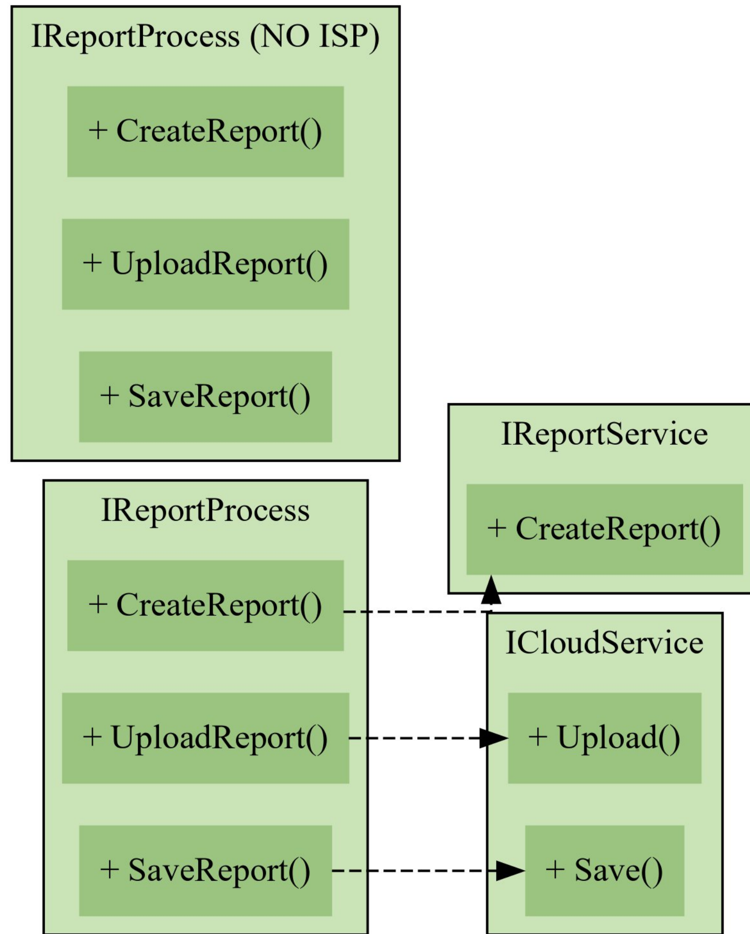



Fig 3.4.a – Example for composition over inheritance

Interface Segregation is a process for discovery of elements that change because for change in component.

A ReportProcess class is a class managing and using dependencies to achieve functionality mentioned for contract  Each dependencies has a trait of Single Responsibility as test foundation. ISP works on contracts – interfaces for C#

**Bottom up approach**

A common process of creating structure

1. Create a collection of class entities holding all information at same level.
2. Compare two or more entities for common members properties, methods.
3. Continue step 2 to compare recursively.

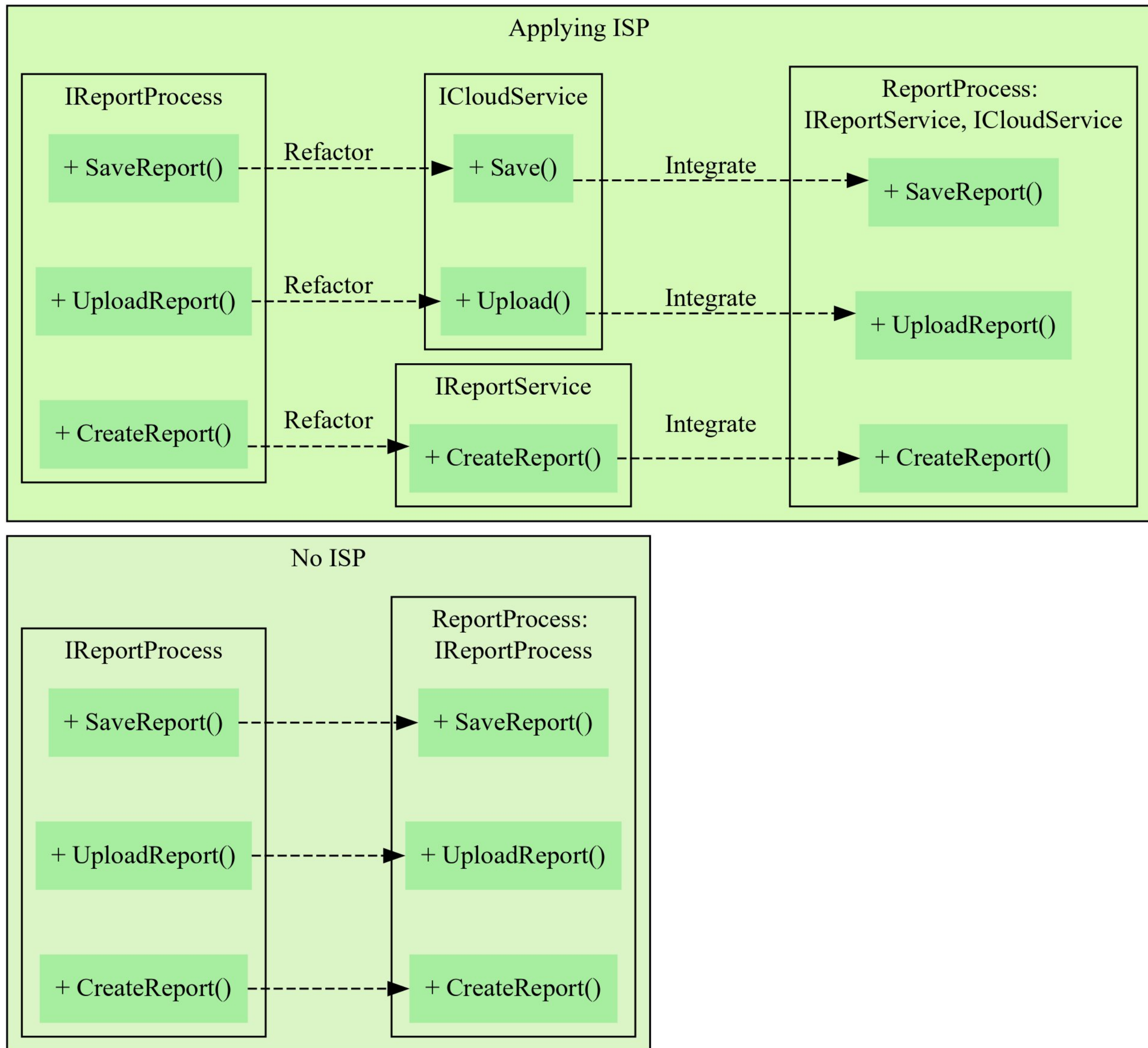



Fig 3.4.b – Example for composition over inheritance

Consider example of very sophisticated printer, below interface highlights all functionalities.

```
interface IPrinterTasks {
    void Print();
    void Scan();
    void Fax();
    void DuplexPrint();
}
```

Fig 3.4.c – Example for sophisticated printer

If a basic printer only supports Print() and Scan(), it shouldn't be forced to implement Fax() and DuplexPrint().There are two devices printer and scanner.

Single responsibility principle can help separate responsibilities, while Interface Segregation Principle removes unnecessary contracts definition. Design two interfaces separate

```
interface IPrint {
    void Print();
}

interface IScan {
    void Scan();
}
```

Fig 3.4.d – Example for separated printer, scanner

This principle asks for clean separation for individual components and applying combination of interface. As a side effect of separation, you are not required to implement unnecessary methods.

The Interface Segregation Principle (ISP) supports Test-Driven Development (TDD) by fostering better design thinking.

1. TDD naturally encourages frequent refactoring.
2. Smaller, focused interfaces reduce unnecessary dependencies.
3. ISP makes refactoring safer by isolating responsibilities.
4. You can modify one interface without disrupting unrelated clients or tests.
5. Tests become more precise and less fragile.
6. ISP ensures mocks stay lightweight and relevant, minimizing setup overhead.
7. By nudging you toward purpose-driven, minimal interfaces, ISP complements the practice of writing effective tests.

**Observations**:

You implement a bottom up approach to separate out unnecessary implementation.

## 3.4.1 Exercises

Each exercise contains instructions, Please make honest attempts before reviewing solution

## GitHub Repository

Repo URL: https://github.com/abhishekcbhattacharya-svg/BreakItMakeIt_Exercises_v2

Folder: *BreakItMakeIt_Exercises/Chapter_03_SOLID/ISP*

**Steps**

git clone https://github.com/abhishekcbhattacharya-svg/BreakItMakeIt_Exercises_v2.git

cd BreakItMakeIt_Exercises_v2

Use VS Code or Visual Studio 2022 C# IDE

## 3.5 Dependency Inversion principle

Lower level implementation details should depend on higher level specifications instead of dictating them.

In practical code examples, a class contains technical implementation. A business rule is implemented as concept, details can or will include class for specific technical implementation.

If a technology supports a unique feature, it does not become precursor for business rule change. A sorting algorithm performs sort operation, you are free to select that works best for case. It should not have any business rules changes.

Dependency inversion principle DIP is important as foundation next to single responsibility principle. It works closely with constructor injection, method injection, property as last resort.

For implementation of Dependency injection, create an interface from class and used in all implementation references. Create abstract class for advanced cases where common functionality exists for reuse

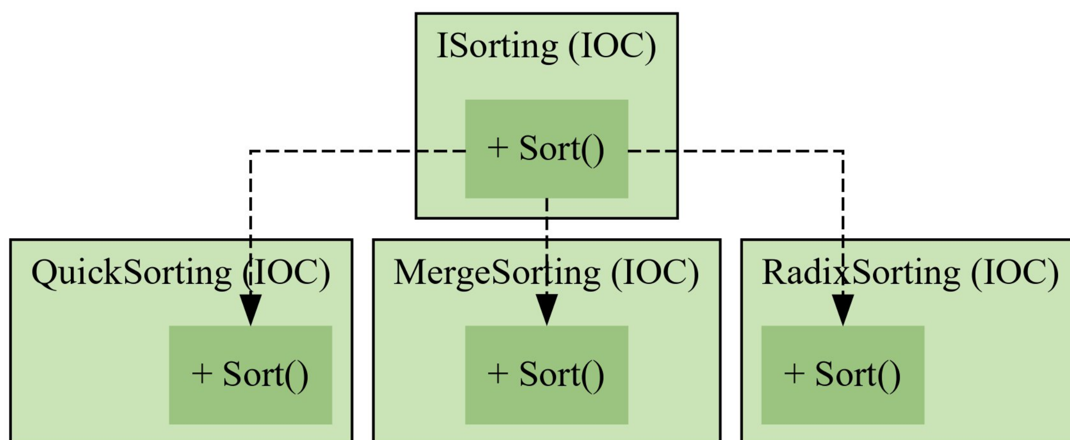



Fig 3.5.a – Example for separation of specification and  implementation

ISorting is an example here are important to sort information.

Complicated systems use logging for debug for error case. You may implement logger based on local file system, online file storage, database. In case of change of vendor, you have option to create new implementation

Implementation of logger can be file based, or file storage on internet. In case of change of vendor, create a new implementation.

For example, Instead of UserService depending on SqlUserRepository, it depends on IUserRepository. The concrete type is registered in the DI container.

**Inversion of Control (IoC)**

A design principle where control of program flow is inverted from code to framework / container. Application does not decide how and when dependencies are created / used. A framework or container is responsible for creating dependencies for application code

In .NET Core, the built-in IoC container manages services.

You register services in **Startup.cs** (or **Program.cs** in newer versions), and the framework decides when to instantiate them. A common setup is to create an interface mapped to one among possible multiple class implementations.

**Dependency Injection (DI)**

This is one of most used technique to achieve Inversion IoC. DI can be achieved using constructor, method or property injection.

1. Constructor injection is a common format to pass mandatory arguments as parameter that can exceed lifetime of object.
2. Method injection is mandatory arguments for method parameters.
3. Property injection is optional that may be last available option if class has default constructor only.

For example, Instead of UserService creating its own UserRepository, the repository is injected via constructor or setter.

## 3.5.1 Exercises

Each exercise contains instructions, Please make honest attempts before reviewing solution

## GitHub Repository

Repo URL: https://github.com/abhishekcbhattacharya-svg/BreakItMakeIt_Exercises_v2

Folder: *BreakItMakeIt_Exercises/Chapter_03_SOLID/DIP*

**Steps**

git clone https://github.com/abhishekcbhattacharya-svg/BreakItMakeIt_Exercises_v2.git

cd BreakItMakeIt_Exercises_v2

Use VS Code or Visual Studio 2022 C# IDE

## 3.6 Law of Demeter

A method should interact only with immediate friends; method arguments, class instance object.

