



코드 기능 단위테스트 교육

C# 기반 테스트 자동화 및 TDD 실무 역량 강화

교육명
SDC SW 직군 대상(코딩단위테스트) 교육

교육 대상
SDC SW 직군

교육 기간 및 내용
총 3일(24H), C# 기반 단위테스트/TDD/Moq/CI-CD

교육 목표 단위 테스트의 이해와 TDD 기반 개발, 테스트 자동화를 통한 실무형 테스트 역량 강화

목차

1 단위테스트 기초

개념, FIRST 원칙, TDD 개요

2 C# 테스트 환경

VS 설정, NUnit/xUnit 프레임워크

3 기초 실습

Calculator 클래스, TestCase 활용

4 의존성 처리(Moq)

테스트 더블, 외부 의존성 격리

5 TDD 실전

Red-Green-Refactor 사이클

6 고급 기법

예외 처리, 비동기, 의존성 제어

7 코드 커버리지

ReportGenerator, 품질 분석

8 테스트 자동화/CI-CD

GitHub Actions, 파이프라인

9 종합 프로젝트

TDD 미니 프로젝트

10 부록

FAQ, 용어집, 참고자료

교육 개요 및 운영



교육 대상

SDC SW 직군



교육 시간

3일 × 8시간 (총 24시간)



교육 방식

강의 40% + 실습 60%

실무 중심의 핸즈온 워크샵



교육 도구

Visual Studio, .NET, NUnit/xUnit
Moq, GitHub Actions, ReportGenerator



교육 목표

단위테스트와 TDD 기반 개발 역량 확보를 통해 소프트웨어 품질을 높이고, 지속적 통합 및 배포(CI/CD) 환경에서의 테스트 자동화 구축 능력을 강화합니다.

커리큘럼 로드맵

단위테스트 기초부터 CI/CD 자동화까지 체계적인 교육 과정



Step 1

단위테스트 기초

FIRST 원칙, TDD 개념, 필요성



Step 2

C# 테스트 환경

NUnit/xUnit, VS 설정, Assert



Step 3

기초 실습

Calculator 테스트, TestCase



Step 5

TDD 실전

Red-Green-Refactor, 실습



Step 4

의존성 처리(Moq)

테스트 더블, 외부 의존성 격리



Step 6

고급 기법

예외, 비동기, 경계값 테스트



Step 9

종합 프로젝트

팀별 TDD 프로젝트, 코드 리뷰



실무 이관 중심 설계: 기초부터 실전 역량을 단계적으로 강화

단위테스트란?

정의

단위테스트는 소프트웨어의 최소 단위(함수/메서드)를 격리된 환경에서 검증하는 테스트입니다. 코드가 의도대로 작동하는지 확인하는 자동화된 테스트 코드를 작성해 반복 실행합니다.



빠름

외부 의존성 없이 빠르게 실행되며, 각 테스트는 독립적으로 작동합니다.



안전망

리팩토링 시 회귀 문제를 조기에 발견하고, 코드 변경의 영향을 즉시 확인합니다.



설계 개선

테스트하기 쉬운 코드를 작성하도록 유도해 결합도는 낮추고 응집도는 높입니다.



예시: Calculator 클래스 테스트

```
[Test]
public void Add_TwoNumbers_ReturnsSum()
{
    var calc = new Calculator();
    Assert.AreEqual(5, calc.Add(2, 3));
}
```

단위테스트의 비즈니스 가치

\$ 버그 비용 절감

초기 단계에서 결함 발견 시 수정 비용은 프로덕션 단계의 약 1/100 수준
고객 신뢰도 손실과 관련된 간접 비용 방지

🕒 리팩토링 속도 향상

코드 변경에 대한 회귀 테스트 자동화
안전한 리팩토링으로 기술 부채 감소와 품질 향상

🛡️ 배포 안정성 강화

프로덕션 환경의 예상치 못한 장애 감소
장애 복구 시간(MTTR) 단축 및 가용성 증가

CI/CD 품질 게이트

지속적 통합 환경의 핵심 품질 지표로 활용
자동화된 릴리스 파이프라인의 기반 구축

↳ ROI(투자수익률)

단위테스트에 투자하는 초기 시간 대비, 디버깅 및 유지보수 시간 감소, 고객 만족도 증가, 신뢰성 있는 릴리스 주기 단축 등 장기적 이득이 큽니다.

FIRST 원칙 개요

좋은 단위 테스트의 5가지 원칙

FIRST는 효과적인 단위 테스트를 위한 핵심 원칙으로, Fast(빠르게), Independent(독립적으로), Repeatable(반복 가능하게), Self-Validating(자기 검증), Timely(적시에)의 약자입니다. 이 원칙들은 테스트 코드의 품질을 평가하는 기준이자 가이드라인입니다.



명확한 기준

일관된 품질의 테스트를 작성하기 위한 명확한 지침을 제공합니다.



테스트 품질 향상

유지보수가 쉽고 신뢰할 수 있는 고품질 테스트를 작성할 수 있습니다.



CI/CD 지원

자동화된 빌드 및 배포 프로세스에 적합한 테스트를 작성할 수 있습니다.



FIRST 원칙 적용 예시

```
// FIRST 원칙을 따른 테스트  
  
[Test]  
public void IsValidEmail_WithValidFormat_ReturnsTrue()  
{  
    var validator = new EmailValidator();  
    var result = validator.IsValidEmail("user@example.com");  
    Assert.IsTrue(result);  
}
```

FIRST - Fast



밀리초~초 단위 실행

단위 테스트는 가능한 빠르게 실행되어야 합니다

이상적으로 각 테스트는 밀리초 단위로, 전체 테스트 묶음은 수 초 내로 완료



느린 I/O 제거

DB, 파일 시스템, 네트워크 호출 배제

외부 서비스나 API 의존성은 모의 객체(Mock)로 대체



테스트 묶음 병렬 실행

독립적인 테스트 설계로 병렬 실행 가능

CI/CD 파이프라인에서 빠른 피드백 루프 확보



빠른 테스트의 중요성

개발자 피드백 루프 단축으로 생산성 향상

실행이 느린 테스트는 개발자들이 실행을 회피하게 됨



Fast 원칙 실천 방법

테스트 실행 시간이 1초 이상 소요된다면, 테스트 코드를 검토하여 외부 의존성 격리, 무거운 로직 분리, 테스트 범위 축소 등을 통해 실행 속도를 개선해야 합니다. 느린 테스트는 지속적 통합(CI) 및 TDD의 효율성을 저해합니다.

FIRST - Independent & Repeatable



테스트 간 순서 독립성

테스트는 다른 테스트와 독립적이어야 합니다.
테스트 실행 순서에 의존하면 안 됩니다.



상태 독립성

테스트 간 공유 상태를 피해야 합니다.
각 테스트는 필요한 상태를 스스로 설정합니다.



환경 독립성

어떤 환경에서도 동일한 결과를 내야 합니다.
개발, 테스트, CI 환경 등에서 동일하게 동작해야 합니다.



고정 입력 사용

시간, 랜덤 값, 환경변수 등을 고정합니다.
테스트 결과의 일관성과 예측 가능성을 확보합니다.



베스트 프랙티스

SetUp/TearDown 메서드를 활용해 테스트마다 깨끗한 상태를 유지하고, 시간/랜덤 의존성은 인터페이스로 추상화하여 테스트에서 제어할 수 있도록 만듭니다. 글로벌 상태를 피하고, 데이터베이스 연결이나 외부 API 호출 같은 외부 의존성은 경리합니다.

FIRST - Self-Validating & Timely 원칙



성공/실패 이분 결과

테스트는 스스로 성공 또는 실패를 명확히 판단해야 합니다.
수동 해석이나 로그 확인 과정이 필요하면 안 됩니다.



사전조건/사후조건 명확화

테스트에 필요한 모든 데이터와 예상 결과를 명시합니다.
외부 확인 없이도 결과 검증이 완전해야 합니다.



코드 작성 전 또는 직후에 작성

TDD에서는 코드 작성 전에, 테스트 후기 방식에서는 기능 구현 직후에 테스트를 작성합니다.
시간이 지날수록 테스트 작성의 가치와 정확성이 떨어집니다.



즉각적인 피드백 제공

테스트는 빠르게 실행되어 즉각적인 피드백을 제공해야 합니다.
코드 변경 후 바로 테스트하여 문제를 조기에 발견합니다.



FIRST 원칙 적용 팁

Self-Validating과 Timely 원칙은 서로 연관성이 높습니다. 테스트를 적시에 작성할 때 검증 기준이 명확하며, 자가 검증이 확실한 테스트는 지속적인 통합(CI) 환경에서 더 큰 가치를 발휘합니다.

TDD 개요

테스트 주도 개발(Test-Driven Development)

테스트 코드를 먼저 작성한 후 실제 코드를 구현하는 개발 방법론입니다. Red-Green-Refactor 사이클을 통해 요구사항을 테스트로 구체화하고, 점진적인 개발과 리팩토링을 반복하며 코드 품질을 향상시킵니다.

1. Red

실패하는 테스트 작성

구현 전에 요구사항을 검증하는 테스트를 먼저 작성합니다.

2. Green

테스트 통과 코드 작성

테스트를 통과하는 가장 간단한 코드를 작성합니다.

3. Refactor

코드 개선 및 리팩토링

테스트를 통과하는 상태를 유지하며 코드 품질을 향상시킵니다.

TDD의 핵심 가치

설계 중심 개발

테스트를 먼저 작성함으로써 더 나은 설계와 API를 자연스럽게 도출합니다.

회귀 방지 안전망

모든 기능에 테스트가 있어 변경 시 기존 기능이 손상되지 않았는지 확인합니다.

실전 예시

문자열 계산기 TDD 개발:

```
// 1. Red: 실패하는 테스트
[Test]
public void Add_EmptyString_ReturnsZero()
{
    var calc = new StringCalculator();
    Assert.AreEqual(0, calc.Add(""));
}

// 2. Green: 최소 구현
public int Add(string s) { return 0; }
```

TDD 안티패턴



테스트 없는 대규모 구현

많은 코드를 한 번에 작성하고 나중에 테스트를 추가하는 방식
TDD의 짧은 피드백 사이클을 무시하게 됨



과도한 통합테스트 의존

단위테스트를 건너뛰고 통합테스트에만 의존
실행 속도 저하 및 테스트 실패 원인 파악 어려움



불안정 테스트(Flaky) 방치

간헐적으로 실패하는 테스트를 무시
테스트 신뢰성 저하 및 빌드 실패 증가



테스트 후 리팩토링 없음

Red-Green 단계 후 Refactor 단계 생략
코드 품질 저하 및 중복 발생



TDD 성공의 핵심

위 안티패턴을 피하고, 작은 단위로 테스트-구현-리팩토링 사이클을 충실히 반복하는 것이 TDD 성공의 핵심입니다. 테스트는 코드 품질의 안전망이자 설계 개선의 도구입니다.



Part 2

C# 단위테스트 환경 구축

Visual Studio 테스트 설정, NUnit/xUnit 프레임워크,
테스트 프로젝트 구조 및 기본 설정

개발 환경 요구사항

C# 단위테스트 실습을 위한 필수 개발 환경 항목입니다.