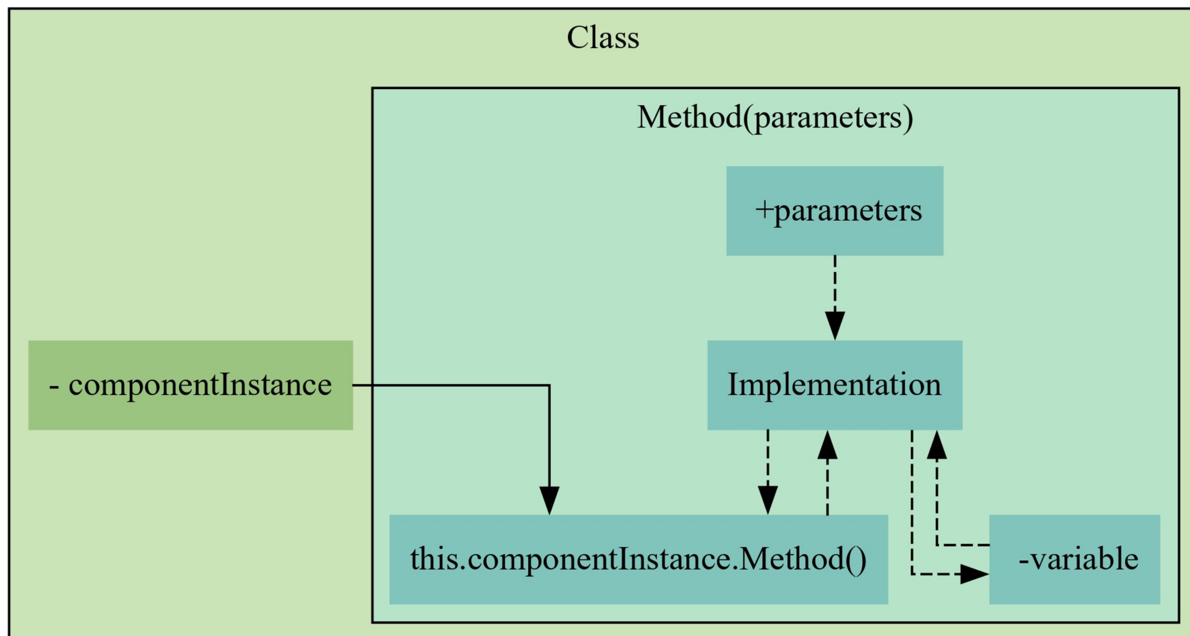



Fig 3.6.a – Example of valid interaction for method

Method is directly connected to

1. Object - class instance
2. Arguments passed to method
3. Private members of object – any dependency
4. Variables created within scope of method

Calling a method chain is an easy form of LoD violation. “Obj.MethodA().MethodB()” A method response variation to response object creates a increasing combination for error at nested method calls

When you call more than level one nested class within same class, you are bound to handle internal details for called nested classes. A common case to create class to manage logic for other class in violation of single responsibility principle.

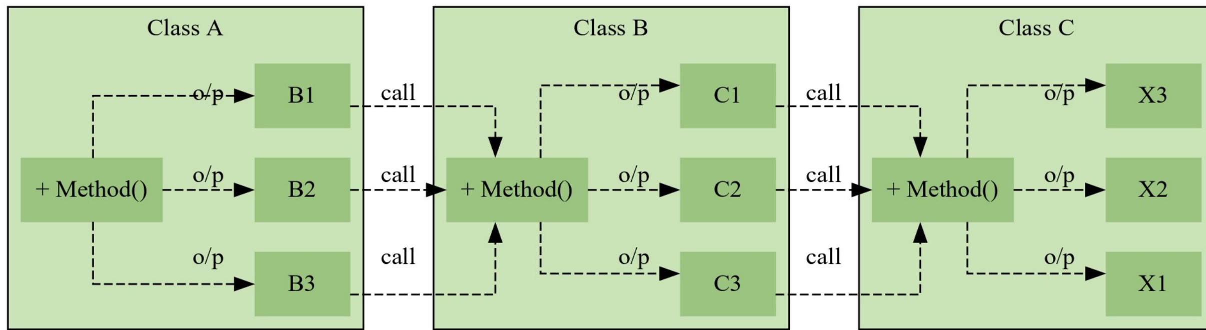



Fig 3.6.b – Example of invalid interaction for method

For each method call provides a variation of instance of next class.

A will call B,B call C,Thus A calls C transitively.

If each method had 3 output variations, you will end up in 27 combinations exploding count with each increasing chain. “Obj.MethodA().MethodB()” is a classic example to highlight exponential combination of possible variations.

Note: LINQ is a set of Extension Methods that do not involve / represent state change.

**Data transfer object**

For data transfer object reading information, you would read “**dto.PropertyA.PropertyB**”. This is a important distinguishing point, you are accessing information that should not ideally create any state changes on read access in short - readonly access.

**Handling violations**

A shorthand for LoD is to check ‘No. of dots for property, method access’. It should not exceed more than single.

**Quick Tips to Apply LoD**

- Please avoid method chaining (a.AccessB().AccessC().ProcessSomething()).
- Use delegation: let objects handle their internal subcomponents.
- Design interfaces that expose only valid.

Avoiding method chaining

```
1 reference
class A
{
    1 reference
    B AccessB()
    {
        return new B();
    }
}

2 references
class B
{
    1 reference
    C AccessC()
    {
        return new C();
    }
}

2 references
class C
{
    1 reference
    void ProcessSomething()
    {
        //Process
    }
}
```

Fig 3.6.c – Example of invalid interaction for method

```
(new A()).AccessB().AccessC().ProcessSomething();
```

Fig 3.6.d – Example of invalid interaction for LoD

Design interfaces

```
0 references
interface A
{
    0 references
    void ProcessSomething();
}
```

Fig 3.6.e – Example of valid interface for LoD

Handle internal components

```
2 references
class LoD_A
{
    2 references
    private readonly LoD_B _b;
    1 reference
    public LoD_A(LoD_B b)
    {
        _b = b;
    }
    0 references
    void ProcessSomething()
    {
        _b.ProcessSomething();
    }
}

4 references
class LoD_B
{
    2 references
    private readonly LoD_C _c;
    1 reference
    public LoD_B(LoD_C c)
    {
        _c = c;
    }
    1 reference
    void ProcessSomething()
    {
        _c.ProcessSomething();
    }
}

3 references
class LoD_C
{
    1 reference
    void ProcessSomething()
    {
        //Process
    }
}
```

Fig 3.6.f – Example of fix for LoD

```
var a = new LoD_A(new LoD_B(new LoD_C()));
a.ProcessSomething();
```

Fig 3.6.f – Example of method execution for Law of Demeter

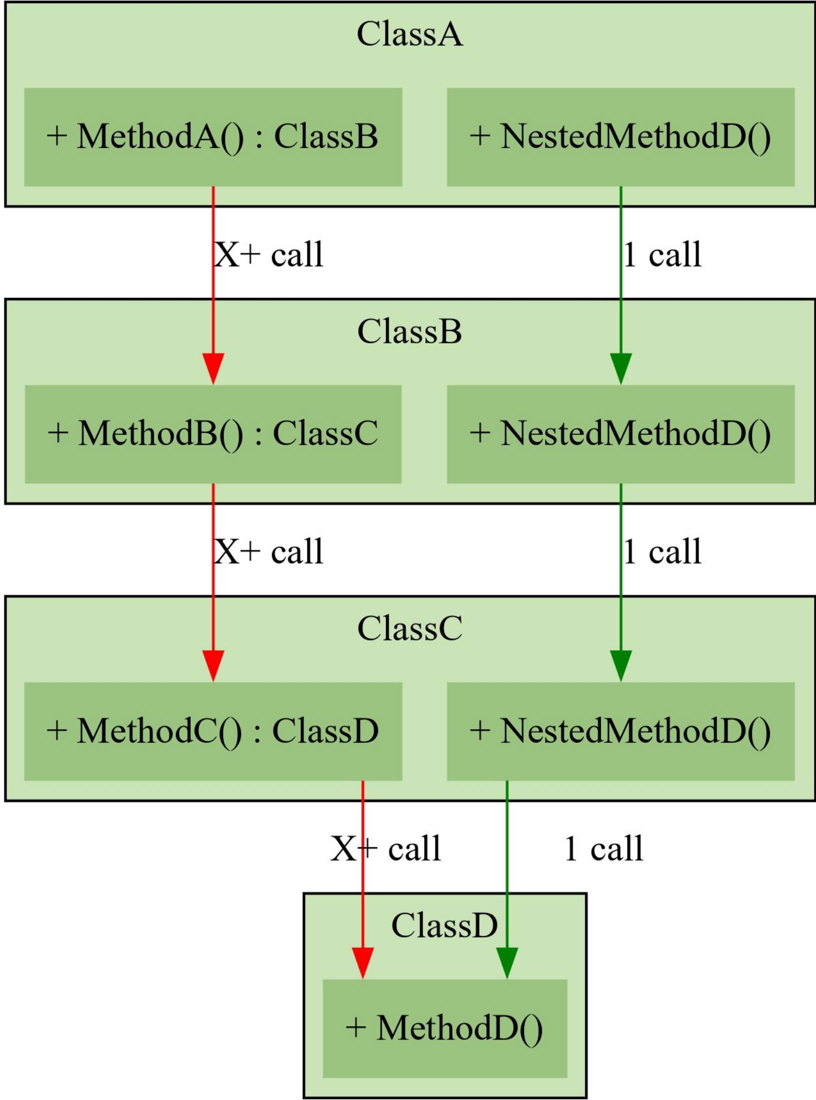



Fig 3.6.g – Example of handling nested interaction for method

## 3.6.1 Exercises

Each exercise contains instructions, Please make honest attempts before reviewing solution

## GitHub Repository

Repo URL: https://github.com/abhishekcbhattacharya-svg/BreakItMakeIt_Exercises_v2

Folder: *BreakItMakeIt_Exercises/Chapter_03_SOLID/LOD*

**Steps**

git clone https://github.com/abhishekcbhattacharya-svg/BreakItMakeIt_Exercises_v2.git

cd BreakItMakeIt_Exercises_v2

Use VS Code or Visual Studio 2022 C# IDE

# 3.7 MCQ

1. What does the Single Responsibility Principle (SRP) state?

A. A class should handle multiple unrelated tasks
B. A class should have only one reason to change
C. A class must always inherit from another class
D. A class should depend on multiple external services

Answer: B Explanation: SRP ensures each class focuses on a single responsibility. If a class has multiple reasons to change, it becomes harder to maintain and test.

2. Which principle emphasizes “open for extension, closed for modification”?

A. Liskov Substitution Principle
B. Dependency Inversion Principle
C. Open/Closed Principle
D. Interface Segregation Principle

Answer: C Explanation: OCP means you can add new functionality (extend) without altering existing code, reducing the risk of breaking what already works.

3. Which of the following is a violation of the Liskov Substitution Principle (LSP)?

A. A subclass adds new functionality.
B. A subclass reduces capabilities compared to its parent
C. A subclass overrides methods with consistent behavior
D. A subclass uses composition instead of inheritance

Answer: B Explanation: LSP requires that subclasses behave consistently with their parent classes. If a subclass removes or weakens expected behavior, it violates LSP.

4. The Interface Segregation Principle (ISP) suggests that:

A. Interfaces should be large and cover all possible methods.

B. Clients should only depend on methods they use
C. Interfaces must always inherit from abstract classes
D. Interfaces should enforce multiple responsibilities

Answer: B Explanation: ISP promotes smaller, more specific interfaces so clients aren't forced to implement methods they don't need.

5. Which principle promotes relying on abstractions rather than concrete implementations?

A. Single Responsibility Principle
B. Dependency Inversion Principle
C. Open/Closed Principle
D. Law of Demeter

Answer: B Explanation: DIP encourages depending on abstractions (interfaces) instead of concrete classes, making systems more flexible and easier to change.

6. The Law of Demeter discourages:

A. Using delegation for internal components
B. Long chains of method calls across objects
C. Creating small, focused interfaces
D. Applying constructor injection

Answer: B Explanation: The Law of Demeter (also called "Don't talk to strangers") advises against chaining multiple calls, which increases coupling and reduces clarity.

7. Which principle helps reduce coupling by ensuring classes only interact with immediate collaborators?

A. Interface Segregation Principle
B. Dependency Inversion Principle
C. Law of Demeter
D. Liskov Substitution Principle

Answer: C Explanation: The Law of Demeter ensures that objects only communicate with their direct collaborators, reducing dependency chains.

8. In practice, how does SRP improve testability?

A. By forcing classes to depend on multiple services
B. By reducing the need for mocking and isolating responsibilities
C. By requiring inheritance for all classes
D. By eliminating the need for refactoring

Answer: B Explanation: SRP makes classes smaller and focused, which simplifies unit testing since each class has fewer dependencies and responsibilities.

9. Which design pattern often supports the Open/Closed Principle?

A. Singleton Pattern
B. Template Method Pattern
C. Observer Pattern
D. Adapter Pattern

Answer: B Explanation: The Template Method Pattern allows extending behavior by overriding methods without modifying existing code, aligning with OCP.

10. Why is Dependency Injection (DI) commonly used with the Dependency Inversion Principle?

A. It allows direct instantiation of concrete classes
B. It enforces method chaining for dependencies
C. It enables flexible substitution of implementations via abstractions
D. It eliminates the need for interfaces

Answer: C Explanation: DI provides a way to supply dependencies through abstractions, making it easy to swap implementations without changing client code.

# Chapter 4: Testing Types and Strategies

## Learning outcomes

1. Compare thought process between Unit, Integration, Component and E2E testing.
2. Process and understanding unit testing rules.