필수 설치 항목

.NET 8 SDK

최신 .NET SDK 설치 필요

Visual Studio 2022

Community 이상, .NET 데스크톱 개발 워크로드

NuGet 패키지 관리자

테스트 프레임워크/도구 설치용

권장 설치 항목

Fine Code Coverage

VS 확장, 커버리지 시각화 도구

Git 버전 관리

GitHub Actions 연동용

확인 방법

`dotnet --version`으로 SDK 버전 확인. 교육 당일 노트북 지참 필수

Visual Studio 테스트 설정

Visual Studio에서 단위 테스트 환경을 구축하는 단계별 가이드입니다.

1 Test Explorer 활성화

메뉴: [보기] > [테스트 탐색기] 또는 Ctrl+E,T

2 테스트 프로젝트 생성

솔루션 우클릭 > [추가] > [새 프로젝트] > [NUnit/xUnit 테스트 프로젝트]

명명 규칙: 프로젝트명.Tests

3 NuGet 패키지

```
dotnet add package Microsoft.NET.Test.Sdk NUnit NUnit3TestAdapter Coverlet.collector
```

4 프로젝트 참조 추가

테스트 프로젝트 > [종속성] > [프로젝트 참조 추가]



팁

Test Explorer에서 태그/이름으로 검색하거나 상태(통과/실패)로 필터링하여 특정 테스트를 찾을 수 있습니다.

.NET Test SDK/패키지



Microsoft.NET.Test.Sdk

.NET 테스트 기본 SDK, VS 및 CLI 테스트 실행 지원



NUnit /xUnit

테스트 프레임워크, 어트리뷰트, Assertion 제공



테스트 어댑터

VS Test Explorer 연동 (NUnit3TestAdapter,
xunit.runner.visualstudio)



Coverlet.collector

테스트 실행 시 코드 커버리지 데이터 자동 수집



패키지 관리

프로젝트 템플릿: dotnet new nunit 또는 dotnet new xunit

전체 솔루션 복원: dotnet restore

NUnit 소개와 설치

개요

NUnit은 .NET 플랫폼을 위한 오픈소스 단위 테스트 프레임워크로, 성숙한 생태계와 풍부한 어트리뷰트 기반 테스트 기능을 제공합니다.



풍부한 어트리뷰트

[TestCase], [SetUp], [TearDown], [Category] 등 테스트 구성과 실행을 유연하게 제어합니다.



확장성과 통합성

Visual Studio, Rider와 쉽게 통합되며, CI/CD 파이프라인과 연동이 용이합니다.



테스트 구조화

복잡한 테스트 시나리오를 TestFixture와 매개변수화된 테스트로 효율적 구성 가능합니다.



설치 방법

NuGet 패키지를 통해 설치:

```
// NUnit 프레임워크 및 어댑터 설치  
dotnet add package NUnit  
dotnet add package NUnit3TestAdapter  
dotnet add package Microsoft.NET.Test.Sdk
```

xUnit 소개와 설치

개요

xUnit은 .NET Core 및 .NET 5+ 환경에 최적화된 현대적인 단위 테스트 프레임워크입니다. 오픈소스로 개발되었으며, 심플하고 확장 가능한 디자인을 갖고 있어 마이크로소프트에서도 권장하는 테스트 프레임워크입니다.



현대적 특징

생성자 기반 테스트 픽스처, [Fact]/[Theory] 어트리뷰트 방식의 간결한 구문 제공



확장성

DI 지원, 병렬 테스트 실행, 데이터 주도 테스트 (DataAttribute) 제공



간편한 설치

NuGet 패키지로 빠른 설치, 최소한의 설정으로 테스트 시작 가능



설치 방법

```
dotnet add package xunit  
dotnet add package xunit.runner.visualstudio
```

또는 새 프로젝트 생성:

```
dotnet new xunit -n YourTestProject
```

프레임워크 비교

	NUnit	xUnit	MSTest
기본 개념	JUnit 기반, 오픈소스 v1부터 역사가 깊	NUnit 창시자가 개발한 최신형 관례보다 설정 중시	Microsoft 공식 프레임워크 VS와 긴밀한 통합
주요 속성	[Test], [TestCase] [SetUp], [TearDown]	[Fact], [Theory] 생성자 /IDisposable	[TestMethod], [DataRow] [TestInitialize]
데이터 기반 / 병렬화	[TestCase], [TestCaseSource] 병렬: 설정으로 지원 (Assembly 수준)	[Theory] + [InlineData] 병렬: 기본 내장 지원	[DataTestMethod] + [DataRow] 병렬: VS2019+ 설정 필요
장단점	+ 풍부한 기능과 문서화 - 레거시 API 일부 존재	+ 현대적 설계와 심플함 - 상대적 학습 곡선	+ VS 통합 및 쉬운 시작 - 일부 기능 제약 존재
추천 사용처	레거시 프로젝트, 풍부한 기능 필요	신규 프로젝트, .NET Core 환경	MS 중심 환경, 간단한 테스트

선택 가이드

신규 애플리케이션은 xUnit(현대적, .NET Core 호환), 레거시/다양한 기능은 NUnit, Microsoft 생태계 내 간단한 테스트는 MSTest가 적합합니다.

첫 테스트 프로젝트 생성

dotnet CLI를 사용하여 C# 단위 테스트 프로젝트를 생성하는 기본 단계입니다.

1 솔루션 및 프로젝트 생성

```
mkdir UnitTestDemo && cd UnitTestDemo  
dotnet new sln  
dotnet new classlib -n Demo
```

2 테스트 프로젝트 생성

```
dotnet new nunit -n Demo.Tests # 또는 xunit  
dotnet sln add Demo Demo.Tests
```

3 프로젝트 참조 연결

```
dotnet add Demo.Tests reference Demo
```

이제 Demo.Tests 프로젝트에서 Demo 프로젝트의 클래스를 테스트할 수 있습니다.

팁: 프로젝트 구조

프로젝트는 src와 tests 폴더로 분리하는 구조가 권장됩니다. 테스트 프로젝트 이름은 테스트 대상 프로젝트 이름 뒤에 .Tests를 붙이는 것이 관례입니다.

Test Explorer 활용

Visual Studio의 Test Explorer를 통한 효율적인 테스트 관리 및 실행



Step 1

테스트 발견

빌드 시 테스트 어셈블리에서 테스트 메서드 자동 검색



Step 2

필터링/구성

프로젝트, 네임스페이스, 클래스별 그룹화 및 필터



Step 3

테스트 실행

전체/선택/실패 테스트 실행, 병렬 실행 지원



Step 4

결과 분석

성공/실패 결과, 오류 메시지 및 스택 추적 분석



Step 5

디버깅

디버깅 모드 실행, 중단점 설정 및 변수 검사



Step 6

반복 실행

코드 수정 시 Live Unit Testing 또는 수동 재실행



단축키 활용

Ctrl+R, T: 모든 테스트 실행 | Ctrl+R, A: 디버깅 | Ctrl+R, F: 실패 테스트



테스트 탐색기 패널

좌측 트리 뷰 테스트 계층구조



테스트 결과 패널

결과, 출력, 예외 정보 확인



도구 모음

실행, 중지, 디버그 등 주요 기능

Assert 핵심 메서드

값 비교 Assert

Assert.AreEqual(expected, actual)

Assert.AreNotEqual(notExpected, actual)

조건 Assert

Assert.IsTrue(condition)

Assert.IsFalse(condition)

Null 검증

Assert.IsNull(object)

Assert.IsNotNull(object)

예외 검증

Assert.Throws<Exception>(() => method())

Assert.DoesNotThrow(() => method())

컬렉션 및 특수 비교

컬렉션: CollectionAssert.Contains(collection, item)

부동소수점: Assert.AreEqual(expected, actual, delta)

가독성: AreEqual(..., "실패 시 메시지")

데이터 주도 테스트

데이터 주도 테스트란?

다양한 입력 값으로 동일한 테스트 로직을 반복 실행하는 방법입니다. 테스트 코드와 데이터를 분리해 코드 중복을 줄이고 효율적으로 테스트합니다.



NUnit의 TestCase

여러 매개변수 조합으로 테스트를 실행합니다. 각 TestCase는 테스트 메서드에 값을 전달합니다.



xUnit의 Theory

파라미터화된 테스트용 속성으로, [Fact]가 단일 테스트라면 [Theory]는 여러 데이터로 테스트합니다.



데이터 소스

InlineData, MemberData, ClassData 등 다양한 속성을 통해 테스트 데이터를 제공합니다.



코드 예제

NUnit 예제:

```
[TestCase(1, 1, 2)]
[TestCase(2, 3, 5)]
public void Add_Returns_Sum(
    int a, int b, int expected)
{
    Assert.AreEqual(expected,
        calculator.Add(a, b));
}
```

xUnit 예제:

```
[[Theory]
[InlineData(1, 1, 2)]
[InlineData(2, 3, 5)]
public void Add_Returns_Sum(
    int a, int b, int expected)
{
    Assert.Equal(expected,
        calculator.Add(a, b));
}
```

테스트 명명과 조직



AAA 패턴

Arrange (준비) - 테스트에 필요한 객체 준비
Act (실행) - 테스트할 기능 수행
Assert (검증) - 결과 검증



메서드 명명 규칙

UnitOfWork_Condition_Expected
예: Add_NegativeNumbers_ReturnsSum
의도를 명확히 표현하는 이름 사용



테스트 조직화

클래스별/기능별 테스트 그룹화
한 테스트 클래스는 한 가지 주제에 집중
관련 테스트 사이의 중복 최소화



단일 책임 테스트

한 테스트는 한 가지만 검증
여러 Assert를 사용하더라도 검증 대상은 하나
테스트 실패 시 원인 파악이 쉬움



베스트 프랙티스

좋은 테스트 명명과 조직은 코드 문서화 효과가 있으며, 새로운 팀원이 코드를 이해하는 데 도움을 줍니다. 테스트 코드는 프로덕션 코드만큼 중요하므로 동일한 품질 기준을 적용하세요.



Part 3

단위테스트 기초 실습

Calculator 예제를 통한 AAA 패턴 실습,
TDD 기본 사이클 및 테스트 픽스처 구성 연습

요구사항: Calculator



계산기 클래스 실습

기본적인 수학 연산을 수행하는 Calculator 클래스를 TDD 방식으로 개발하고 테스트합니다.

AAA 패턴을 적용하여 단위 테스트의 기본 원칙을 실제로 적용해 봅니다.



기본 테스트 작성법과 경계값 테스트, 예외 처리 테스트 방법을 학습합니다.

기능 요구사항

기본 연산

두 정수의 덧셈(Add) 연산 구현

두 정수의 뺄셈(Subtract) 연산 구현

두 정수의 곱셈(Multiply) 연산 구현

두 정수의 나눗셈(Divide) 연산 구현

예외 처리

0으로 나눌 때 DivideByZeroException 발생

정수 범위 초과 시 예외 처리

경계값 테스트

int 최대값/최소값 연산 검증

음수 계산 정확성 검증

0 포함 연산 정상 동작 검증

Red: 실패하는 테스트 작성

RED

구현하기 전에 실패하는 테스트를 먼저 작성합니다. 이는 TDD 사이클의 첫 번째 단계입니다.