## 4.1 Test pyramid structure

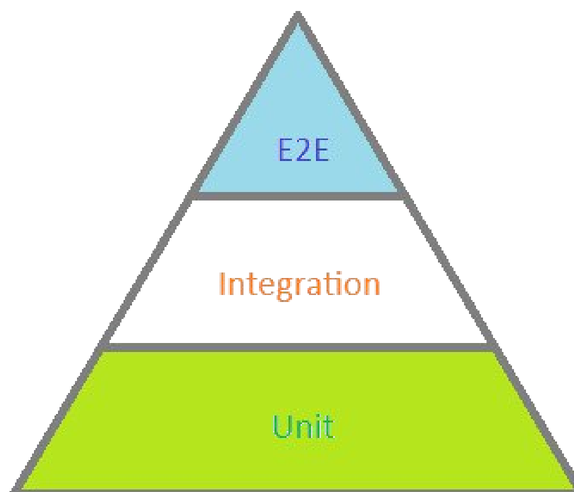



Fig 4.1.a – Test pyramid

A pyramid structure below indicates tests and count per level.

1. At bottom, Unit tests for single focus code tests.
2. At middle, integrated tests for interaction among components.
3. At top, E2E tests for monitoring user operations with all external components.

If no. of unit tests are greater than integration, E2E tests, so you can be more confident.

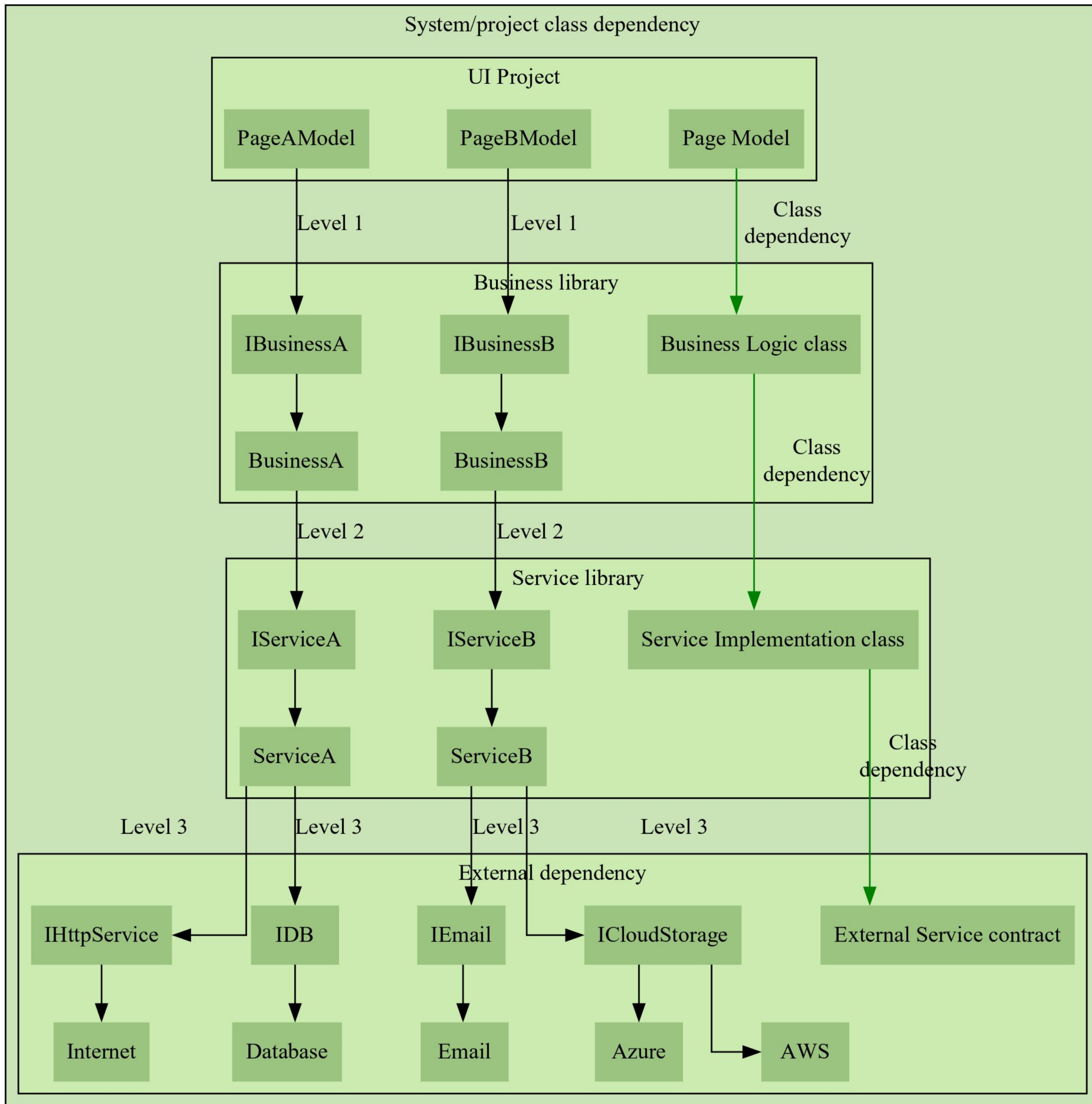
Unit tests focuses on individual class based outcome. This helps to isolate individual errors

Integration tests are focused at aggregated outcome of tests. You are aiming for collective success assuming every dependency works

E2E tests focuses on single user feature flow from start to finish. You deploy to a environment with all dependencies setup. These are most brittle but provide real time results

You are creating E2E tests for insurance against source code modification as first line of defense where project is not monitored for any changes.

## 4.2 Class dependency Tree




Class Dependency Tree

Fig 4.2.a – Project dependency tree

Here is  a typical project implementing dependency injection, including mapping interface to class mapping for flexibility reasons.

Please note transitions marked by ‘Level 1 / Level 2 / Level 3’. Each class is indicating a dependency on interface.

1. BusinessClassA class relies on IServiceClassA interface

2. ServiceClassA class relies on IDB interface
3. IDB represents a class for database related operations.

If you start counting from class and increment at class you meet across transition.

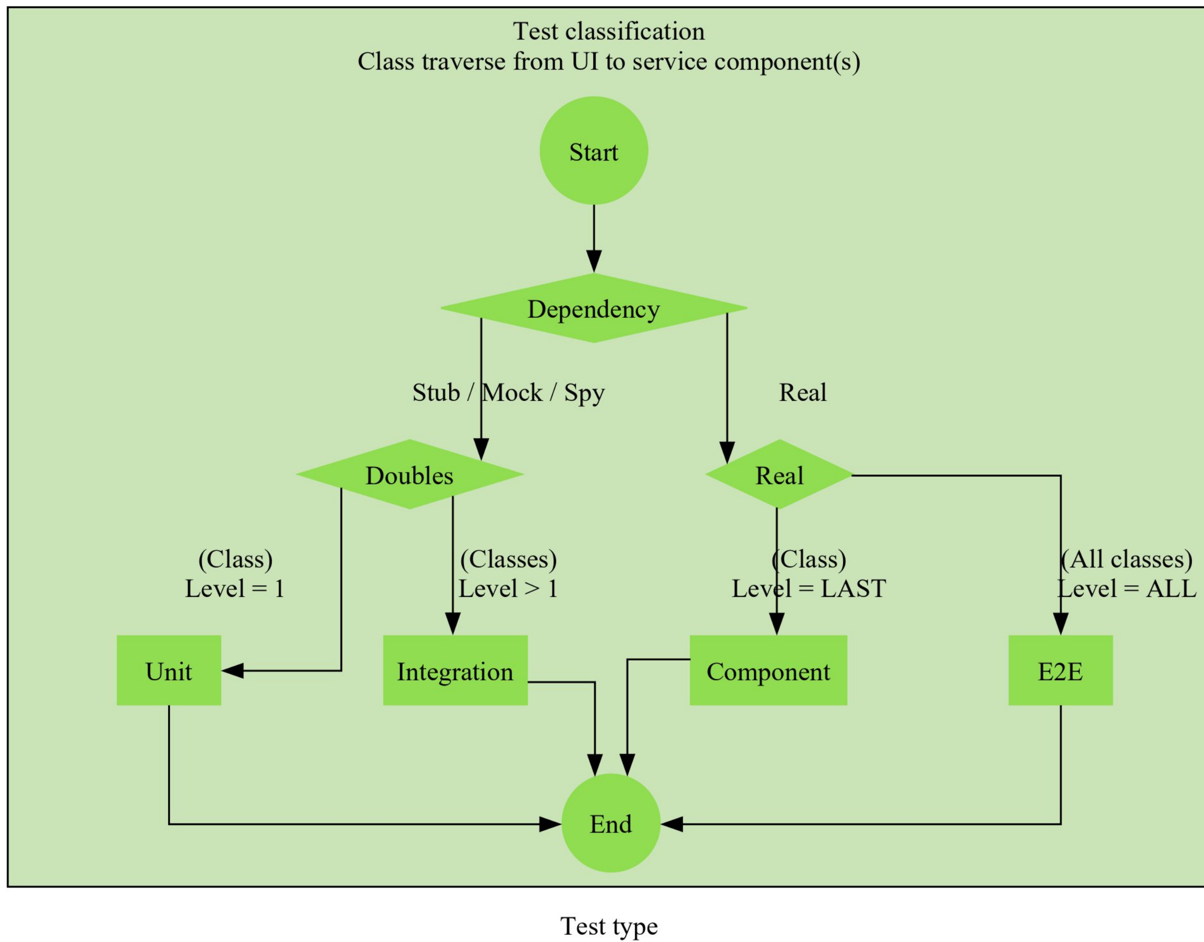



Fig 4.2.b – Test classification type

There are combination of dependency implementation and level of nested component class in system under test.

## 4.3 Unit testing

A unit test is basic building block for testing application or system under test framework. Developers are responsible to write unit tests in order to support code class developed by them. You as developer will make sure will be not to create / access real external resources. External dependencies require substituted response by using stub / mock / spy.

1. All unit tests require isolation.
2. You are not required to test framework code.
3. You will test all business rules.

4. You are testing one class / method as scope of discussion.

This tests are fast execution based philosophy. In order to validate while ensuring all vectors - external components, you will require understanding of important concepts like stub, mock and spy explained in separate chapter.

## 4.3.1 Unit testing scope mechanism explained

Each dependent component calls its implementation based on DI. Based on referenced libraries, each library implementation will have nested dependent call. So, a tree structure of method calls will be made by code. Refer Fig 4.2.a – Project dependency tree.

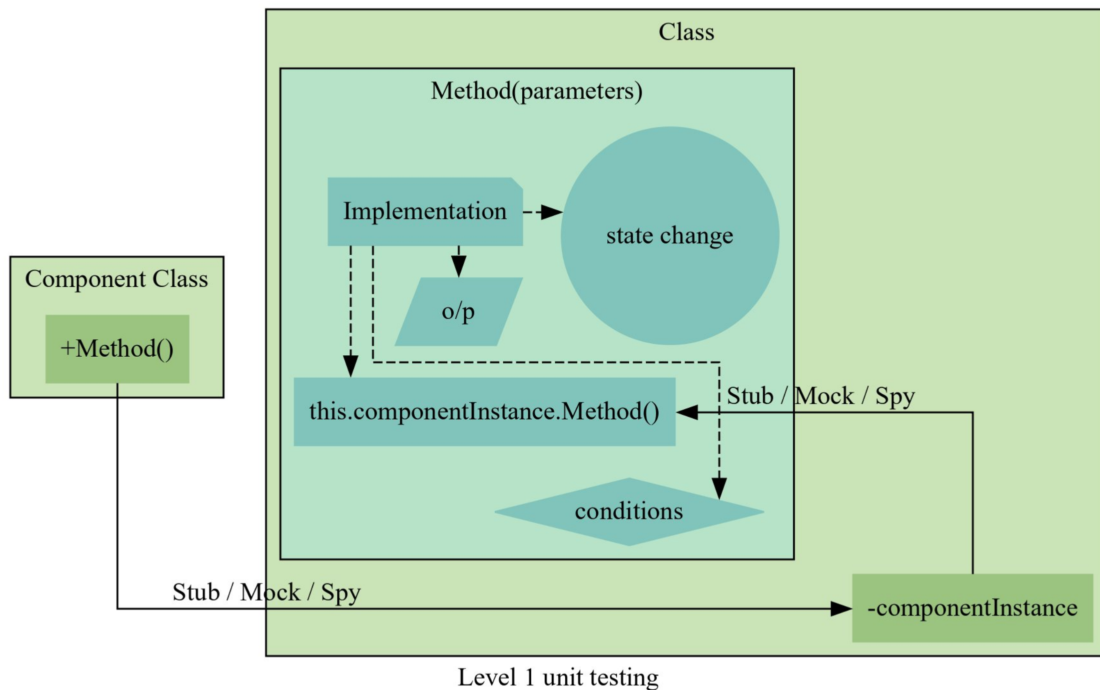



Fig 4.3.1.a – Unit testing scopes – state change / output / component interaction

A class instance is implemented with nested component instance. This is level 1 criteria that will set as unit test. You are testing only immediate one level of method implementation with help of stub / mock / spy component.

We are writing unit tests based on outside in approach. Outside in approach does not handle all details. Outside in approach handles nested complications by making sure to provide a response based on stub / mock / spy setup.

If you Refer Fig 4.2.a – Project dependency tree, first unit tests is written for Page Model class then followed by Business class unit tests and concluded by Service class unit tests.

1. PageModel Unit tests <- BusinessClass stub / mock / spy
2. BusinessClass Unit tests <- ServiceClass stub / mock / spy+

3. ServiceClass Unit tests <- External dependency stub / mock / spy.

Each stub / mock / spy provides information like input, output or state change for next level test case requirement.

The simplest example in real life is a electric bulb in a socket. If you setup current source, when you click switch on, it provides a light as response.

### 4.3.2 Testing goals

A unit test verifies output, state change or component interaction. Assertion can require for combination of above mentioned goals. You required information for asser

#### 4.3.2.1 Output

This is easiest test to achieve for unit tests. Each method can provide publicly accessible response to an execution.

A Sum() method adding two numbers returns sum of inputs. You externally assert calculated value to external expected

#### 4.3.2.2 State change

A method can make state changes, this involves tracking changes for assertion. Changes are usually focusing on database, session related to user interaction. You will create mock to record state change. Using mocked TempData Data Dictionary to handle all message for MVC Controller

You create state changes and compare before and after changes for asserting change.

#### 4.3.2.3 Component interaction

This is hardest to achieve for unit tests. You are accessing internal details for method executed and requires additional mechanism like spy. A method can include interaction with external dependency to achieve functionality like send email, SMS or OTP to devices.

You are assuming component interaction works on fire and forget approach. You cannot verify directly by response, thus you are assuming trust on execution. A common process is to check method execution count

Testing a execution will mean trusting a call to component that is best possible with spy.

## 4.4 Component testing

A component testing is testing external dependency like cloud storage utility, email, OTP individually. Developer test components one by one as cheaper replacement to integration test. There is clear assumption that each service does not interfere with other existing dependencies.

In reference to outside in approach, we are testing last level components. You had avoided testing external dependency to handle isolation requirements for previously written unit tests.

If you Refer Fig 4.2.a – Project dependency tree, you are verifying Internet, Database, Email, Cloud storage - Azure, AWS.

Component testing allows making sure important components are working real time.

Last level of class called will require mocking as operations are performed with external dependency. Example — using Moq for Entity Framework.

Use local SQL database for older version code. Use **[Setup] → [Test] → [Teardown]** to set, test code, reset database. respectively.

Note: A separation of component tests from unit tests allow to minimize no. of tests.

## 4.5 Integration testing

A integration test is for testing application modules or system under test framework. Developers write tests to ensure altogether code and all external dependencies are stub / mock / spy.

You are testing interaction among components for scope of discussion. You are testing multi level testing for classes.

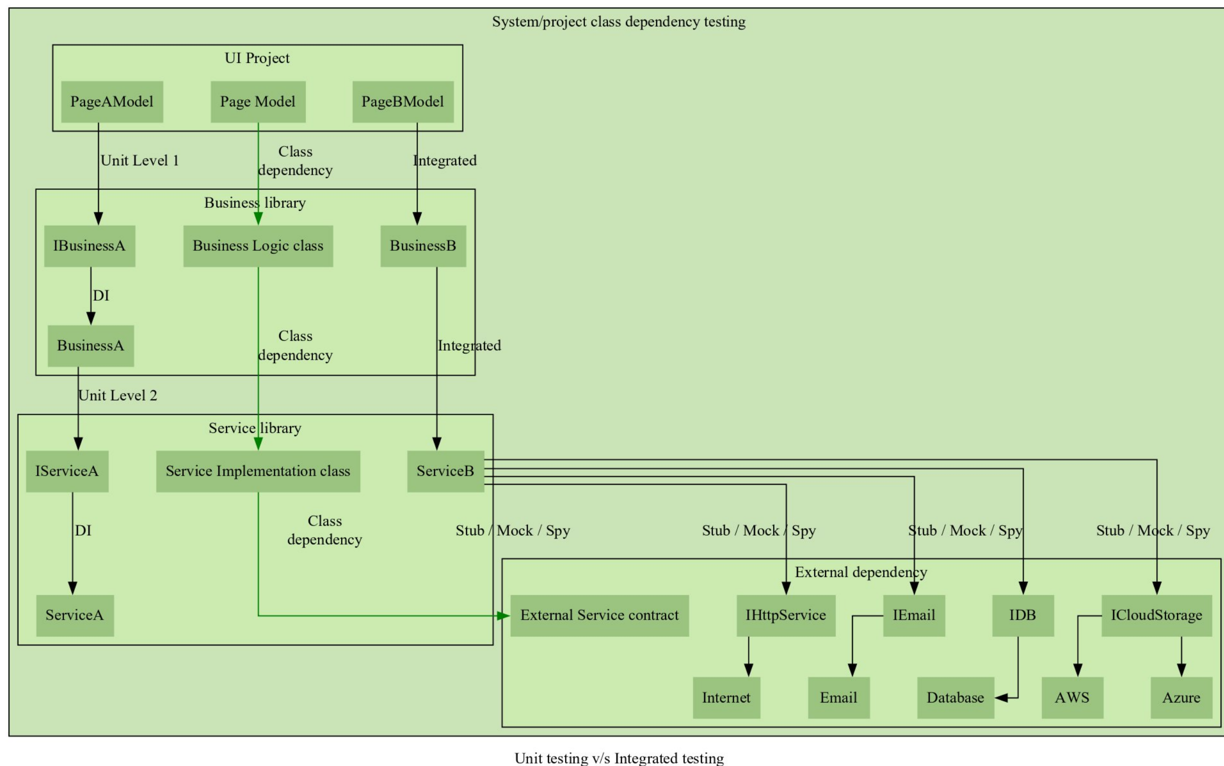



Fig 4.5.a – Unit testing Level 1 v/s Integration testing Level 2+

The thought process involves combining unit tests and next level mocked component tests together with input, output while stub / mock / spy concept still apply.

If you Refer Fig 4.2.a – Project dependency tree, you are verifying PageAModel with actual

1. Unit test: Verify **PageAModel class** using stub / mock / spy for **IBusinessA interface**
2. Integration test: Verify **PageAModel class** using implemented **BusinessA class** using stub / mock / spy for dependent components – **IServiceA interface**.

Refer Fig 4.5.a : Integrated test holds 2 or more live class for testing directly, ServiceB class uses stub / mock / spy for replacement in test. Unit tests are handling level 1 and results creating to 2 separate unit tests

# 4.6 E2E testing

A End to End testing is setup to test a user flow as a end user for processing. This is most elaborate to setup external dependencies like Cloud Storage like AWS, Azure, Email processing for real life scenarios.

It requires environment setup like website/app deployment, database migrations, setting environment variables for security concerns.

UAT is supposed to be structure equivalent to Production environment. If there are minor differences , it causes friction for automation scripting.

You will see use case for playwright to load website, perform actions, validate messages / notifications. Playwright allows automation of websites along with API calls. Older process used Selenium.

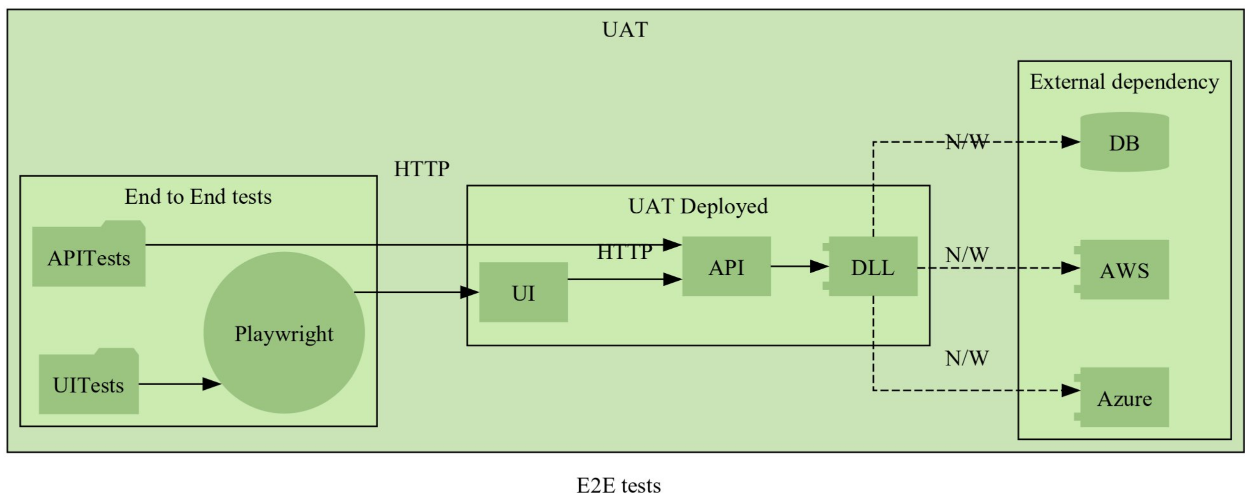



Fig 4.6.a – E2E tests

In comparison we have better support for new websites using modern front end frameworks like React 16+, Vue 3, Angular 12+

E2E tests are seperatedly handled by QA team and  developers help to setup any identifier for page, message or user flow.

## 4.7 Stub v/s Mock v/s Spy

**Stub**: A simplified substitute for a real dependency that returns fixed outputs. It helps a test run by controlling inputs but does not check how the dependency was used.

*Reference*: Fowler, M. (2007). *Mocks Aren't Stubs*. martinfowler.com.

Book Reference: Meszaros, G. (2007). *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley.

**Mock**: A test double that not only provides outputs but also ensures that specific interactions occurred. It is used for *behavior verification*, confirming that the system under test calls its collaborators correctly.

*Reference*: Fowler, M. (2007). *Mocks Aren't Stubs*. martinfowler.com.

Book Reference: Fowler, M. (2003). *UML Distilled* (3rd ed.). Addison-Wesley.

**Spy**: A test double that records details of how it was used (such as which methods were called and with what arguments). Unlike mocks, spies don't enforce expectations upfront but allow verification afterward.

*Reference*: Fowler, M. (2007). *Mocks Aren't Stubs*. martinfowler.com.

Book Reference: Meszaros, G. (2007). *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley.

A stub is a form of test double introduced to make sure code works in replacement of framework based class.

A mock is a form of test double introduced for test code that allows code execution to support. They needs to implement business rules validation.

A spy is a form of test double introduced for code that allows code execution, business rules validation and also measure internal details. This setup has ability to confirm interaction with internal component like sending email service.

Test doubles are part of equation for outside in approach when you are testing a unknown nature. Each mock, spy represents a pending unit test for next level.

### Outside in approach

Refer Fig 4.5.a, you see level of execution classes: PageModel -> Business -> Service. You create a tests based on level implementation. Each implementation is harder to control response and easier to replace by a stub or mock or spy to manage

Note: Creating a separate stub /mock / spy is technical debt created for ease of tests. The moment parent code for stub /mock has changed, you will be required to monitor related tests.

1. PageModel can include Stub / Mock / Spy for Business class.
2. Business class implementation will include Stub / Mock / Spy for Service class.

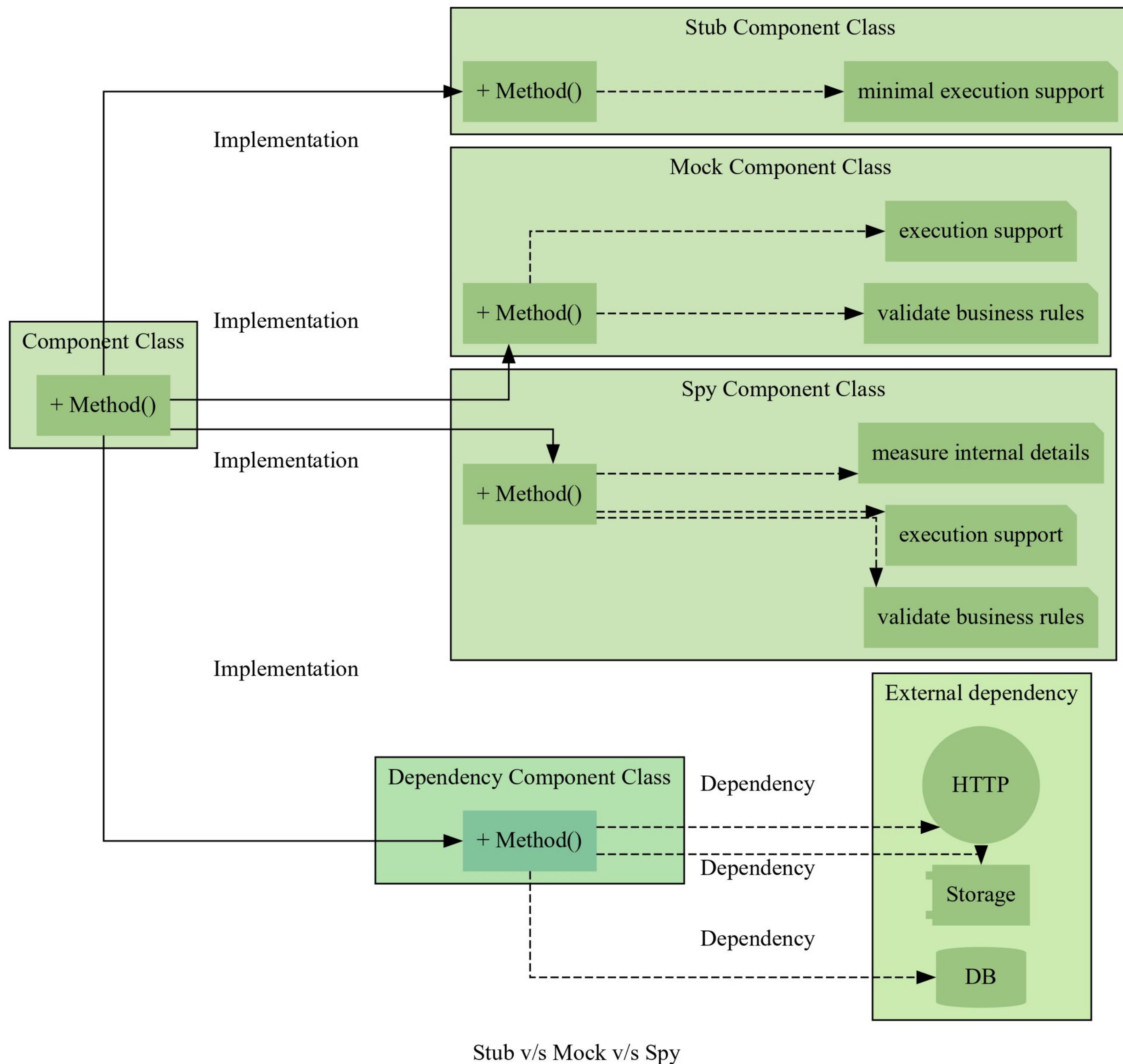



Fig 4.7.a – Comparison for test doubles

You write tests layer by layer as you need to implement and verify details. Write unit tests regularly to make it easier to incorporate along with S.O.L.I.D principles.