📝 CalculatorTests.cs

실패

```
using NUnit.Framework; using System;
namespace Demo.Tests {
    [TestFixture]
    public class CalculatorTests {
        [Test]
        public void Add_TwoNumbers_ReturnsSum() {
            // Arrange
            var calculator = new Calculator();
            int a = 2, b = 3, expected = 5;
            // Act
            int result = calculator.Add(a, b);

            // Assert
            Assert.AreEqual(expected, result);
        }
        [Test] public void Divide_ByZero_ThrowsException()
        {
            Assert.Throws<DivideByZeroException>(() => new Calculator().Divide(10, 0));
        }
    }
}
```

! 이 테스트가 실패하는 이유

Calculator 클래스와 메서드가 아직 구현되지 않았기 때문입니다. TDD에서는 이 '실패'가 다음 단계(Green)로 진행하는 신호입니다.

Green: 최소 구현

GREEN

테스트를 통과시키기 위한 최소한의 코드를 구현합니다. TDD 사이클의 두 번째 단계입니다.

</> / ⚡️

성공

```
using System;
namespace Demo {
    public class Calculator {
        public int Add(int a, int b) {
            return a + b;
        }
        public int Divide(int a, int b) {
            if (b == 0) {
                throw new DivideByZeroException();
            }
            return a / b;
        }
        // Subtract, Multiply 등 다른 메서드도 유사한 방식으로 구현 가능
    }
}
```

✓ 최소한의 구현으로 테스트 통과

이 구현은 앞서 작성한 테스트를 통과시키기 위한 가장 간단한 코드입니다. Add 메서드는 두 수를 더하고, Divide 메서드는 0으로 나누기를 방지하는 예외 처리를 포함합니다. TDD에서는 이 단계에서 과도한 최적화를 하지 않습니다.

Refactor: 중복 제거



중복 로직 메서드 추출

반복되는 코드 블록을 별도의 메서드로 추출
공통 기능을 재사용하여 코드량 감소



매직넘버 상수화

하드코딩된 숫자를 의미 있는 상수로 변환
코드의 가독성과 유지보수성 향상



의도를 드러내는 네이밍

메서드와 변수명을 기능과 목적이 명확하게 변경
주석 없이도 이해 가능한 코드 작성



불필요한 코드 제거

사용되지 않는 코드 제거
단일 책임 원칙에 맞게 코드 정리



리팩토링 효과

TDD 사이클에서 Green 단계 이후 Refactor 단계는 동작을 변경하지 않으면서 코드의 가독성, 유지보수성, 확장성을 높이는 과정입니다. 테스트는 리팩토링 중 코드가 여전히 정상 동작함을 보장해주는 안전망 역할을 합니다.

경계값/예외 테스트



경계값 테스트

Min/Max 입력 검증
경계 직전, 경계, 경계 직후 값
예: 0, 1, -1, int.MaxValue



예외 처리 테스트

Assert.Throws<예외타입>() 활용
올바른 예외 메시지 검증
특정 조건 예외 검증



부동소수점 정밀도 처리

델타값을 이용한 근사치 비교
AreEqual(expected, actual, delta)
IEEE 754 표준의 한계 인지



테스트 전략

동등 분할 클래스 테스트 결합
DoesNotThrow 예외 미발생 검증
체계적인 경계값 설정



테스트 자동화 팁

파라미터화된 테스트로 다양한 경계값 검증을 자동화하세요. NUnit [TestCase], xUnit [Theory][InlineData]를 활용하면 코드 중복을 줄이고 가독성을 높일 수 있습니다.

Fixture/Setup/TearDown

테스트 픽스처란?

테스트 픽스처는 테스트 실행에 필요한 사전 조건과 환경을 의미합니다. 테스트 대상 객체(SUT), 의존성, 테스트 데이터를 포함하며, Setup/TearDown 메서드로 반복적인 준비 작업을 캡슐화합니다.



NUnit

[SetUp]: 각 테스트 전 호출 [TearDown]: 각 테스트 후 호출



xUnit

생성자: 테스트 전 Setup 역할 IDisposable.Dispose():
TearDown 역할



수준별

[OneTimeSetUp]: 클래스 전 1회 [ClassInitialize]: 클래스 수준

</> NUnit 픽스처 예제

```
[TestFixture]
public class CustomerServiceTests {
    private CustomerService _service;
    private Mock<ICustomerRepository> _mockRepo;
    [SetUp]
    public void Setup() {
        _mockRepo = new Mock<ICustomerRepository>();
        _service = new CustomerService(_mockRepo.Object);
    }
}
```

테스트 냄새와 개선



과도한 Mocking

실제 객체보다 모의 객체가 많으면 테스트가 구현에 과도하게 결합
→ 필수 의존성만 Mock, 나머지는 실제 객체 사용



다중 Assert 남발

하나의 테스트에서 여러 기능을 검증하면 실패 원인 파악 어려움
→ 단일 개념만 검증하는 작은 테스트 작성



비결정성 방지

시간, 날짜, 랜덤값에 의존하여 실행할 때마다 결과가 달라짐
→ 시계/랜덤 주입으로 제어 가능하게 설계



불명확한 의도

테스트 코드만 보고는 무엇을 검증하는지 이해하기 어려움
→ Given-When-Then 구조로 명확하게 표현



테스트 코드 리팩토링

테스트 코드도 프로덕션 코드만큼 정기적인 리팩토링이 필요합니다. 가독성과 유지보수성을 높이기 위해 중복을 제거하고, 테스트 헬퍼와 유틸리티 메서드를 적극 활용하세요. 단, 이로 인해 테스트의 명확성이 저하되지 않도록 주의하세요.

실습 체크리스트

테스트 작성 후 아래 항목을 체크하여 테스트 품질을 점검하세요.

AAA 패턴 준수

구조 명확성

Arrange-Act-Assert 구조가 명확하게 구분되어 있나요? Act는 단일 메서드 호출인가요?

Assert 명확성

검증문이 명확한 기대값과 실제값을 비교하고 있나요?

단일 책임 원칙

테스트 목적 단일화

각 테스트는 하나의 기능 또는 동작만 검증하고 있나요?

명확한 테스트 이름

테스트 메서드 이름이 목적을 명확히 설명하고 있나요?

경계값 및 예외 처리

경계값 테스트

최소값, 최대값, 경계 직전/직후 값을 테스트하나요?

예외 케이스 및 특수값

예외 발생 상황, null, 빈 문자열 등의 특수 케이스를 테스트하나요?

코드 품질 검사

테스트 간 중복된 코드가 제거되고 공통 세팅 코드가 적절히 활용되나요?

가독성

테스트 코드가 다른 개발자도 쉽게 이해할 수 있도록 작성되었나요?

팀 리뷰

팀원과 테스트 코드를 상호 리뷰하고 이 체크리스트를 활용하세요.

요약 및 퀴즈(기초)

학습 내용 요약

- ✓ FIRST 원칙: Fast, Independent, Repeatable, Self-Validating, Timely
- ✓ AAA 패턴: Arrange(준비), Act(실행), Assert(검증)
- ✓ 테스트 메서드 명명 규칙: 테스트대상_조건_기대결과

확인 퀴즈

1

FIRST 원칙에서 S는 무엇을 의미하나요?



Self-Validating (자가 검증)



Specific (구체적)



Simple (단순)

학습 포인트

다음 장에서는 이러한 단위테스트 개념들을 C# 코드로 적용해보겠습니다.



Part 4

외부 의존성 처리(Moq)

테스트 더블 개념, Moq 프레임워크 활용,
API 및 데이터베이스 의존성 격리 테스트 전략

테스트 더블 개념

테스트 더블이란?

테스트 더블은 단위 테스트에서 외부 의존성을 대체하는 객체로, 테스트의 격리성과 속도를 높여줍니다. 영화의 스텐트 더블처럼 실제 의존성을 대신해 테스트 환경을 단순화 합니다.



Dummy

단순 파라미터 채우기용 객체, 실제로는 사용되지 않음



Stub

미리 준비된 답변만 제공하는 객체, 일관된 결과 반환



Fake

실제 구현은 있지만 단순화된 구현체 (예: 인메모리 DB)



Spy

실제 객체를 감싸 호출을 기록, 원래 구현도 사용 가능



Mock

기대 호출과 결과를 미리 프로그래밍, 행동 검증용

Mock 객체 사용 예시 (Moq)

```
// Arrange
var mockRepo = new Mock<IUserRepository>();
mockRepo.Setup(r => r.GetById(1)).Returns(new User { Id = 1, Name = "홍길동" });
var service = new UserService(mockRepo.Object);

// Act
var result = service.GetUser(1);

// Assert
Assert.AreEqual("홍길동", result.Name);
mockRepo.Verify(r => r.GetById(1), Times.Once);
```

주요 라이브러리: Moq, NSubstitute, FakeltEasy 등을 통해 C#에서 테스트 더블을 쉽게 생성할 수 있습니다.

의존성 역전과 인터페이스



DI (의존성 주입)

컴포넌트에 필요한 의존성을 외부에서 주입
구체적인 구현보다 추상화(인터페이스)에 의존



IoC (제어의 역전)

객체 생성/관리 책임을 프레임워크에 위임
의존성 해결을 외부 컨테이너가 담당



인터페이스 설계

작고 명확한 역할 중심 인터페이스 설계
단일 책임 원칙(SRP)을 준수한 분리



생성자 주입 권장

필수 의존성은 생성자를 통해 주입
테스트 시 목(Mock) 객체로 쉽게 교체 가능



테스트 가능한 설계

의존성 역전 원칙(DIP)은 테스트 가능한 코드의 핵심입니다. 고수준 모듈이 저수준 모듈에 의존하지 않고, 둘 다 추상화(인터페이스)에 의존함으로써 쉽게 테스트 대역(Test Double)으로 교체할 수 있는 구조를 만듭니다.

Moq 기본 사용법

Moq

목 객체를 생성하고 Setup 메서드를 통해 의존성 동작을 정의하는 기본 사용법입니다.