Note: This chapter focuses on setting foundation for tests and doubles interaction. Stub , mock and spy will be discussed for implementation in details in next upcoming chapters.

## 4.8 Unit testing rules

There are rules applicable to make tests useful for long run.

1. Isolated
2. Independent order of execution
3. Repeatable

### 4.8.1 Isolated

A unit test executes within isolation. A code that executed well under framework cannot be guaranteed to work perfectly well under isolated cases.

All tests are required to work independently. It should be affected by other simultaneous tests.

Databases are common example of shared resource. You can try to run in serial order.

Use local SQL database for older version code. Use **[Setup] → [Test] → [Teardown]** to set, test code, reset database. respectively.

### 4.8.2 Independent order of execution

All tests are required to execute independent of order. If tests require a specific order for proper test results then your tests are brittle.

A required order execution of unit tests can indicate grouping under integrated tests.

### 4.8.3 Repeatable

A unit test require to be repeatable.It has to provide consistent results. Ambiguous tests results have created trust issues.

A unit test should focus on testing behavior instead of mechanism details.

You create unit tests to offload manual testing to automation. If tests are required to be executed manually, then it has scaling issue.

## 4.9 Writing NUnit 3 tests

Starting to write TDD style requires you to have a clear thought for brainstorming, As you write tests you are clamping down details. Structure is required and response is thought out

Consider a currency conversion scenario to convert amount in USD to INR. You set amount in USD and provide a converted amount based on currency rate.

### 4.9.1 Writing Unit tests

**Focus:** Test a single method in isolation.

**Example:**

- Method: decimal ConvertCurrency(decimal amount,decimal rate)
- Test: If the exchange rate is 0.9, then ConvertCurrency(100, 0.9) should return 90.
- Scope: Focus on pure math calculation, no API calls or UI.

```
public class ConversionLogicTests
{
    [Test]
    public void Given_exchangeRate_When_AmountIsProvided_Then_ReturnsCalculation()
    {
        //Arrange
        decimal amount = 100M;
        decimal rate = 0.9M;
        decimal expected = 90M;
        var conversionLogic = new ConversionLogic();
        //Act
        var calculatedAmount = conversionLogic.ConvertCurrency(amount, rate);
        //Assert
        Assert.That(calculatedAmount, Is.EqualTo(expected));
    }
}
```

Fig 4.9.1.a – Unit tests

A method provides input for amount and rate

```
public class ConversionLogic
{
    public decimal ConvertCurrency(decimal amount, decimal rate)
    {
        return amount * rate;
    }
}
```

Fig 4.9.1.b – Method implementation

## 4.9.2 Writing Integration tests

**Focus:** Test how modules interact.

**Example:**

- Modules: ExchangeRateService + ConversionLogic
- Test:
    - ExchangeRateService fetches the latest USD→INR rate (say 91.51).
    - ConversionLogic applies it to 100 USD.
    - Result should be 9151 INR.
- Scope: Service + logic working together, but not the full UI.

```
public class ExchangeBusinessTests
{
    [Test]
    public void Given_CurrencyRateMappingIsAvailable_when_ConvertsAmountInUSD_Then_ReturnsAmountInINR()
    {
        //Arrange
        string from = "USD";
        string to = "INR";
        decimal amount = 10;
        decimal expectedAmount = 915.13M;
        var moq = new Moq.Mock<IExchangeRateService>();
        moq.Setup(rate => rate.CurrencyRate(from,to)).Returns(91.513M);
        IExchangeRateService exchangeRateService = moq.Object;

        var conversionLogic = new ConversionLogic();
        var exchangeBusiness = new ExchangeBusiness(exchangeRateService, conversionLogic);
        //Act
        var convertedAmount = exchangeBusiness.Convert(from, to, amount);
        //Assert
        Assert.That(convertedAmount, Is.EqualTo(expectedAmount));
    }
}
```

Fig 4.9.2.a – Integration tests

For Integration test - 4.9.2.a – you are checking response based on two component interactions

1. Currency rate service
2. Current amount calculation

```
public class ExchangeBusiness
{
    private readonly IExchangeRateService _exchangeRateService;
    private readonly ConversionLogic _conversionLogic;
    public ExchangeBusiness(IExchangeRateService exchangeRateService, ConversionLogic conversionLogic)
    {
        _exchangeRateService = exchangeRateService;
        _conversionLogic = conversionLogic;
    }
    public decimal Convert(string fromCurr, string toCurr, decimal amount)
    {
        var currencyRate = _exchangeRateService.CurrencyRate(fromCurr, toCurr);
        return _conversionLogic.ConvertCurrency(amount, currencyRate);
    }
}
```

Fig 4.9.2.b – Method implementation for integration

The method implementation for sequence of component interaction more than one thus it exceeds one to classify as integration tests instead creating of two separate unit tests.

## 4.9.3 Writing E2E tests

**Focus:** Test the full user journey.

**Example:**

- Scenario: A user opens the currency converter web app, selects **USD → INR**, enters 10, clicks "Convert," and sees 915.13 INR displayed.
- Test: Verify that:
  - The UI accepts input.
  - The backend fetches the correct exchange rate.
  - The conversion is calculated.
  - The result is shown to the user.
- Scope: Browser automation simulates real user interaction across UI, backend, and output.

Playwright test includes a set of steps

1. Load website
2. Fill amount - 10
3. Fill Currency - USD
4. Fill Currency – INR
5. Submit Page
6. Read message

```
public class PriceE2ETests
{
    private IBrowser _browser;
    private IPage _page;
    private IPlaywright _playwright;

    [SetUp]
    public async Task Setup()
    {
        _playwright = await Playwright.CreateAsync();
        _browser = await _playwright.Chromium.LaunchAsync(new BrowserTypeLaunchOptions
        {
            Headless = false // Set to true for headless mode
        });

        var context = await _browser.NewContextAsync();
        _page = await context.NewPageAsync();
    }
```

Fig 4.9.3.a – E2E tests

This section highlights a setup executed by framework before each test.

```
[Test]
public async Task ConvertPriceOnline()
{
    //Arrange - SetUp

    string url = $"http://localhost:5242/Price";
    await _page.WaitForURLAsync(url);

    await _page.FillAsync("#Amount", "100");
    await _page.FillAsync("#From", "USD");
    await _page.FillAsync("#To", "INR");

    await _page.ClickAsync("input.btn.btn-primary");

    string txt = await _page.TextContentAsync("div.text-info");

    // Assert
    Assert.That(txt, Is.Not.Empty);
}
```

Fig 4.9.3.b – E2E tests automation

This section contains automation test for playwright executed as browser instruction to mimic user operation.

```
[TearDown]
public async Task Teardown()
{
    await _browser.CloseAsync();
    _playwright.Dispose();
}
}
```

Fig 4.9.3.c – E2E tests tear down

This section highlights a disposal code executed by framework before each test.

E2E tests are useful for real time feedback instead of assuming all dependencies are working.

# 4.10 Exercises

Each exercise level allows to test a fundamental concept

## GitHub Repository

Repo URL: https://github.com/abhishekcbhattacharya-svg/BreakItMakeIt_Exercises_v2

Folder: *BreakItMakeIt_Exercises/Chapter_04_TestTypes*

**Steps**

git clone https://github.com/abhishekcbhattacharya-svg/BreakItMakeIt_Exercises_v2.git

cd BreakItMakeIt_Exercises_v2

Use VS Code or Visual Studio 2022 C# IDE

**Level 1: Unit**

Topics:

1. Write your first unit test,
2. Focus on writing test

Challenge: Create method implementation with help of provided test information.

**Level 2: Integration**

Topics:

1. Handle edge case
2. Create logic to handle range of input and combination

Challenge: Create method implementation with help of provided test information and helper class as real life scenario.

**Level 3: E2E**

Topics:

1. Implement Playwright code
2. Interact with website input
3. Check interaction response for validation

Challenge: Create E2E tests for existing website to assess behavior validation.

## 4.11 MCQ

1. What does the test pyramid suggest about the distribution of tests?

A. More E2E tests than unit tests
B. Equal number of unit, integration, and E2E tests
C. More unit tests than integration and E2E tests
D. No unit tests, only integration tests

**Answer**: C – The pyramid emphasizes having more unit tests at the base, fewer integration tests, and the least E2E tests.

2. Which type of test focuses on validating a single class or method in isolation?

A. Integration test
B. Unit test
C. Component test
D. End-to-End test

**Answer**: B – Unit tests isolate one class/method without external dependencies.

3. What is the main purpose of integration testing?

A. To validate individual methods
B. To test external services independently
C. To ensure multiple modules interact correctly
D. To simulate user behavior across the system

**Answer**: C – Integration tests verify interactions among components and modules.

4. Which type of test is considered the most brittle but provides real-time validation of user flows?

A. Unit test
B. Integration test
C. Component test
D. End-to-End test

**Answer**: D – E2E tests simulate user operations and are prone to breaking with UI changes.

5. In unit testing, what is commonly used to replace external dependencies?

A. Real database connections
B. Stubs, mocks, and spies
C. Full integration setups
D. Manual user inputs

**Answer**: B – Unit tests rely on stubs, mocks, and spies to isolate dependencies.

6. Which testing goal verifies that a method produces the correct return value?

A. State change testing
B. Output testing
C. Component interaction testing
D. Regression testing

**Answer**: B – Output testing checks the correctness of returned values.

7. What does component testing primarily validate?

A. Internal business rules only
B. External dependencies like cloud storage or email services
C. User interface interactions
D. Framework code execution

**Answer**: B – Component tests validate external dependencies individually.

8. Which analogy best describes unit testing scope?

A. A bulb lighting up when switched on
B. A car engine tested with all parts
C. A user navigating a website
D. A team collaborating on multiple modules

**Answer**: A – Unit testing is like testing a bulb in isolation with a socket and current source.

9. Which test type verifies aggregated outcomes assuming all dependencies work correctly?

A. Unit test
B. Component test
C. Integration test
D. Regression test

**Answer**: C – Integration tests focus on collective success across modules.

10. Why are E2E tests often described as "insurance" for projects?

A. They are the fastest to execute
B. They validate isolated methods
C. They protect against unintended changes in legacy systems
D. They eliminate the need for unit tests

**Answer**: C – E2E tests act as insurance against modifications in existing codebases.

# Chapter 5: Tools for the Trade

## Learning outcomes

1. Boolean algebra and relation with code.
2. Arrange, Act, Assert pattern for Test code.
3. Moq code examples for mock/spy implementation

## 5.1 Boolean algebra

| Operator | Symbol | Equation | Description |
|---|---|---|---|
| **AND** | . | A.B = 1 | Returns true if both are true |
| **OR** | + | A + B = 1 | Returns true if at least one is true |
| **NOT** | ! | !A | Inverts the truth value (negation) |

### AND operation

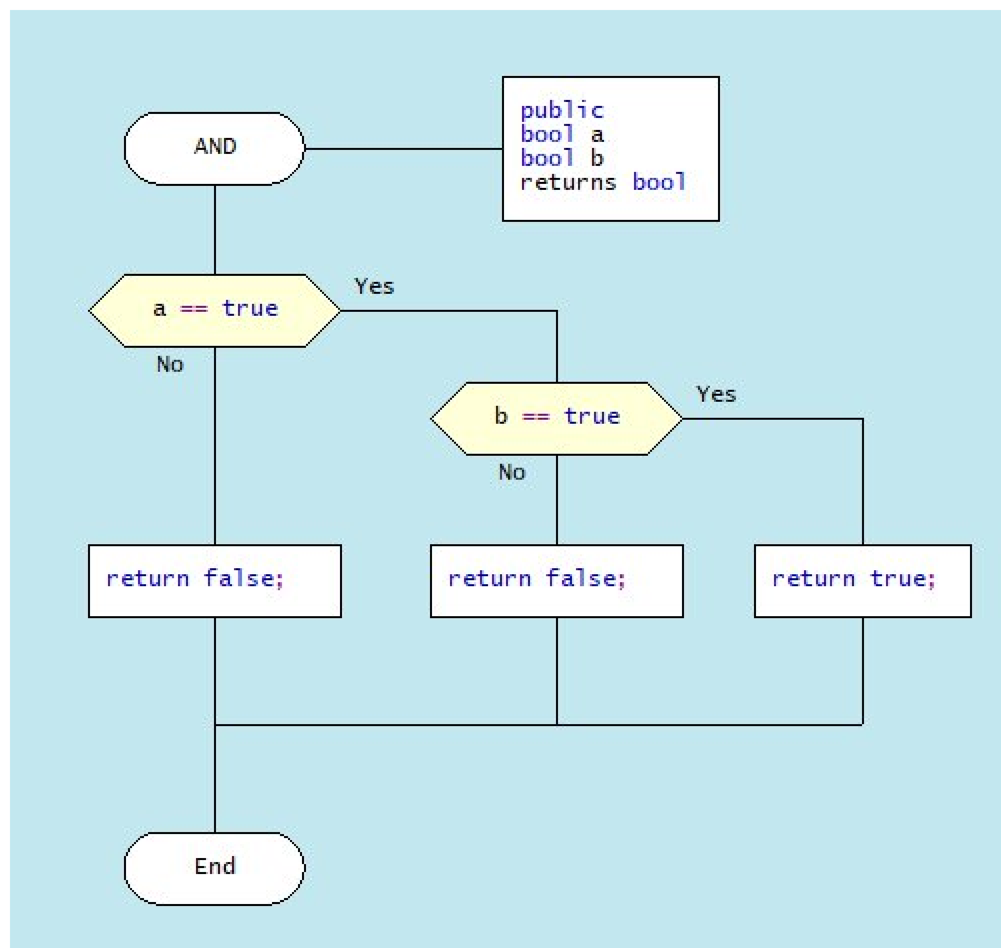



Fig 5.1.a – Flowchart for AND operation

A.B = 1 is a AND boolean expression where 1 represents TRUE, 0 represents FALSE.

Each Individual input A,B require TRUE then it provide TRUE.

If any input A or B is FALSE then expression returns FALSE.

| A | B | A.B |
|---|---|---|
| **0** | 0 | 0 |
| **0** | 1 | 0 |
| **1** | 0 | 0 |
| **1** | 1 | 1 |

## OR operation

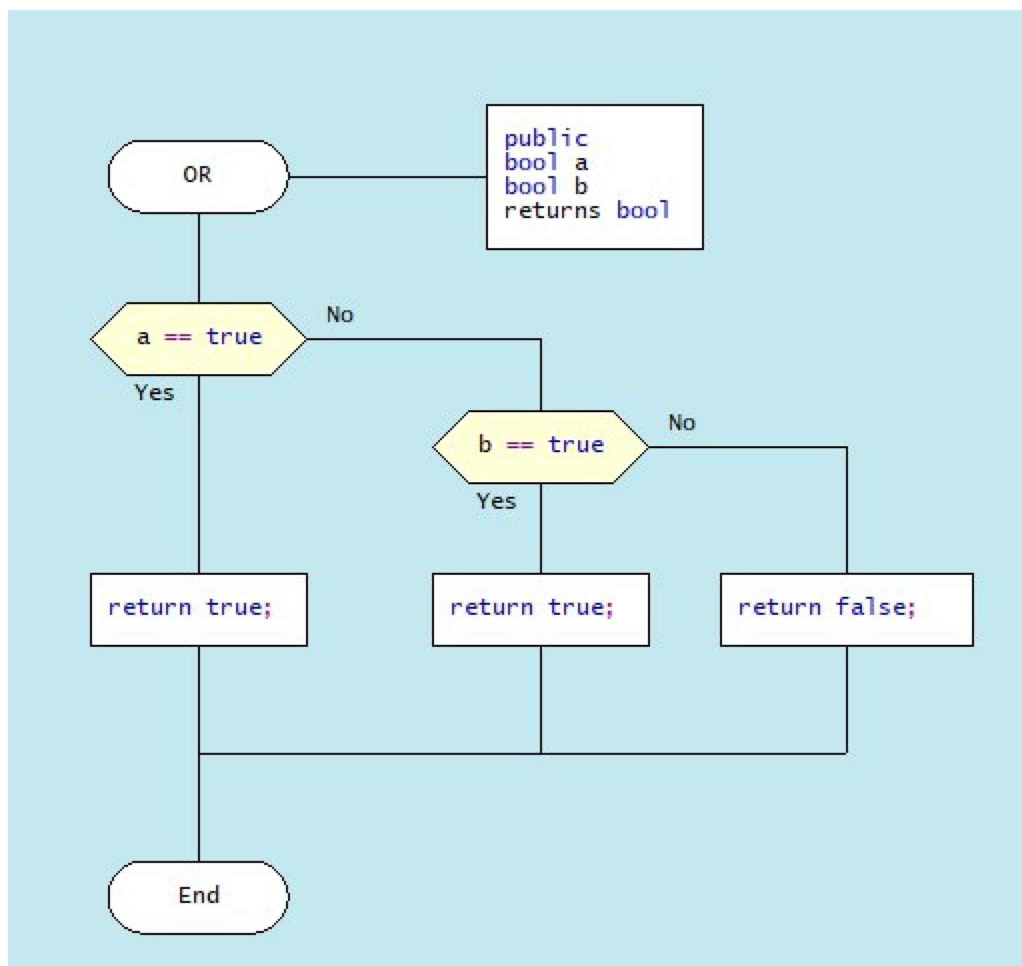



Fig 5.1.b – Flowchart for OR operation

A + B = 1 is a OR boolean expression where 1 represents TRUE, 0 represents FALSE.

Each Individual input A,B require FALSE then it provide FALSE.

If any input A or B is FALSE then expression returns FALSE.

| A | B | A+B |
|---|---|---|
| **0** | 0 | 0 |
| **0** | 1 | 0 |
| **1** | 0 | 0 |
| **1** | 1 | 1 |

## NOT operation

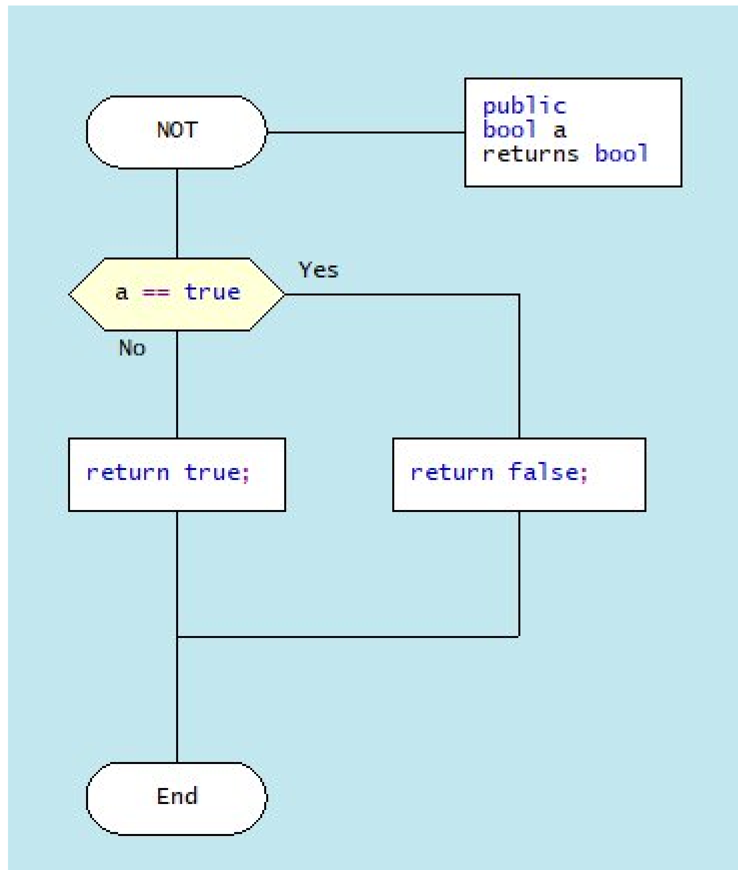



Fig 5.1.c – Flowchart for NOT operation

!A is a NOT boolean expression where 1 represents TRUE, 0 represents FALSE.

If A is TRUE then NOT boolean expression returns FALSE then A is FALSE then NOT boolean expression return TRUE.

| A | !A |
|---|---|
| **0** | 1 |
| **1** | 0 |

Note: Boolean expressions are represented by if, switch, validation expressions for loop in flowchart

Flowcharts are important to understand code logical flow. All conditional elements like if , switch , loop execution parts take decision to execute a state change based on value.

## 5.1.2 Basic Laws & Identities

**Identity Laws**

$A + 0 = A$

$A \cdot 1 = A$

**Null (Domination) Laws**

$A + 1 = 1$

$A \cdot 0 = 0$

**Idempotent Laws**

$A + A = A$

$A \cdot A = A$

**Complement Laws**

$A + !A = 1$

$A \cdot !A = 0$

**Double Negation**

$!(!A) = A$

Above laws are useful to reduce complexity for single term related boolean expression.

## 5.1.3 DeMorgan’s Theorems

**Negation**

$!(A \cdot B) = !A + !B$

$!(A + B) = !A \cdot !B$

Note: Negation is important to identify individual error from collective success.

A simple example of $\underline{A.B.C = 1}$ is boolean expression describing a combination of A, B, C for collective success.

!(A.B.C) converts to !A + !B +!C = 0

### 5.1.4 Logic Laws

**Commutative Laws**

A + B = B + A

A · B = B · A

**Associative Laws**

A + (B + C) = (A + B) + C

A . (B · C) = (A · B) · C

**Distributive Laws**

A · (B + C) = A · B + A · C

A + (B · C) = (A + B) · (A + C)

Above laws are useful to reduce complexity of boolean expressions. It helps to reduce boolean expression for simpler code interpretation.

We will discuss creating boolean expression for existing code in future chapters.

**Relatable fields and implementations**

Conditional Statements: Boolean expressions instruct **if**, **switch** , **while**, and **for** loops in programming languages.

Search Algorithms: Boolean operators (**AND**, **OR**, **NOT**) are used in search queries to filter results.

Database Queries: SQL uses Boolean logic to combine conditions (WHERE **A AND B**, WHERE **A OR B**)

## 5.2 NUnit 3

This is a popular framework for Test implementation for .NET Core ecosystem. A implementation for Tests is foundation for TDD approach. A project template includes tests based on arrange, act and assert pattern

**Installation**

Visual Studio (Package Manager Console)

*Install-Package NUnit -Version 3.**
*Install-Package NUnit3TestAdapter*

*Install-Package Microsoft.NET.Test.Sdk*

.NET CLI

*dotnet add package NUnit --version 3.**
*dotnet add package NUnit3TestAdapter*
*dotnet add package Microsoft.NET.Test.Sdk*

Setup in csproj file project

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <IsPackable>false</IsPackable>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="NUnit" Version="3.*" />
    <PackageReference Include="NUnit3TestAdapter" Version="3.*" />
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="17.*" />
  </ItemGroup>

</Project>
```

This .csproj configuration works seamlessly with dotnet test and Visual Studio's Test Explorer. The following helps to setup project dependencies via PMC or .NET CLI or csproj file.

**Test project execution**

Execute tests using CLI

*dotnet test*

**Running Unit Tests in Visual Studio (IDE)**

Using Test Explorer

1. Open Test Explorer (Test > Test Explorer menu).
2. Run all tests: Click Run All Tests.
3. Run selected tests: Right-click a specific test/class and choose Run Selected Tests.
4. Debug tests: Right-click and choose Debug Selected Tests.
5. Filter tests: Use search or group by outcome, duration, or traits.

Using VSTest.Console.exe (Modern Tool)

Located in: %Program Files (x86)%\Microsoft Visual Studio\<version>\<edition>\Common7\IDE\CommonExtensions\Microsoft\TestWindow\VSTest.Console.exe

*vstest.console.exe MyTests.dll*

```
using NUnit.Framework;

[TestFixture]
public class ProjectTests
{
    [SetUp]
    public void Init()
    {
        //setup before every test
        //initialise
    }

    [TearDown]
    public void Cleanup()
    {
        //cleanup after every test
        //dispose
    }

    [Test]
    public void TestFunctionality()
    {
        //Arrange
        //Act
        //Assert
    }
}
```

Fig 5.2.a – NUnit 3 project template

**Fig 5.2.a** indicates common usage you use for write tests, attributes are applied on class and methods. **[TestFixture]** is a class level attribute for recognizing tests by Test explorer within IDE.

The order of execution is **[Setup] -> [Test] -> [TearDown]** for each test.

## 5.2.1 Arrange – Act – Assert pattern

Arrange – Act –Assert pattern includes three subsections:

**Arrange**

Developer is responsible to setup all dependencies for code under test. We are defining scenario that will create response to executed code.

Basic: Simple class do not have dependencies - Calculator class to add two numbers. Refer 5.2.1 Arrange section.

Complex: Define all dependencies and pass using dependency injection based on constructor / method parameter.

For optional dependencies, use null values

For required dependencies, Select Stub / Mock / Spy based on scenario to force a pre-configured behavior

**Act**

This includes method execution with carefully set information from arrange set to simulate a scenario. Refer 5.2.1.a Act section.

**Assert**

Developer needs to verify based response for success confirmation.

Output

Calculator is a simple class adding two numbers and returning a sum. A comparison of actual Sum to expected sum determines successful test. Refer 5.2.1.a Assert section.

Basic example

```
using NUnit.Framework;

[TestFixture]
public class CalculatorTests
{
    [Test]
    public void Add_TwoNumbers_ReturnsSum()
    {
        // Arrange
        var calculator = new Calculator();
        int a = 2, b = 3, expectedSum = 5;

        // Act
        int actualSum = calculator.Sum(a, b);

        // Assert
        Assert.That(actualSum, Is.EqualTo(expectedSum));
    }
}
```

Fig 5.2.1.a – NUnit 3 output verification example

For given example for test code, sum output returned by calculator code is verified.

State Change

A bank account class with transaction like check balance, deposit and withdrawal operations. A code test can work checking balance before and after deposit operation.

You may not be notified like proving output for verification

State changes focus on comparing internal data that changes because of interaction. Complexity related to measuring information change is lower than spy.

A simplified example is checking database before and after transaction It is accessing public level API access.

Intermediate example

```
using NUnit.Framework;

[TestFixture]
public class BankAccountTests
{
    [Test]
    public void Deposit_PositiveAmount_ChangesBalanceState()
    {
        // Arrange
        var account = new BankAccount();
        decimal initialBalance = account.Balance;
        decimal depositAmount = 100m;

        // Act
        account.Deposit(depositAmount);

        // Assert (before state)
        Assert.That(initialBalance, Is.EqualTo(0), "Initial balance should be zero");

        // Assert (after state)
        Assert.That(account.Balance, Is.EqualTo(initialBalance + depositAmount),
            "Balance should increase by deposit amount");
    }
}
```

Fig 5.2.1.b – NUnit 3 state change verification example

For given code example, starts with initial balance check.Act section processes Add balance operation and is expected to create a balance change

<u>Component interaction</u>

This is most difficult version of code testing. A developer is aiming to measure execution calls assuming it will be success.

A testable class will have clear dependency injection for internal component execution. Most of time developers use spy implementation to measure all internal execution details. Moq is a popular nuget library to setup dependencies for mock / spy purpose.

Advanced example

For given code example in Fig 5.2.1.c you setup state change comparison for amount and assert spy based method execution count for email confirmation.