</> UserServiceTests.cs

성공

```
using Moq; using NUnit.Framework;

[TestFixture] public class UserServiceTests {
    [Test] public void GetUser_UserIdExists_ReturnsUser() {
        // Arrange
        var userRepository = new Mock<IUserRepository>();
        var userId = 1; var expectedUser = new User { Id = userId, Name = "홍길동" };
        userRepository.Setup(repo => repo.GetUser(userId)).Returns(expectedUser);
        var userService = new UserService(userRepository.Object);
        // Act
        var result = userService.GetUserById(userId);
        // Assert
        Assert.AreEqual(expectedUser, result);
    }
}
```



Moq 핵심 사용법

Mock<T>: 인터페이스/추상 클래스 목 생성 Setup(): 메서드 호출 동작 정의

Returns()/Throws(): 반환/예외 설정 Object: 실제 사용할 목 인스턴스 접근

Moq Setup/Verify

Moq Setup으로 동작 정의, Verify로 호출 검증

</> UserServiceTests.cs

핵심 예제

```
var mockRepo = new Mock<IUserRepository>(); // Setup: 메서드 동작 정의
mockRepo.Setup(r => r.GetById(1)).Returns(new User(1, "홍길동"));
var service = new UserService(mockRepo.Object);
var user = service.GetUserById(1); // Verify: 메서드 호출 검증
mockRepo.Verify(r => r.GetById(1), Times.Once());

// 매개변수 매치
mockRepo.Verify(r => r.Save(It.IsAny<User>()), Times.Never());
```

i 핵심 메서드

Setup>Returns

메서드 호출과 반환값 정의
파라미터 매칭 가능

Verify

메서드 호출 검증
Times: Once, Never, Exactly(n)

Moq 예외/콜백

Moq의 예외 발생, 콜백, 순차적 응답 기능을 활용한 테스트 패턴입니다.

</> Moq 고급 기능 예제

핵심 패턴

```
// 1. 예외 발생 (Throws)
mockRepo.Setup(r => r.GetById(999)).Throws<KeyNotFoundException>();

// 2. 콜백 함수 (Callback)
mockRepo.Setup(r => r.Save(It.IsAny<User>()))
.Callback<User>(u => capturedUser = u);

// 3. 순차적 응답 (SetupSequence)
mockRepo.SetupSequence(r => r.GetNext())
.Returns(new User { Id = 1 })
.Returns(new User { Id = 2 })
.Throws(new Exception());
```



Throws

Callback

SetupSequence

Throws: 예외 상황 테스트 | Callback: 파라미터 캡처 및 사용자 정의 액션 | SetupSequence: 순차적 다른 결과 반환

HTTP 클라이언트 모킹



HTTP 통신 테스트

외부 API 호출 코드를 테스트하기 위해 HttpClient 의존성을 격리하고 모킹하는 실습입니다.

 **HttpMessageHandler**를 모킹하여 HTTP 응답을 시뮬레이션합니다. 이 패턴은 외부 서비스에 의존하는 모든 클라이언트 테스트에 활용 가능합니다.

HttpMessageHandler 모킹 실습

모킹 시나리오

- 성공 응답 (HTTP 200 OK)
- 서버 오류 (HTTP 500 Error)
- 타임아웃 (TaskCanceledException)

검증 포인트

- URL, HTTP 메서드, 헤더 검증
- 요청 본문 직렬화 확인
- HTTP 상태 코드별 예외 처리

모킹 코드 예시

```
var handlerMock = new Mock<HttpMessageHandler>();
handlerMock.Protected()
    .Setup<Task<HttpResponseMessage>>("SendAsync", ItExpr.IsAny<HttpRequestMessage>(),
    ItExpr.IsAny<CancellationToken>())
    .ReturnsAsync(new HttpResponseMessage(HttpStatusCode.OK));
```

DB 연동 테스트 전략



리포지토리 인터페이스 분리

데이터 액세스 로직을 인터페이스로 추상화
비즈니스 로직과 데이터 액세스 관심사 분리
테스트 시 대체 구현체로 교체 가능



InMemory 대체

실제 DB 연결 없이 메모리 내 데이터로 테스트
EF Core InMemory Provider 활용
빠른 실행 속도와 격리성 확보



트랜잭션 롤백

실제 DB 사용 시 테스트 후 변경사항 되돌림
TransactionScope로 자동 롤백 구현
테스트 간 데이터 독립성 유지



테스트 데이터 관리

Builder 패턴으로 테스트 데이터 구성
테스트 픽스처 공유 및 재사용
시드 데이터 초기화 자동화



베스트 프랙티스

단위 테스트는 실제 DB에 의존하지 않도록 설계하고, 통합 테스트에서만 제한적으로 실제 DB 연동을 테스트합니다. Docker 컨테이너로 격리된 테스트 DB 환경을 구성하면 CI/CD 파이프라인에서도 안정적인 테스트가 가능합니다.

시간/랜덤/정적 제어



IClock 주입

시간에 의존하는 코드를 테스트하기 위한 방법
고정된 시간 제공 및 시간 흐름 제어 가능

DateTime.Now 대신 IClock.Now 사용



Random 시드 고정

랜덤 값을 테스트 시 항상 동일한 결과로 제어
new Random(seed: 42)와 같이 시드 값 고정
랜덤 인터페이스 주입으로 테스트 가능성 확보



정적 메서드 래핑

테스트가 어려운 정적 메서드를 인스턴스 메서드로 변환
인터페이스를 통한 추상화로 모킹 가능하게 함
예: File.Exists(path) → IFileSystem.FileExists(path)



시스템 시간 가상화

특정 시점이나 날짜 기반 로직 테스트
시간 흐름 시뮬레이션 (오래된 데이터, 미래 날짜)
타임존, 서머타임 관련 로직 안정적 테스트



구현 패턴

비결정적 요소(시간, 랜덤, 정적 메서드)는 테스트 가능성을 저해하는 주요 원인입니다. 인터페이스 주입과 래퍼 클래스 패턴을 활용하면 이러한 요소들을 효과적으로 제어하고 격리된 테스트를 작성할 수 있습니다.

단위 vs 통합 경계

테스트 경계의 이해

단위 테스트는 외부 의존성을 모의(mock)하여 격리된 컴포넌트를 검증하는 반면, 통합 테스트는 실제 의존성과의 상호작용을 포함합니다. 경계를 명확히 하여 테스트 효율성을 높이세요.



경계 정의

명확한 테스트 경계는 중복을 줄이고 테스트 유형별 가치를 극대화합니다.



테스트 피라미드

단위(Unit) > 서비스(Service) > UI(E2E) 순으로 비율을 구성하세요.



균형적 접근

로직은 단위 테스트로, 통합 지점은 통합 테스트로 검증하세요.



테스트 유형 비교:

```
// 단위 테스트: 외부 의존성 모의 처리
var mockRepo = new Mock<IOrderRepository>();
var service = new OrderService(mockRepo.Object);
Assert.AreEqual(expectedTotal, service.CalculateTotal(items));
// 통합 테스트: 실제 의존성 사용
var repo = new OrderRepository(realDbConnection);
var service = new OrderService(repo);
Assert.IsNotNull(repo.GetById(service.PlaceOrder(order)));
```

실습: 서비스 레이어 모킹 (1/2)

주문합계 계산 서비스 모킹 실습 - Moq를 활용한 서비스 레이어 테스트

1

서비스 인터페이스 정의

```
public interface IOrderCalculationService {  
    decimal CalculateTotal(Order order);  
    decimal ApplyDiscount(decimal total, string couponCode);  
}
```

2

테스트할 컨트롤러

```
public class OrderController {  
    private readonly IOrderCalculationService _service;  
    public OrderController(IOrderCalculationService service) { _service = service; }  
    public decimal ProcessOrder(Order order, string coupon) {  
        // 1. 주문 합계 계산  
        var total = _service.CalculateTotal(order);  
  
        // 2. 쿠폰 적용 (있는 경우)  
        if (!string.IsNullOrEmpty(coupon)) {  
            total = _service.ApplyDiscount(total, coupon);  
        }  
        return total;  
    }  
}
```

팁: 인터페이스는 단일 책임으로 작고 명확하게 설계하세요. 모킹 테스트에 최적화됩니다.

실습: 서비스 레이어 모킹 (2/3)

Moq를 활용한 주문합계 계산 서비스 테스트

3

Moq 테스트 코드 구조

Mock 리포지토리 생성 및 주입 구조

```
public class OrderServiceTests
{
    private OrderService _orderService;
    private Mock<IOrderRepository> _mockRepository;

    [SetUp]
    public void Setup()
    {
        _mockRepository = new Mock<IOrderRepository>();
        _orderService = new OrderService(_mockRepository.Object);
    }
    // 다음 페이지에서 테스트 메서드 구현을 확인하세요
}
```

// 다음 페이지에서 테스트 메서드 구현을 확인하세요



팁

Moq를 활용할 때 Setup() 메서드에서는 필요한 목 객체를 초기화하고, 테스트에서는 목 객체의 동작을 설정합니다.

요약/베스트프랙티스(Moq)

Moq 라이브러리를 효과적으로 활용하기 위한 핵심 베스트 프랙티스를 정리했습니다.

인터페이스 중심 설계

구체 클래스보다 인터페이스를 모킹하세요. 테스트 대상 코드가 인터페이스에 의존하도록 설계하면 Moq를 훨씬 효과적으로 활용할 수 있습니다.

최소한의 행위 검증

모든 호출을 검증하는 대신 핵심 동작만 검증하세요. 과도한 Verify는 테스트를 구현 세부사항에 결합시켜 리팩토링을 어렵게 만듭니다.

공개 인터페이스만 모킹

비공개 메서드나 내부 구현 세부사항은 모킹하지 마세요. 테스트는 공개 API에 초점을 맞춰야 합니다. 내부 로직을 검증하려면 설계를 재고하세요.

실제 버그에 가까운 시나리오

현실적인 시나리오를 테스트하세요. 실무에서 발생할 수 있는 문제 상황(타임아웃, 네트워크 오류 등)을 Moq로 재현하여 견고한 코드를 작성하세요.

핵심 요약

모킹은 격리된 테스트 환경을 제공하는 도구일 뿐, 모든 테스트에 필요한 것은 아닙니다. 단순한 로직은 실제 객체로 테스트하는 것이 더 명확할 수 있습니다. Moq를 사용할 때는 테스트 의도가 명확히 드러나도록 작성하고, 테스트 자체가 유지보수의 부담이 되지 않도록 주의하세요.



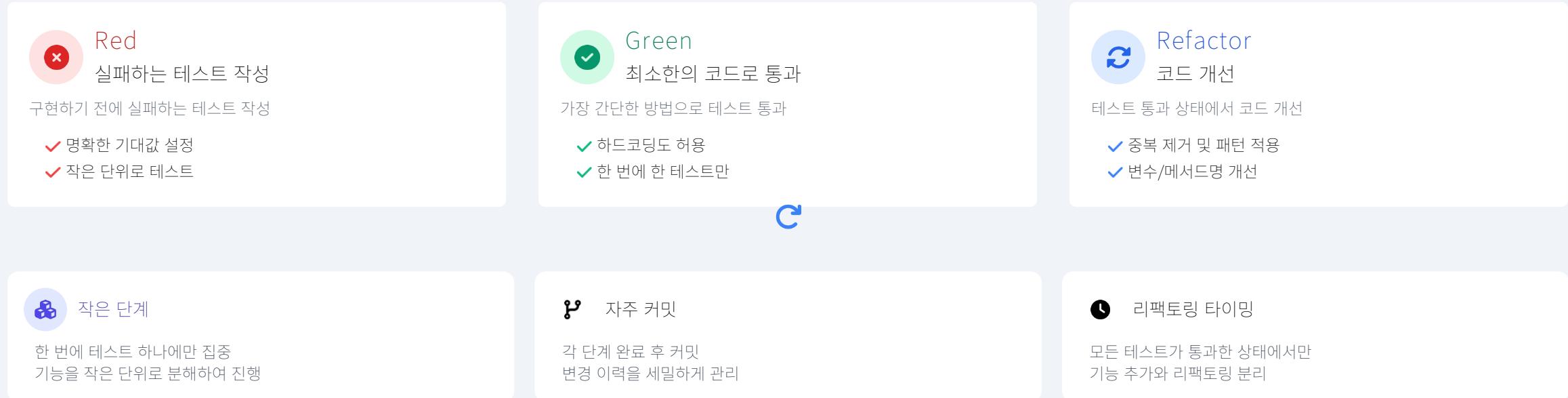
Part 5

TDD 실전

요구사항을 테스트로 구체화하고
실무 프로젝트에 TDD를 적용하는 실전 기법

TDD 사이클 심화

Red-Green-Refactor 사이클을 더 효과적으로 적용하는 방법



💡 TDD는 작은 단계의 반복적인 사이클입니다. 하나의 큰 단계로 건너뛰지 말고 각 단계를 충실히 수행하세요.