```csharp
using NUnit.Framework;
using Moq; // Popular mocking library

[TestFixture]
public class BankAccountTests
{
    [Test]
    public void Deposit_PositiveAmount_ChangesBalanceAndSendsEmail()
    {
        // Arrange
        var mockEmailService = new Mock<IEmailService>();
        var account = new BankAccount(mockEmailService.Object);
        decimal initialBalance = account.Balance;
        decimal depositAmount = 100m;

        // Act
        account.Deposit(depositAmount);

        // Assert (state change)
        Assert.That(account.Balance, Is.EqualTo(initialBalance + depositAmount));

        // Assert (interaction)
        mockEmailService.Verify(
            e => e.SendConfirmation(It.Is<string>(msg => msg.Contains("Deposit of 100"))),
            Times.Once,
            "Email confirmation should be sent once after deposit"
        );
    }
}
```

Fig 5.2.1.c – NUnit 3 state change verification with email notification example

In comparison to state change verification, code executes mock.Verify() for one time execution.

In  case of success, code executes and no output / exception is thrown.

In case of failure, test code sees exception thrown.

1. If the method was **not called the expected number of times**, Moq raises a MockException.
2. The exception message is **very descriptive**:
3. It shows the **expression** developer was verifying (e.g., e => e.SendConfirmation(...)).
4. It states the **expected invocation count** (e.g., Times.Once).
5. It shows the **actual invocation count** (e.g., 0 times, or 2 times).

## 5.2.2 Common attributes with examples

```csharp
// Mock logic for execution
public class SecurityService : IDisposable
{
    public bool IsLocked { get; } = true;
    public bool ValidateUser(string user) => user == "admin";
    public void ConnectToLegacy() => throw new Exception("Offline");
    public void Dispose() { /* Cleanup */ }
}
```

Fig 5.2.2.a – NUnit 3 Stub class example

Service class example created to demonstrate possibility.

```csharp
// [TestFixture] identifies the class as a test container
[TestFixture]
public class SecurityServiceTests
{
    private SecurityService _service;

    [OneTimeSetUp]
    public void SetupGlobalResources()
    {
        // Executed once before all tests
        Console.WriteLine("Connecting to Global Auth Provider...");
    }

    [SetUp]
    public void CreateServiceInstance()
    {
        // Executed before every [Test] or [TestCase]
        _service = new SecurityService();
    }

    [Test]
    [Category("Unit")] // [Category] groups this as a Unit test
    public void Service_ByDefault_IsLocked()
    {
        // [Test] marks this as an executable test
        Assert.That(_service.IsLocked, Is.True);
    }

    [TestCase("admin", true)]
    [TestCase("guest", false)]
    [TestCase("", false)]
    public void ValidateUser_ReturnsCorrectAccess(string username, bool expectedAccess)
    {
        // [TestCase] runs this method 3 times with different data
        bool result = _service.ValidateUser(username);
        Assert.That(result, Is.EqualTo(expectedAccess));
    }
```

```
    [Test]
    [Ignore("Legacy system is offline")] // [Ignore] skips this test
    public void TestLegacyConnection()
    {
        _service.ConnectToLegacy();
    }

    [TearDown]
    public void ResetService()
    {
        // Executed after every test
        _service.Dispose();
    }

    [OneTimeTearDown]
    public void TeardownGlobalResources()
    {
        // Executed once after everything is done
        Console.WriteLine("Disconnecting from Global Auth Provider...");
    }
}
```

Fig 5.2.2.b – NUnit 3 Test class example

**1. Test Fixture Management**

*[TestFixture]*: Marks a class that contains test methods.

Use Case: Essential for the NUnit runner to identify which classes to scan for executable tests.

*[OneTimeSetUp]*: A method that runs exactly once before any tests in the class begin.

Use Case: Used for "heavy" initialization, such as starting an external service, connecting to a database, or generating a shared data set.

*[OneTimeTearDown]*: A method that runs once after all tests in the fixture have finished.

Use Case: Used for global cleanup, like closing database connections, releasing resources or deleting temporary directories.

**2. Individual Test Execution**

*[Test]*: Marks a specific method as a test to be executed.

Use Case: The standard attribute for verifying a single unit of logic where no external parameters are required.

*[SetUp]*: Runs before every individual test method in the class.

Use Case: Ensures a "clean slate" for every test, such as resetting the state of an object or clearing a list.

*[TearDown]*: Runs after every individual test method concludes.

Use Case: Used to clean up resources specific to a single test run, ensuring no side effects leak into the next test.

**3. Data-Driven and Metadata Attributes**

*[TestCase(...)]*: Facilitates data-driven tests by passing parameters into the test method.

Use Case: Validating a single logic path (like a math formula or string parser) against multiple sets of input data and expected results.

*[Ignore("reason")]*: Instructs the runner to skip the test and provides a reason why.

Use Case: Useful when a feature is temporarily disabled, a bug is known but not yet fixed, or a test is flaky and needs investigation.

*[Category("tag")]*: Assigns a label or group name to a test.

Use Case: Allows developers to run specific subsets of tests, such as "SmokeTests," "Integration," or "FastTests," in CI/CD pipelines.

## 5.2.3 Common assertions with examples

```
[TestFixture]
public class AssertionTests
{
    [Test]
    public void AreEqual_ShouldPass_WhenValuesMatch()
    {
        int result = 2 + 3;
        Assert.AreEqual(5, result); // Checks equality
    }

    [Test]
    public void IsTrue_ShouldPass_WhenConditionIsTrue()
    {
        int age = 20;
        Assert.IsTrue(age >= 18); // Asserts that condition is true
    }

    [Test]
    public void IsFalse_ShouldPass_WhenConditionIsFalse()
    {
        string password = "SecurePass123";
        Assert.IsFalse(password.Contains(" ")); // Asserts that condition is false
    }

    [Test]
    public void IsNull_ShouldPass_WhenObjectIsNull()
    {
        User user = new User(); // Assume User has Email property
        Assert.IsNull(user.Email); // Checks if object is null
    }
```

```
    [Test]
    public void IsNotNull_ShouldPass_WhenObjectIsCreated()
    {
        UserService service = new UserService();
        var user = service.GetUserById(1);
        Assert.IsNotNull(user); // Checks if object is not null
    }

    [Test]
    public void Throws_ShouldPass_WhenExceptionIsThrown()
    {
        Assert.Throws<DivideByZeroException>(() =>
        {
            int result = 10 / 0;
        }); // Verifies that exception of type T is thrown
    }

    [Test]
    public void That_ShouldPass_WhenValuesAreEqual()
    {
        string name = "NUnit";
        Assert.That(name.Length, Is.EqualTo(5)); // Fluent style assertion
    }
}
```

Fig 5.2.3.a – NUnit 3 Test assertion class example

**Assert.AreEqual(expected, actual)**

Purpose: Verifies that two values are equal.

Typical Use: Comparing numeric results, string values, or object properties to ensure they match expected outcomes.

**Assert.IsTrue(condition)**

Purpose: Ensures that a given condition evaluates to true.

Typical Use: Validating logical expressions, such as checking if a number is greater than another or if a string starts with a certain prefix.

**Assert.IsFalse(condition)**

Purpose: Ensures that a given condition evaluates to false.

Typical Use: Confirming that invalid states or unwanted conditions do not occur, such as ensuring a password does not contain spaces.

**Assert.IsNull(object)**

Purpose: Checks that an object reference is null.

Typical Use: Validating that an object has not been initialized or a property has not been set.

**Assert.IsNotNull(object)**

Purpose: Checks that an object reference is not null.

Typical Use: Ensuring that a service returns a valid object or that a variable has been properly initialized.

**Assert.Throws<T>(() => ...)**

Purpose: Verifies that a specific exception type is thrown during execution.

Typical Use: Testing error handling logic, such as division by zero or passing invalid arguments.

Assert.That(actual, Is.EqualTo(expected))

Purpose: Provides a fluent, readable style for assertions.

Typical Use: Writing more expressive tests, often used for comparisons like equality, greater than, or string length checks.

## 5.3 Moq

This is a popular class library that provides mechanism to setup mock / spy allowing measuring information internal.

**Installation**

1. Using Visual Studio (Package Manager Console)

Open Tools → NuGet Package Manager → Package Manager Console.

Run in powershell:

*Install-Package Moq -Version 4.20.72*

2. Using .NET CLI

*dotnet add package Moq --version 4.20.72*

**Documentation**

1.Creating a Mock

```
// Basic mock of an interface
var mockService = new Mock<IService>();

// setup

// Access mocked object
IService service = mockService.Object;
```

Explanation:

Mock<IService> creates a proxy object implementing IService. Developer is configuring proxy object to return specific response, throw exception.

mockService.Object gives developer the instance to inject into your code under test.

2.Common Methods

<u>Setup</u>

```
mockService.Setup(x => x.Get(1)).Returns(new Item { Id = 1 });
```

Defines behavior for a method/property. Here, calling Get(1) returns a specific Item.

<u>Returns</u>

```
mockService.Setup(x => x.Count).Returns(5);
```

Specifies return values. Useful for properties or deterministic methods.

<u>Throws</u>

```
mockService.Setup(x => x.Save()).Throws<Exception>();
```

Simulates exceptions to test error-handling logic.

<u>Verify</u>

```
mockService.Verify(x => x.Update(It.IsAny<Item>()), Times.Once);
```

Checks that a method was called with expected arguments and frequency.

<u>Callback</u>

```
int captured = 0;
mockService.Setup(x => x.Add(It.IsAny<int>())).Callback<int>(i => captured = i);
```

Captures arguments or triggers side effects when invoked.

<u>Raise</u>

```
mockService.Raise(x => x.Completed += null, EventArgs.Empty);
```

Simulates event firing from the dependency.

<u>SetupProperty</u>

```
mockService.SetupProperty(x => x.Name, "default");
```

Auto-stubs a property with getter and setter. Useful for stateful mocks.

VerifyNoOtherCalls

```
mockService.VerifyNoOtherCalls();
```

Ensures no unexpected invocations occurred beyond verified ones.

3.Setting Up Methods

```
mockService.Setup(x => x.Find(It.Is<int>(i => i > 0))).Returns((int id) => new Item { Id = id });
```

It.IsAny<T>() → matches any argument.

It.Is<T>(predicate) → matches arguments with custom logic.

4.Returning Values and Exceptions

Single Return

```
mockService.Setup(x => x.Sum(2, 3)).Returns(5);
```

Explanation: Always returns 5 when called with (2,3).

Sequence of Returns

```
mockService.SetupSequence(x => x.Next())
.Returns(1)
.Returns(2)
.Throws<InvalidOperationException>();
```

Explanation : First call → returns 1.

Second call → returns 2.

Third call → throws exception.

Throw Exception

```
mockService.Setup(x => x.Delete(0))
.Throws<ArgumentException>();
```

Explanation: Calling Delete(0) will throw ArgumentException.

5. Verifying Calls

```
mockService.Verify(x => x.Process(), Times.Never);   // Ensure not called
mockService.Verify(x => x.Process(), Times.Once);    // Ensure called once
mockService.Verify(x => x.Process(), Times.Exactly(3)); // Ensure called 3 times
mockService.Verify(x => x.Process(), Times.AtLeastOnce); // Ensure called at least once
```

Explanation: Verifies call frequency to ensure expected interactions.

6. Callbacks

```
int captured = 0;
mockService.Setup(x => x.Add(It.IsAny<int>()))
.Callback<int>(i => captured = i);
```

Explanation: Captures the argument passed into Add for later assertions.

7. Raising Events

```
mockService.Raise(x => x.Updated += null, new UpdateEventArgs("data"));
```

Explanation: Simulates event firing with custom event args.

8. Behavior Modes

```
var strictMock = new Mock<IService>(MockBehavior.Strict);
```

Loose (default): unexpected calls return defaults (null, 0).

Strict: unexpected calls throw MockException.

9. Advanced Features

Setup All Properties

```
mockService.SetupAllProperties();
```

Auto-implements all properties with default values.

Async Support

```
mockService.Setup(x => x.GetAsync(1)).ReturnsAsync(new Item { Id = 1 });
mockService.Setup(x => x.SaveAsync()).ThrowsAsync(new InvalidOperationException());
```

Supports asynchronous methods.

Custom Matchers

```
mockService.Setup(x => x.GetName(It.IsRegex("^[A-Z].*")))
.Returns("ValidName");
```

Matches arguments using regex or ranges.

**Disadvantages**

1. Moq requires virtual methods or interfaces; It cannot mock sealed classes or static methods without wrappers
2. Over-specification leads to brittle tests that break on refactoring
3. Expression trees can obscure compile-time errors until runtime
4. Complex setups become hard to read and maintain for large interfaces
5. Performance overhead in large test suites due to proxy generation

## 5.4 Fluent Validation

FluentValidation is a popular NuGet package used for class validation. It is important tool for instance property validation. You are able to execute validation from framework including test framework. It sits within definition of class library thus it can be executed without code related quirks.

Model validation within MVC / WebAPI framework is a dependency, production code requires additional work for execution within test framework.

FluentValidation allow developers to write strongly-typed validation logic for their classes. Creating separate validation class allows to follow Single Responsibility principle.

**Core Concepts**

- **Validator Class**: You create a class that inherits from AbstractValidator<T> where T is your model.
- **RuleFor**: Defines validation rules for each property.
- **Chaining**: You can chain multiple rules for a single property.
- **Custom Rules**: Supports custom logic via Must, When, and Unless.

UserRegisteration class with corresponding UserRegisterationValidator class. It includes rules definitions within constructor. Refer **Fig 5.4.a** for code example.

**Important Features**

- Separation of Concerns: Keeps validation logic out of your models.
- Integration: Works seamlessly with ASP.NET Core, MVC

**Common validators**

```
using FluentValidation;

public class UserRegistration
{
    public string Username { get; set; }
    public string Email { get; set; }
    public string CreditCardNumber { get; set; }
    public int Age { get; set; }
    public string Password { get; set; }
    public int Score { get; set; }
    public string ConfirmPassword { get; set; }
}
```

Fig 5.4.a – Fluent validation test class example

Class includes property definitions that will be subjected to rules.

```csharp
public class UserRegistrationValidator : AbstractValidator<UserRegistration>
{
    public UserRegistrationValidator()
    {
        // NotEmpty
        RuleFor(user => user.Username)
            .NotEmpty().WithMessage("Username cannot be empty.");

        // NotNull
        RuleFor(user => user.Email)
            .NotNull().WithMessage("Email cannot be null.");

        // Length(min, max)
        RuleFor(user => user.Username)
            .Length(3, 20).WithMessage("Username must be between 3 and 20 characters.");

        // MinimumLength
        RuleFor(user => user.Password)
            .MinimumLength(8).WithMessage("Password must be at least 8 characters long.");

        // MaximumLength
        RuleFor(user => user.Password)
            .MaximumLength(20).WithMessage("Password must not exceed 20 characters.");
```

```csharp
        // Matches (regex)
        RuleFor(user => user.Password)
            .Matches(@"^(?=.*[A-Z])(?=.*\d).+$")
            .WithMessage("Password must contain at least one uppercase letter and one number.");

        // EmailAddress
        RuleFor(user => user.Email)
            .EmailAddress().WithMessage("Invalid email format.");

        // CreditCard
        RuleFor(user => user.CreditCardNumber)
            .CreditCard().WithMessage("Invalid credit card number.");

        // InclusiveBetween
        RuleFor(user => user.Age)
            .InclusiveBetween(18, 99).WithMessage("Age must be between 18 and 99.");

        // GreaterThan
        RuleFor(user => user.Score)
            .GreaterThan(50).WithMessage("Score must be greater than 50.");

        // LessThan
        RuleFor(user => user.Score)
            .LessThan(100).WithMessage("Score must be less than 100.");

        // Equal
        RuleFor(user => user.ConfirmPassword)
            .Equal(user => user.Password).WithMessage("Passwords must match.");

        // NotEqual
        RuleFor(user => user.Password)
            .NotEqual(user => user.Username).WithMessage("Password cannot be the same as username.");
    }
}
```

Fig 5.4.b – Fluent validation class example

Instead of using [DataAnnotations] directly on properties, you encapsulate rules in dedicated validator classes.

1. NotEmpty – Ensure a field isn't blank

*RuleFor(user => user.Username)*
*.NotEmpty().WithMessage("Username cannot be empty.");*

Use this when a property must contain some value (e.g., username).

2. NotNull – Ensure a property isn't null

*RuleFor(user => user.Email)*
*.NotNull().WithMessage("Email cannot be null.");*

Useful for required fields that must exist (e.g., email).

3. Length(min, max) – Restrict string length

*RuleFor(user => user.Username)*
*.Length(3, 20).WithMessage("Username must be between 3 and 20 characters.");*

Perfect for usernames or codes with fixed length ranges.

4. MinimumLength – Enforce minimum characters

*RuleFor(user => user.Password)*
*.MinimumLength(8).WithMessage("Password must be at least 8 characters long.");*

Commonly used for strong password policies.

5. MaximumLength – Limit maximum characters

*RuleFor(user => user.Password)*
*.MaximumLength(20).WithMessage("Password must not exceed 20 characters.");*

Prevents overly long input, e.g., comments or passwords.

6. Matches(regex) – Validate with patterns

```
RuleFor(user => user.Password)
.Matches(@"^(?=.*[A-Z])(?=.*\d).+$")
.WithMessage("Password must contain at least one uppercase letter and one number.");
```

Great for enforcing formats like phone numbers or password complexity.

7. EmailAddress – Validate email format

*RuleFor(user => user.Email)*
*.EmailAddress().WithMessage("Invalid email format.");*

Ensures proper email syntax.

8. CreditCard – Validate credit card numbers

```
RuleFor(user => user.CreditCardNumber)
.CreditCard().WithMessage("Invalid credit card number.");
```

Checks if the input looks like a valid card number.

9. InclusiveBetween(min, max) – Value within a range

```
RuleFor(user => user.Age)
.InclusiveBetween(18, 99).WithMessage("Age must be between 18 and 99.");
```

Useful for age, ratings, or bounded values.

10. GreaterThan(x) – Must be above a threshold

```
RuleFor(user => user.Score)
.GreaterThan(50).WithMessage("Score must be greater than 50.");
```

Ensures values exceed a minimum.

11. LessThan(x) – Must be below a threshold

```
RuleFor(user => user.Score)
.LessThan(100).WithMessage("Score must be less than 100.");
```

Restricts values to stay under a maximum.

12. Equal(x) – Must match another value

```
RuleFor(user => user.ConfirmPassword)
.Equal(user => user.Password).WithMessage("Passwords must match.");
```

Often used for confirm password fields.

13. NotEqual(x) – Must differ from another value

```
RuleFor(user => user.Password)
.NotEqual(user => user.Username).WithMessage("Password cannot be the same as username.");
```

Prevents insecure or duplicate values.

**Validation examples**

```
var user = new UserRegistration
{
    Username = "JD",
    Email = "invalid-email",
    CreditCardNumber = "1234",
    Age = 15,
    Password = "pass",
    Score = 120,
    ConfirmPassword = "different"
};

var validator = new UserRegistrationValidator();
var results = validator.Validate(user);

if (!results.IsValid)
{
    foreach (var error in results.Errors)
    {
        Console.WriteLine($"Property: {error.PropertyName}, Error: {error.ErrorMessage}");
    }
}
```

Fig 5.4.c – Fluent validation test code example

Developer executes Validate() method that allows to implement rules validation. The ValidationResult object in FluentValidation is holding

1. IsValid: A boolean property that tells developer if all rules passed (true) or if there were any failures (false).
2. Errors: A collection of ValidationFailure objects.
   Each failure contains:
   a. PropertyName: The name of the property that failed validation.
   b. ErrorMessage: The descriptive message explaining the failure.
   c. AttemptedValue: The actual value that caused the failure.
   d. Other metadata like Severity or ErrorCode.

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}

public class PersonValidator : AbstractValidator<Person>
{
    public PersonValidator()
    {
        RuleFor(x => x.Name).NotEmpty();
        RuleFor(x => x.Age).InclusiveBetween(18, 60);
    }
}
```

Fig 5.4.d – Fluent Class with corresponding validator – Example

### Conditional Validation

```
RuleFor(x => x.Name)
    .NotEmpty()
    .When(x => x.Age > 18);
```

Fig 5.4.e - Conditional validation rule – Example

### Collection Validation

```
RuleForEach(x => x.Children)
    .SetValidator(new ChildValidator());
```

Fig 5.4.f - Collection validation rule – Example

### Custom Validation

```
RuleFor(x => x.Name)
    .Must(name => name.StartsWith("A"))
    .WithMessage("Name must start with 'A'");
```

Fig 5.4.g - Custom validation rule – Example

### Custom message

```
RuleFor(x => x.Age)
    .GreaterThan(18)
    .WithMessage("You must be over 18 years old.");
```

Fig 5.4.i - Custom message rule – Example

### Manual validation

```
var validator = new PersonValidator();
var result = validator.Validate(person);

if (!result.IsValid)
{
    foreach (var error in result.Errors)
    {
        Console.WriteLine(error.ErrorMessage);
    }
}
```

Fig 5.4.j - Manual validation – Example

**Custom rule set**

```
RuleSet("Admin", () => {
    RuleFor(x => x.IsAdmin).Equal(true);
});
```

Fig 5.4.k - Custom ruleset – Example

**Options for custom rule set validation**

```
// Validate using "Admin" ruleset
var result = validator.Validate(person, options =>
{
  //option 1: explicit
  options.IncludeRuleSets("Admin");
  //option 2: default + explicit
  options.IncludeRuleSets("default","Admin");
});
```

Fig 5.4.l - Rule set validations options – Example

Choose options default / explicit / (default + explicit) rule set.

**Manual custom rule set validation**

```
var person = new Person { Name = "", Age = 65 };
var validator = new PersonValidator();

// Validate using "Admin" ruleset
var result = validator.Validate(person, options => {
    options.IncludeRuleSets("Admin");
});

if (!result.IsValid)
{
    foreach (var error in result.Errors)
    {
        Console.WriteLine(error.ErrorMessage);
    }
}
```

Fig 5.4.m - Custom rule set validation using “Admin” - Example

You can apply validation in options for **Fig 5.4.l**. In example **Fig 5.4.m**, explicit rule set "Admin" is executed.

"default" – execute rule set with no rule set name.

Ruleset validation helps to customize configurations.

Note: Refer https://docs.fluentvalidation.net/en/latest/rulesets.html for documentation.

## 5.5 Outside-In Test Approach for Developers

Developers are guilty for being attached for code written. It is great ability to test code as manual tester do for black box test

**1. Start from the User Perspective**

Manual testers begin with user stories or requirements, not code internals.

Developers should ask: "What does the user expect this feature to do?" before writing a single assertion.

**2. Define External Behaviors**

Document expected inputs, outputs, and side effects.

Example: "When a user enters an invalid email, the system must show an error message."

This becomes the unit test case, not an improvised check.

**3. Translate into Test Scenarios**

Manual testers typically write structured cases. Developers can mirror this format:

| Test Case ID | Scenario | Input | Expected Output | Notes |
|---|---|---|---|---|
| TC001 | Valid email login | user@example.com | Login success | Normal path |
| TC002 | Invalid email format | user@.com | Error message | Edge case |
| TC003 | Empty email field | `` | Prompt for input | Negative case |

**4. Apply Layered Thinking**

Happy path: Does the feature work as intended?

Edge cases: What happens with unusual or extreme inputs?

Error handling: How does the system respond to invalid actions?

Integration points: Does it interact correctly with other modules?

**5. Document Before Coding**

Manual testers write test cases first; developers should document unit test intent before implementation.

This prevents “logic drift” where tests reflect developer assumptions instead of user expectations.

**6. Traceability**

Each unit test should map back to a documented external case.

This creates a clear audit trail: requirement → test case → unit test → result.

**Example Developer Unit Test Documentation Template**

*Feature: User Login*

*Requirement: System must validate email format before login.*

*Test Case ID: TC002*
*Scenario: Invalid email format*
*Input: "user@.com"*
*Expected Output: Error message "Invalid email address"*
*Unit Test Name: test_invalid_email_format*
*Notes: Derived from external test case TC002*

# References


Beck, K. (2003). Test-Driven Development: By Example. Addison-Wesley.

Martin, R. C. (2002). Agile Software Development: Principles, Patterns, and Practices. Prentice Hall.

Martin, R. C. (2008). Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall.

Martin, R. C. (2017). Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall.

Meyer, B. (1988). Object-Oriented Software Construction. Prentice Hall.

Liskov, B., & Wing, J. M. (1994). A Behavioral Notion of Subtyping. ACM Transactions on Programming Languages and Systems, 16(6), 1811–1841.

Riel, A. J. (1996). Object-Oriented Design Heuristics. Addison-Wesley.

Holland, I. (1987). Law of Demeter: A Structural Guideline for Designing Object-Oriented Systems. Northeastern University, Technical Report.

Meszaros, G. (2007). xUnit Test Patterns: Refactoring Test Code. Addison-Wesley.

Fowler, M. (2007). Mocks Aren't Stubs. martinfowler.com.