예제1: 문자열 계산기 요구사항



문자열 계산기 TDD 실습

문자열로 된 숫자를 입력받아 덧셈 연산을 수행하는 계산기를 TDD 방식으로 개발합니다.

작은 단계로 진행하며 TDD의 핵심 원칙을 체득합니다.

기능 요구사항

기본 동작

빈 문자열을 입력할 경우 0을 반환

숫자 하나를 문자열로 입력할 경우 해당 숫자를 반환 (예: "1" → 1)

쉼표(,)로 구분된 숫자의 합 반환 (예: "1,2,3" → 6)

구분자

개행문자(\n)도 구분자로 사용 가능 (예: "1\n2,3" → 6)

커스텀 구분자 지정: "//[구분자]\n[숫자]" (예: "//;\n1;2" → 3)

예외 처리 및 제한

음수 입력 시 예외 발생: "음수는 허용되지 않습니다: [음수목록]"

1000 초과 숫자는 무시 (예: "2,1001" → 2)

예제1 테스트 목록

문자열 계산기 Add 메서드를 TDD로 구현하기 위한 주요 테스트 케이스입니다.

1 빈 문자열 테스트

빈 문자열이 입력되면 0을 반환해야 합니다.

```
Assert.That(calculator.Add(""), Is.EqualTo(0));
```

2 단일 숫자 테스트

하나의 숫자가 입력되면 해당 숫자를 반환합니다.

```
Assert.That(calculator.Add("1"), Is.EqualTo(1));
```

3 쉼표로 구분된 숫자들 테스트

쉼표로 구분된 숫자들의 합을 반환합니다.

```
Assert.That(calculator.Add("1,2"), Is.EqualTo(3));
```

4 고급 기능 테스트

개행 문자, 커스텀 구분자, 음수 예외 처리를 테스트합니다.

```
// 개행 및 커스텀 구분자  
Assert.That(calculator.Add("1\n2,3"), Is.EqualTo(6));
```

```
// 음수 예외  
Assert.Throws<ArgumentException>(() => calculator.Add("-1,2"));
```

💡 팁: 테스트 순서

가장 간단한 테스트부터 시작하고, 테스트를 통과시킬 수 있는 최소한의 코드만 작성하세요.

예제1 단계별 구현

문자열 계산기 예제를 통한 TDD의 단계별 구현 프로세스



각 단계마다 실패 테스트부터 시작하여 통과시킨 후 개선하는 TDD 사이클을 반복합니다.

코드 기능 단위테스트 교육 | Part 5: TDD 실전

예제1 리팩토링 기법



메서드 분리

중복 코드나 복잡한 로직을 별도 메서드로 추출하여 재사용성과 가독성을 높입니다.

ExtractDelimiter(), SplitNumbers() 등의 헬퍼 메서드 추출



파라미터화

유사한 로직을 매개변수를 받는 단일 메서드로 통합하여 중복을 제거합니다.

여러 구분자 처리 → Parse(string input, string delimiter)



의도 드러내는 이름

메서드와 변수 이름을 통해 코드의 의도를 명확히 표현합니다.

s → numbers, p → customDelimiter, Add → Calculate



불변/순수 함수

사이드 이펙트 없는 순수 함수를 작성하여 안정성을 높입니다.

상태 변경 없이 입력에 따른 결과만 반환하는 메서드 설계



리팩토링의 중요성

TDD에서 리팩토링은 테스트가 성공한 후(Green) 반드시 수행해야 하는 단계입니다. 코드가 올바르게 동작하는지 확인한 후에 코드의 품질을 개선하는 과정입니다. 지속적인 리팩토링은 코드의 기술 부채를 관리하고, 유지보수성을 높이는 핵심 프랙티스입니다.

예제2: 쇼핑카트 모델링



온라인 쇼핑카트 시스템

TDD 방법론을 적용해 장바구니 시스템을 설계하고 구현합니다.



실제 비즈니스 시나리오에서 TDD 적용 사례로, 모델 설계부터 테스트, 구현까지의 전체 과정을 경험합니다.

도메인 모델링

Item (상품)

상품 ID, 이름, 가격, 수량 속성을 가집니다.

할인 적용 가능 여부와 품질 상태를 표시합니다.

Cart (장바구니)

상품 추가/수정/삭제 및 총액 계산 기능을 제공합니다.

쿠폰 코드 적용 및 해제 기능을 구현합니다.

PricingRule (가격 정책)

다양한 할인 규칙 적용: BOGO(하나 사면 하나 무료), 대량 구매 할인, 비율 할인

여러 할인 규칙 조합과 시간/지역별 가격 정책을 설계합니다.

예제2 규칙 테스트 (1/2)

쇼핑카트 시스템의 다양한 할인 규칙 테스트 방법 - BOGO와 Bulk 할인

1 BOGO(Buy One Get One) 할인 테스트

동일 상품 2개 구매 시 1개 무료 할인 검증

```
[Test]
public void BogoDiscount_WhenTwoItems_AppliesDiscount()
{
    var cart = new ShoppingCart();
    cart.AddItem(new Product { Price = 100.0m }, 2);
    var result = new BogoDiscountRule().ApplyDiscount(cart);
    Assert.AreEqual(100.0m, result); // 2번째 상품 무료
}
```

2 Bulk 할인 테스트

동일 상품 3개 이상 구매 시 10% 할인 적용

```
[[Test]
public void BulkDiscount_WhenThreeOrMore_AppliesDiscount()
{
    var cart = new ShoppingCart();
    cart.AddItem(new Product { Price = 100.0m }, 3);
    var result = new BulkDiscountRule(0.1m).ApplyDiscount(cart);
    Assert.AreEqual(270.0m, result);
}
```

테스트 전략 팁

각 할인 규칙은 단독 테스트 후 규칙 간 충돌이나 우선순위를 테스트하세요. 다음 페이지에서 Percent 할인과 엣지케이스를 다룹니다.

예제2 규칙 테스트 (2/3)

쇼핑카트 시스템의 다양한 할인 규칙을 테스트하는 방법 - Percent 할인

3

Percent 할인 테스트

구매 금액의 특정 비율을 할인해주는 규칙 테스트

```
[Test]
public void ApplyPercentDiscount_Should_Reduce_Price_By_Percentage()
{
    // Arrange
    var cart = new ShoppingCart();
    var item = new Item("노트북", 1000000);
    cart.AddItem(item, 1);
    var rule = new PercentDiscountRule(10); // 10% 할인

    // Act
    rule.ApplyDiscount(cart);

    // Assert
    Assert.AreEqual(900000, cart.TotalAmount); // 10% 할인 후 금액
}
```

테스트 작성 팁

할인율 변경 테스트, 최소/최대 할인 금액 제한 테스트, 소수점 처리 테스트 등 다양한 비율 할인 시나리오를 고려하세요. 각 테스트는 단일 기능에 집중해야 합니다.

설계 진화와 테스트

테스트 주도 설계

테스트를 먼저 작성하는 과정에서 자연스럽게 인터페이스가 설계됩니다. 테스트 코드는 '첫 번째 클라이언트'가 되어 사용자 관점에서 직관적인 API를 유도합니다.



응집도 향상

테스트를 먼저 작성하면 단일 책임 원칙을 자연스럽게 따르게 되어 높은 응집도를 가진 설계가 만들어집니다.



결합도 감소

테스트 격리를 위해 의존성 주입과 인터페이스를 활용해 컴포넌트 간 결합도를 낮춥니다.



지속적 개선

새로운 요구사항마다 테스트를 먼저 작성하면서 API가 점진적으로 진화하고 개선됩니다.



설계 진화 예시

주문 시스템 인터페이스 진화:

```
// 1단계: 기본 주문 생성
interface IOrderService {
    Order CreateOrder(Customer customer, List<Item> items);
}
// 2단계: 할인 규칙 추가
interface IOrderService {
    Order CreateOrder(Customer customer, List<Item> items);
    Order ApplyDiscount(Order order, IDiscountRule rule);
}
```

Given-When-Then 스타일



Given (준비)

테스트를 위한 전제 조건과 초기 상태를 설정
특정 조건과 컨텍스트를 명시적으로 정의



When (행동)

테스트 대상 행위를 수행하는 단계
명확한 하나의 행동에 집중



Then (검증)

기대하는 결과와 상태 변화를 검증
관찰 가능한 행동이나 상태에 집중



도메인 언어 반영

비즈니스 용어와 규칙을 테스트에 반영
개발자와 비기술팀 간 소통 향상



BDD 스타일의 이점

Given-When-Then 구조는 테스트 명세를 명확하게 하여 가독성을 높이고, 테스트의 의도를 분명히 전달합니다. 테스트가 문서의 역할도 겸하게 되어 요구사항과 행동을 함께 명세할 수 있습니다.

Outside-in vs Inside-out

Outside-in (Top-down)

접근 방식

사용자 관점 우선 (BDD 철학) API, UI부터 설계

작업 흐름

상위→하위 레벨 진행
Mock 활용도 높음

특징/장단점

더블 루프 TDD (인수+단위) + 사용자 중심 설계 - Mock 의존성 증가

추천 사용처

UI/서비스 개발, 불명확한 요구사항

Inside-out (Bottom-up)

도메인 모델 우선 (전통 TDD) 핵심 로직부터 구현

하위→상위 레벨 진행
실제 구현체 선호

클래식 TDD (단위 중심) + 도메인 로직 명확화 - 전체 방향성 놓칠 수 있음

💡 실무 적용 가이드

실무에서는 두 방식을 상황에 따라 혼합하여 사용하는 것이 효과적입니다. 새로운 기능은 Outside-in으로 흐름을 파악한 후, 복잡한 로직은 Inside-out으로 접근하는 하이브리드 방식이 권장됩니다.

TDD 실전 요약

TDD를 실무에서 효과적으로 적용하기 위한 핵심 원칙과 실천 방법입니다.

작게 시작 - 자주 피드백

구현 범위를 최소화하고 Red-Green-Refactor 사이클을 빠르게 반복하여 방향성 검증을 수시로 진행하세요.

테스트는 명세

테스트는 실행 가능한 문서입니다. 테스트가 요구사항을 명확히 표현하면 코드 의도가 분명해집니다.

리팩토링은 습관

테스트 통과 후에는 반드시 리팩토링하세요. 코드 중복 제거와 가독성 향상에 지속적으로 투자하세요.

점진적 설계

설계는 점진적으로 발전시키고, YAGNI 원칙에 따라 과도한 추상화는 필요할 때만 도입하세요.

핵심 요약

TDD는 단순한 테스트 작성 방법론이 아닌 소프트웨어 설계 접근법입니다. 테스트부터 시작하면 명확한 인터페이스와 유지보수 가능한 코드를 작성할 수 있습니다. 절차 보다 원칙을 이해하고 유연하게 적용하는 것이 중요합니다.



Part 6

고급 테스트 기법

예외 처리, 경계값, 비동기, 파라미터화된 테스트 등
고급 테스트 기법으로 테스트 품질 향상

예외 처리 테스트



Assert.Throws / Assert.ThrowsAsync

예외 발생 명시적 검증

비동기 메서드는 ThrowsAsync 사용



예외 메시지 / 도메인 예외

메시지 내용 및 속성 검증

도메인 특화 예외 클래스 활용



실패 경로 우선 검증

실패 케이스 먼저 작성

경계조건 테스트에 집중



예외 처리 패턴

try-catch 블록 테스트

예외 변환 및 복구 로직 검증



코드 예시

```
[Test]
public void Divide_ByZero_ThrowsDivideByZeroException()
{
    var calculator = new Calculator();
    Assert.Throws<DivideByZeroException>(() => calculator.Divide(10, 0));
}
```

경계값/동등 분할

테스트 케이스 설계 기법

경계값 분석은 경계 부근에서 결함이 발생하기 쉽다는 점을 활용해 경계값을 집중 테스트하며, 동등 분할은 입력값을 동등한 그룹으로 나누어 각 그룹에서 대표값만 테스트합니다.

경계값 분석

최소값, 최대값, 경계 내/외부 값 테스트. 예: 1~100 범위면 0,1,100,101 테스트

동등 분할

유사 동작 입력을 그룹화하고 각 그룹당 대표 케이스만 테스트. 예: 양수, 0, 음수

결정 테이블

여러 입력 조건 조합을 테스트. 복잡한 비즈니스 규칙이나 조건 테스트에 적합

💡 실전 예시: 나이 확인 기능(18~65세)

```
[Theory]
[InlineData(17, false)] // 경계 외부(하한 미만)
[InlineData(18, true)] // 경계값(하한)
[InlineData(40, true)] // 동등 분할 내부값
[InlineData(65, true)] // 경계값(상한)
[InlineData(66, false)] // 경계 외부(상한 초과)
public void IsEligible_ChecksAgeBoundaries(int age, bool expected)
{
    Assert.Equal(expected, userService.IsEligible(age));
}
```

비동기/병렬 테스트



async/await 테스트 패턴

비동기 메서드는 async Task 반환 형식으로 작성
반드시 await 사용 (fire-and-forget 지양)
테스트 메서드에도 async 키워드 필수



타임아웃 처리

Task.Delay로 비동기 동작 시뮬레이션
NUnit: [Timeout(1000)] 어트리뷰트
xUnit: TimeoutAttribute 패키지 설치 필요



취소 토큰 테스트

CancellationTokenSource로 취소 시뮬레이션
token.ThrowIfCancellationRequested() 검증
취소 후 OperationCanceledException 기다



스레드 안전성 검증

Parallel.ForEach로 경쟁 상태 테스트
Interlocked 클래스 활용 검증
불변성 보장 패턴 우선 적용



비동기 테스트 모범 사례

async void는 테스트하기 어려우므로 항상 async Task 사용. 테스트에서는 await를 사용하여 비동기 작업 완료를 보장하고, 모의 객체에서 비동기 메서드를 설정할 때 ReturnsAsync()를 활용하세요. 경쟁 상태나 교착 상태를 방지하려면 테스트 격리와 면등성을 유지하는 것이 중요합니다.

시간 의존 코드

시간 의존 코드의 문제점

DateTime.Now나 DateTimeOffset.UtcNow를 직접 사용하는 코드는 테스트하기 어렵습니다. 테스트 실행 시점에 따라 결과가 달라져 효과적인 테스트를 위해 시간을 제어할 수 있는 추상화가 필요합니다.

IClock 인터페이스

시간 제공을 추상화하는 인터페이스로 의존성 주입. DateTime.Now 대신 clock.Now를 사용합니다.

가짜 시계

테스트에서는 고정된 시간을 반환하거나, 원하는 시간으로 조작 가능한 FakeClock을 사용합니다.

시간 이동 기능

AdvanceBy()나 SetCurrentTime() 같은 메서드로 테스트 중 가상 시간을 조작해 검증합니다.

구현 예시

```
public interface IClock {
    DateTime Now { get; }
}
// 테스트용 가짜 시계
public class FakeClock : IClock {
    private DateTime _now = DateTime.Now;
    public DateTime Now => _now;
    public void SetTime(DateTime time) => _now = time;
}
```

파일/IO/환경 격리



Temp 디렉터리 사용

Path.GetTempPath()를 활용한 임시 파일 테스트
테스트 완료 후 자동 정리 (Setup/TearDown)



환경변수 주입/복원

테스트용 환경변수 값 설정 및 테스트 후 원래 값 복원
Environment.SetEnvironmentVariable 활용



IO 인터페이스화

IFileSystem, IFile 등 인터페이스 도입
System.IO.Abstractions 라이브러리 활용



네트워크/소켓 격리

로컬 포트 사용 테스트
네트워크 래퍼 클래스로 의존성 분리



테스트 격리 원칙

외부 환경에 의존하는 테스트는 불안정하고 실행 환경에 영향을 받기 쉽습니다. 파일 시스템, 환경 변수, 네트워크 등 외부 의존성을 인터페이스로 추상화하고 테스트 중에는 가짜 구현체를 사용하여 안정적이고 격리된 테스트를 만들어야 합니다.

랜덤/비결정성 제어



시드 고정

```
var rnd = new Random(42); // 고정 시드  
테스트마다 동일한 난수 시퀀스 생성
```



랜덤 주입

IRandomProvider 인터페이스 생성
테스트에서 예측 가능한 구현체 주입



분포 검증

모의 난수 생성 결과의 통계적 특성 검증
예상 범위 내 결과값 확인



비결정성 격리

Guid.NewGuid() 등 정적 메서드 래핑
테스트에서 예측 가능한 값으로 대체



랜덤 테스트 패턴

테스트 가능한 코드는 비결정적 요소(시간, 난수)를 의존성으로 주입받아야 합니다. 랜덤 요소가 필요한 테스트는 결정적인 부분과 확률적인 부분을 명확히 구분하고, 확률적 특성만 별도로 테스트합니다.

스냅샷 대안



명시적 Assert

기대 값을 명시적으로 선언하여 테스트 의도를 분명히 표현
스냅샷보다 테스트 가독성과 유지보수성 향상



포맷 변화에 강한 검증

객체 구조나 값에 집중하고 형식 변경에 민감하지 않은 검증
시각적 요소/JSON 포맷 변경에도 테스트 안정성 유지



구조적 검증

전체 객체 대신 핵심 속성만 검증하는 방식
FluentAssertions나 확장 메서드로 가독성 있는 검증



객체 비교 라이브러리

CompareNETObjects, DeepEqual 등 활용
지능적인 차이 비교와 무시할 속성 설정 가능



베스트 프랙티스

스냅샷 테스트는 UI 컴포넌트나 직렬화된 결과물처럼 복잡한 출력에 유용할 수 있지만, 단위 테스트에서는 명시적 검증이 더 효과적입니다. 테스트가 실패했을 때 무엇이 왜 실패했는지 명확히 알 수 있는 방식으로 작성하세요.

성능/회귀 테스트 구분

테스트 유형 분리의 중요성

단위 테스트와 비기능 테스트(성능, 부하)는 목적이 다르므로 분리해야 합니다. 단위 테스트는 기능 정확성 검증에 집중하고, 성능 테스트는 시스템 자원 사용과 응답 시간에 초점을 맞춥니다. 두 유형을 명확히 구분하여 각각 최적화하세요.

실행 속도 차이

단위 테스트는 밀리초 단위로 빠르게 실행되지만, 성능 테스트는 유의미한 데이터 수집을 위해 더 오래 실행됩니다. CI 파이프라인에서 분리하세요.

환경 요구사항

단위 테스트는 모든 환경에서 일관된 결과를 보이지만, 성능 테스트는 실제와 유사한 환경에서 실행해야 의미가 있습니다.

평가 기준

단위 테스트는 성공/실패의 이분법적 결과를 제공하지만, 성능 테스트는 응답 시간, 처리량, 자원 사용률 등 다양한 지표로 평가합니다.

구분 방법

프로젝트 구조와 명명 규칙으로 테스트 유형 분리:

```
OrderService.Tests.Unit // 단위 테스트  
OrderService.Tests.Integration // 통합 테스트  
OrderService.Tests.Performance // 성능 테스트  
[TestCategory("Unit")] // 단위 테스트 태그  
[TestCategory("Performance")] // 성능 테스트 태그
```

고급 기법 요약

단위테스트의 고급 기술을 효과적으로 적용하기 위한 핵심 원칙과 베스트 프랙티스입니다.

결정적인 테스트

시간, 랜덤, 환경 변수는 인터페이스로 격리하세요. 날짜/시간은 IClock 패턴, 랜덤은 시드 고정이나 주입 패턴을 사용합니다.

경계값과 실패 시나리오

경계값 분석과 동등 분할로 테스트를 설계하세요. 정상 경로보다 예외와 실패 경로를 먼저 테스트하면 더 견고해집니다.

명시적 어서션

스냅샷보다 명시적 Assert로 의도를 표현하세요. Given-When-Then 구조로 테스트 목적과 검증 대상을 명확히 하세요.

비동기 코드 테스트

async/await 패턴을 사용하고, 타임아웃과 취소 토큰을 테스트하세요. 병렬 코드는 경쟁 상태를 고려한 설계가 필요합니다.

핵심 요약

고급 테스트의 핵심은 결정성과 격리입니다. 외부 요인을 제어하는 패턴을 익히고, 성능 테스트와 단위 테스트를 구분하세요. 모든 테스트는 의도가 명확하고, 실행이 빠르며, 결과가 일관되어야 합니다.



Part 7

코드 커버리지

지표 이해와 활용, ReportGenerator, 커버리지 분석,
실무적인 코드 품질 향상을 위한 측정 전략

커버리지 지표

코드 커버리지란?

코드 커버리지는 테스트 실행 중 코드의 어느 부분이 실행되었는지 측정하는 지표입니다. 테스트 품질과 미테스트 영역을 파악하는 데 유용합니다.

라인 커버리지

실행된 코드 라인의 비율. 가장 기본적이고 직관적인 지표로 미실행 코드를 빠르게 식별합니다.

브랜치 커버리지

조건문(if, switch)의 분기 테스트 여부. 모든 의사결정 경로가 테스트되었는지 검증합니다.

메서드 커버리지

테스트된 메서드의 비율. 어떤 기능이 테스트에서 완전히 누락되었는지 파악하는 데 유용합니다.

커버리지 리포트 예시

```
Summary:  
Line: 82.5% (165/200) | Branch: 76.3% (45/59)  
Method: 90.1% (91/101) | Class: 95.0% (19/20)
```

 커버리지 목표는 코드의 중요도와 위험도에 따라 차등 설정하는 것이 효과적

Coverlet 설정

Coverlet은 .NET Core 프로젝트를 위한 코드 커버리지 도구입니다. 쉽게 설치하고 실행할 수 있습니다.

>_ Coverlet 설치 및 실행

설치

```
# 프로젝트에 Coverlet 패키지 추가  
dotnet add package coverlet.collector  
  
# 코드 커버리지 수집을 포함하여 테스트 실행  
dotnet test --collect:"XPlat Code Coverage"  
  
# 더 많은 옵션을 지정하여 실행 dotnet test /p:CollectCoverage=true /p:CoverletOutputFormat=cobertura
```

생성된 결과 파일

테스트를 실행하면 ./TestResults/ 경로에 커버리지 파일이 생성됩니다. 기본 형식은 Cobertura XML로, ReportGenerator와 함께 사용할 수 있습니다.

ReportGenerator 사용

도구

코드 커버리지 결과를 시각적인 HTML 보고서로 변환하는 도구입니다.

> ReportGenerator 설치

```
# 글로벌 도구로 설치
```

```
dotnet tool install -g dotnet-reportgenerator-globaltool
```

↳ 리포트 생성

```
reportgenerator -reports:**/coverage.cobertura.xml -targetdir:coveragereport -reporttypes:Html
```

i 주요 매개변수

- reports: 커버리지 결과 파일 경로 (와일드카드 지원)
- targetdir: 보고서를 생성할 대상 디렉토리
- reporttypes: 생성할 보고서 형식 (Html, Cobertura, Badges 등)

리포트 해석



미커버 라인 식별

리포트의 빨간색/노란색 강조 부분 확인
라인 단위 커버리지와 코드 컨텍스트 연결



중요 분기 우선 보강

예외 처리 분기가 미커버된 경우 우선 보강
핵심 비즈니스 로직의 조건부 분기 집중



추세 분석

단일 수치보다 시간에 따른 커버리지 변화 확인
코드 추가/수정에 따른 커버리지 영향 모니터링



가치 기반 개선

단순한 수치 채우기보다 비즈니스 가치 고려
오류 발생 위험이 높은 영역에 테스트 집중



리포트 해석 팁

HTML 보고서의 드릴다운 기능을 활용하여 클래스-메서드 단위로 상세 분석하고, 커버리지가 낮은 모듈의 빈발 패턴을 식별해 공통 해결책을 마련하세요. 수치보다는 품질에 집중하여 의미 있는 테스트를 작성하는 것이 중요합니다.

커버리지 목표/KPI

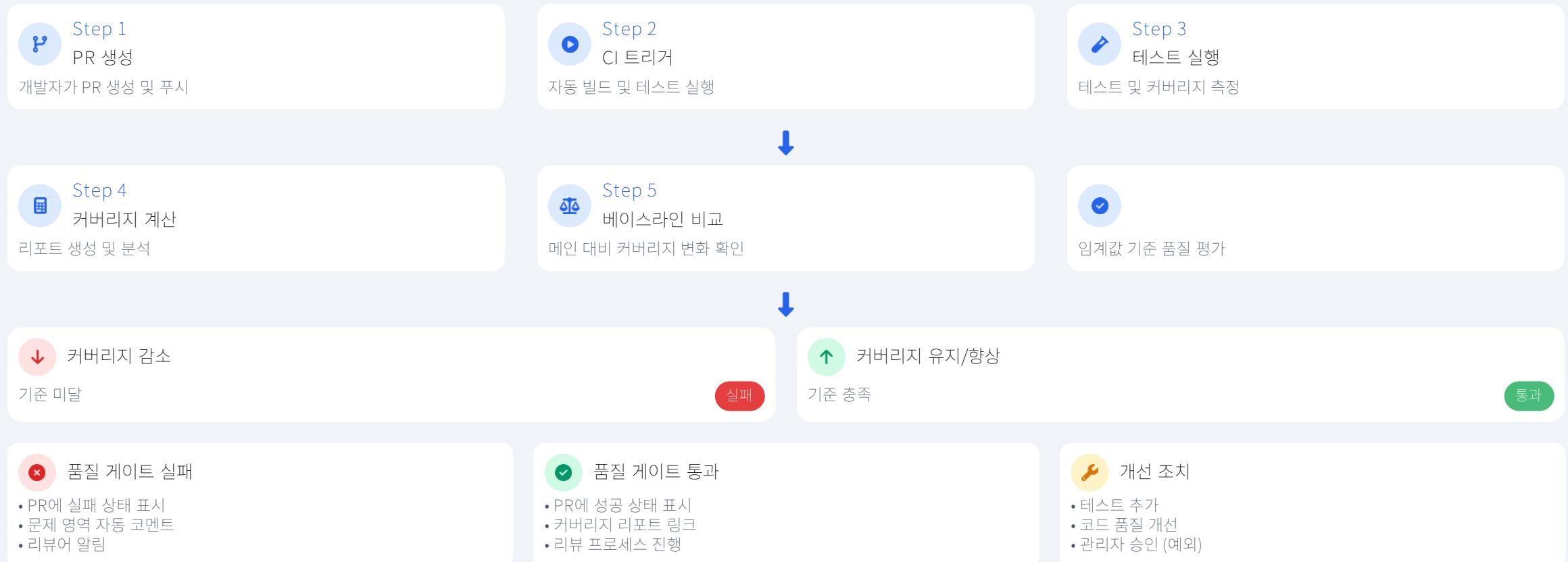
 핵심 모듈	 비즈니스 로직	 인프라/UI	
라인 커버리지	85-95% 높은 품질 요구	70-80% 주요 경로 우선	50-70% 중요 기능 중심
브랜치 커버리지	90%+ 모든 분기 검증	75%+ 주요 결정 경로	40-60% 핵심 경로만
변화 기반 목표	커버리지 감소 방지 변경에 테스트 추가	점진적 개선 변경사항 중심	점진적 개선 핵심 기능 위주
KPI 관리	절대 수치 모니터링 CI 게이트/PR 조건	변화율 모니터링 주간/월간 리포팅	트렌드 모니터링 분기별 리뷰
리스크 관리	고위험, 목표 미달성 시 빌드 실패/경고	중위험, 감소 추세 시 코드 리뷰 강화	저위험, 테크 부채 개선 계획 수립

커버리지 품질 관리 원칙

단순 수치가 아닌 품질 지표로 활용하세요. 모듈 중요도와 위험도에 따라 차등화된 목표를 설정하고, 개선 추세를 관리하는 것이 효과적입니다. 커버리지 감소 영역을 집중 모니터링하고 지속적 개선을 위한 피드백 루프를 구축하세요.

PR 품질 게이트

CI에서 커버리지 체크를 통한 코드 품질 관리 프로세스



💡 모듈 중요도별 차등 커버리지 임계값 설정 및 감소량(-1% 이상)을 기준으로 품질 게이트를 설정하는 것이 효과적입니다.

커버리지 요약/실습

코드 커버리지 분석과 개선을 위한 핵심 원칙과 실습 가이드입니다.

측정-분석-개선 사이클

단순 수치보다 중요 기능의 테스트 품질을 분석하고 점진적 개선에 집중하세요.

핵심 모듈 우선 전략

비즈니스 로직은 80% 이상, UI/인프라는 낮은 목표로 차등화 전략을 적용하세요.

리포트 활용 기법

분기 커버리지가 낮거나 복잡도 높은 미테스트 코드를 식별하고 개선하세요.

CI/CD 통합 방법

자동 리포트 생성 및 커버리지 하락 시 빌드 실패 전략을 도입하세요.

실습 가이드

- 저커버리지 모듈 선택 및 ReportGenerator 리포트 생성
- 미커버 라인 분석 및 테스트 케이스 추가
- GitHub Actions에 coverlet.collector 통합
- PR에 커버리지 변화 표시 스크립트 추가



Part 8

테스트 자동화/CI-CD

GitHub Actions 파이프라인 구축, 테스트 자동화,
Jenkins 연동, 빌드-테스트-배포 통합 워크플로우

CI/CD 개요

CI/CD란?

CI/CD(지속적 통합/지속적 배포)는 코드 변경을 더 자주, 안정적으로 제공하는 자동화된 소프트웨어 릴리스 프로세스입니다. CI는 코드 변경을 자동으로 빌드/테스트하며, CD는 검증된 코드를 배포하여 개발-배포 사이클을 가속화합니다.

지속적 통합(CI)

코드 변경사항을 공유 저장소에 자주 통합하고, 자동화된 빌드와 테스트로 버그를 조기 발견하여 품질을 보장합니다.

지속적 배포(CD)

테스트 통과 코드를 자동으로 스테이징/프로덕션 환경에 배포하여 릴리스 프로세스를 간소화하고 출시 시간을 단축합니다.

자동화 이점

수동 작업 감소, 휴먼 에러 방지, 피드백 주기 단축으로 개발 생산성과 소프트웨어 품질을 향상시킵니다.

CI/CD 파이프라인 예시



GitHub Actions 기본



워크플로우 (Workflow)

YAML 형식의 자동화 프로세스

on: 이벤트 트리거(push, pull_request)



jobs

워크플로우의 실행 단위

독립 환경에서 병렬/순차 실행



steps

uses: 미리 정의된 액션 사용

run: 쉘 명령어 실행(dotnet test)



러너 (Runner)

runs-on: 실행환경 지정

호스팅/셀프호스팅 러너 선택



주요 액션 예시

actions/checkout, setup-dotnet, upload-artifact, cache

.NET 매트릭스 빌드/테스트

OS와 SDK 버전 조합에서 병렬 테스트 실행으로 크로스 플랫폼 호환성 검증

 **매트릭스 정의**

```
strategy :  
matrix :  
os: [ubuntu-latest, windows-latest]  
dotnet: ['6.0', '8.0']
```

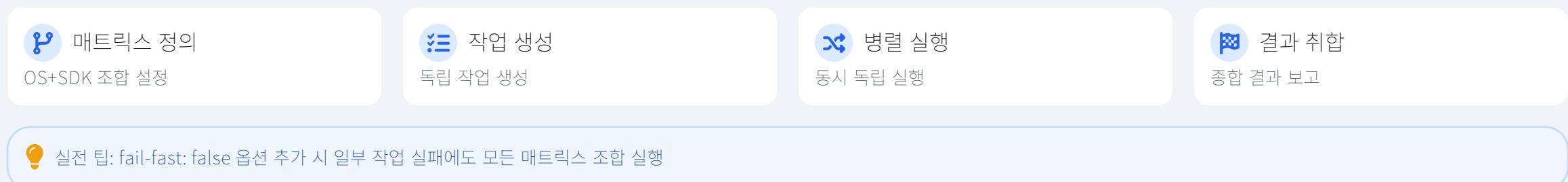
 **병렬 실행 이점**

- 빌드/테스트 시간 단축
- 호환성 문제 조기 발견
- 크로스 플랫폼 이슈 식별

 **매트릭스 실행 결과**

환경	결과
Ubuntu + .NET 6.0	✓ 성공
Ubuntu + .NET 8.0	✓ 성공
Windows + .NET 6.0	✓ 성공
Windows + .NET 8.0	✗ 실패

매트릭스 워크플로우 프로세스



커버리지 아티팩트 업로드

GitHub Actions에서 커버리지 리포트를 아티팩트로 저장하고 공유하는 방법입니다.

1 커버리지 리포트 생성

테스트 실행 단계에서 `dotnet test --collect:"XPlat Code Coverage"` 명령어 사용

2 ReportGenerator 실행

생성된 커버리지 파일을 HTML 리포트로 변환

3 actions/upload-artifact 설정

워크플로우에 아티팩트 업로드 단계 추가:

```
- name: Upload coverage report
uses: actions/upload-artifact@v3
with: { name: code-coverage-report, path: coveragereport/ }
```

4 아티팩트 활용

GitHub Actions 'Artifacts' 섹션에서 리포트 다운로드 및 공유

팁

큰 리포트는 압축하여 업로드하면 시간과 용량을 절약할 수 있습니다. `if-no-files-found: error` 옵션으로 리포트 생성 실패 시 워크플로우를 중단할 수 있습니다.

PR 상태 배지/코멘트



상태 배지 추가

README.md에 코드 품질 배지 추가

coverage: 85% tests: passing

shields.io 또는 codecov 배지로 가시성 향상



리포트 링크

GitHub Pages로 리포트 게시

PR 코멘트에 리포트 URL 자동 추가

형식: github.io/<repo>/coverage/



요약 코멘트 자동화

PR 코멘트 자동 생성 (GitHub Actions)

테스트 결과, 커버리지 변화 정보 포함

팀 리뷰 효율화 및 코드 품질 가시성 제공



커버리지 추이 시각화

커버리지 변화량 표시 (+2.5%)

변경 코드의 커버리지 별도 강조

임계치 미달 시 시각적 경고 표시



구성 예시: GitHub Actions

```
uses: marocchino/sticky-pull-request-comment@v2 message: "## 테스트 결과:  통과 | 커버리지: 85% (+2.3%)"
```

Jenkins 파이프라인(선택)

Jenkins 파이프라인이란?

Jenkins Pipeline은 CI/CD 파이프라인을 코드로 정의하는 방식으로, 빌드, 테스트, 배포 과정을 자동화합니다. Jenkinsfile을 통해 파이프라인을 정의하고 버전 관리 시스템과 함께 관리할 수 있습니다.

Declarative Pipeline

Groovy 기반 선언적 문법으로, 'pipeline' 블록을 사용해 구조화된 형태로 파이프라인을 정의합니다.

Stages & Steps

Stages는 파이프라인의 주요 단계를, Steps는 각 단계의 실행 작업을 정의합니다.

환경 & 에이전트

환경변수 설정 및 실행 환경을 지정하여 다양한 환경에서 동일한 파이프라인을 실행합니다.

Declarative Pipeline 예시

```
pipeline {
    agent any
    stages {
        stage('Build') { steps { sh 'dotnet build' } }
        stage('Test') { steps { sh 'dotnet test --collect:"XPlat Code Coverage"' } }
        stage('Report') { steps { sh 'reportgenerator -reports:**/coverage.cobertura.xml -targetdir:report' } }
    }
}
```

시크릿/보안

CI/CD 파이프라인에서 자격 증명과 토큰을 안전하게 관리하기 위한 보안 체크리스트입니다.



GitHub Secrets 관리

리포지토리 Secrets 설정

Settings > Secrets에서 시크릿을 설정하여 워크플로우에서 안전하게 참조



토큰 최소 권한

최소 권한 원칙

필요한 최소 권한만 부여한 토큰 생성 (읽기 전용 등)

환경별 Secrets 분리

개발/테스트/운영 환경별 시크릿 분리 및 환경별 승인 설정

서드파티 액션 검증

신뢰할 수 있는 소스의 액션만 사용, 토큰 접근 액션 철저 검토

시크릿 순환 정책

정기적 토큰/시크릿 교체 및 노출 시 즉시 갱신

로그 보안 처리

::add-mask::로 민감 정보 로그 노출 방지, 디버그 출력 제한

보안 모범 사례

GitHub Actions에서 \${{ secrets.SECRET_NAME }} 문법으로 시크릿 참조. 절대 시크릿을 직접 하드코딩하거나 커밋하지 마세요.

병렬 테스트/샤딩



테스트 샤딩(Sharding)

큰 테스트 묶음을 여러 샤드로 분할
각 샤드를 별도 머신/프로세스에서 실행
전체 CI 파이프라인 시간 단축



분산 전략

실행 시간 기반 균등 분배
xUnit: %assemblyName%.dll --list-tests
NUnit: TestContext.Parameters["shard"]



Flaky 테스트 관리

비결정적 테스트 식별 및 라벨링
재시도 전략: [Retry(3)]
격리된 별도 실행 파이프라인 구성



병렬 실행 최적화

xUnit: [Collection("이름")] 격리
NUnit: [Parallelizable], MaxCpuCount
공유 자원 충돌 회피 설계



병렬 실행 모범 사례

테스트 간 상태 격리, 공유 리소스 접근 관리, 순서 의존성 제거, 외부 서비스 목킹을 통해 실제 환경에서도 안정적인 병렬 테스트 실행을 확보합니다.

실패 진단/로그

테스트 로그 수집 및 실패 진단, 플레이키(Flaky) 테스트 관리 프로세스



워크플로우 예시(yaml)

YAML

GitHub Actions 워크플로우 파일의 핵심 구성 예시입니다.

dotnet-tests.yml

GitHub Actions

```
name: .NET CI/CD Pipeline
on: [push, pull_request]
jobs:
  build-and-test:
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        os: [ ubuntu-latest, windows-latest ]
        dotnet-version: [ 8.0.x ]
    steps:
      # 코드 체크아웃
      - uses: actions/checkout@v3
      # .NET SDK 설정
      - name: Setup .NET
        uses: actions/setup-dotnet@v3
        with:
          dotnet-version: ${{ matrix.dotnet-version }}
      # 의존성 복원 및 빌드
      - run: dotnet restore && dotnet build --no-restore
      # 테스트 실행 및 커버리지 수집
      - run: dotnet test --no-build --collect:"XPlat Code Coverage"
```

주의 기능

이 워크플로우는 멀티 OS 환경에서 테스트를 자동 실행하고, 코드 커버리지 리포트를 생성하여 아티팩트로 저장합니다. PR이나 커밋 발생 시 자동으로 실행됩니다.

자동화 요약/체크리스트

테스트 자동화 및 CI/CD 파이프라인 구현을 위한 필수 체크리스트입니다.

필수 구현 항목

모든 PR 자동 테스트

모든 Pull Request에서 자동 테스트 실행 및 PR 머지 전 테스트 통과 확인

코드 커버리지 보고

테스트 실행 후 커버리지 수집 및 ReportGenerator 보고서 생성

품질 게이트

커버리지 최소 기준 설정 및 미달 시 워크플로우 실패 구성

시크릿 보안 관리

токен, API 키 등을 GitHub Secrets으로 안전하게 관리

권장 구현 항목

매트릭스 빌드 구성

다양한 OS 및 .NET 버전 조합에서의 매트릭스 테스트 실행

아티팩트 업로드

테스트 결과, 커버리지 보고서, 로그 파일 등의 아티팩트 저장

상태 배지와 PR 코멘트

README 파일에 빌드/테스트 상태 배지 및 PR 결과 자동 코멘트

자동화 구현 팁

NuGet 패키지와 도구를 캐싱하여 파이프라인 실행 시간을 최대 50% 단축하세요. 가독성을 위해 워크플로우를 작업별로 분할하세요.



Part 9

종합 프로젝트

팀별 TDD·자동화 적용 실습 프로젝트,
단위테스트와 CI/CD 파이프라인 구축을 통한 실무 역량 강화

프로젝트 개요/요구



팀별 종합 프로젝트

단위테스트, TDD, Moq, 자동화 기술을 활용하여 실무 환경에서 프로젝트를 수행합니다.

팀별로 도메인을 선택하고 품질 높은 테스트 코드와 함께 개발합니다.

비기능 요구사항(성능, 보안)은 제외하고, 단위테스트 및 TDD 적용에 집중합니다.

프로젝트 요구사항

기능적 요구사항

선택한 도메인의 핵심 비즈니스 로직 구현 (5개 이상)

모든 핵심 로직의 단위테스트 (코드 커버리지 80% 이상)

최소 2개 이상 기능은 TDD로 개발 (커밋으로 증명)

기술적 요구사항

Moq를 활용한 외부 의존성 격리 (최소 1개 이상)

GitHub Actions CI 파이프라인 구축

ReportGenerator 커버리지 리포트 생성 및 PR 코멘트

제출물

GitHub 리포지토리 (소스코드 및 테스트 코드)

TDD 계획서 문서 및 팀 회고 문서

최종 데모 발표 자료 (5분 내외)

아키텍처/폴더 구조

프로젝트 구조 설계

테스트 효율성을 위해 관심사 분리와 의존성 역전 원칙을 적용한 구조가 필요합니다. 도메인 중심 설계로 비즈니스 로직과 인프라 코드를 분리하여 테스트 용이성을 높입니다.



소스와 테스트 분리

src/ 폴더와 tests/ 폴더로 분리하고, 테스트 프로젝트는 [원본 프로젝트].Tests 형식으로 명명합니다.



도메인 중심 설계

Domain, Application, Infrastructure 계층으로 구분하여 책임을 명확히 합니다.



인터페이스 기반 설계

의존성 주입을 위한 인터페이스를 활용해 테스트 용이성과 커포넌트 교체를 용이하게 합니다.



프로젝트 폴더 구조 예시

```
Solution/  
└── src/  
    └── ProjectName.Domain/  
    └── ProjectName.Application/  
    └── ProjectName.Infrastructure/
```

```
Solution/ (계속)  
└── tests/  
    └── ProjectName.Domain.Tests/  
    └── ProjectName.Application.Tests/  
    └── ProjectName.Infrastructure.Tests/
```

스프린트/역할 분담

팀별 종합 프로젝트를 위한 협업 프로세스와 역할 분담



Day 1

스프린트 계획

요구사항 분석, 작업 분할



Day 1-2

구현 단계

테스트 작성, 구현, 페어링



Day 3

리뷰/회고

코드 리뷰, 테스트 커버리지 확인



일간 싱크업 (15분)

진행 상황, 차단 요소, 계획 공유 - 매일 오전 10시

역할 분담



제품 책임자

요구사항 명확화
우선순위 결정
완료 여부 확인



개발자

테스트 코드 작성
기능 구현 (TDD)
리팩토링



테스트 담당

테스트 전략 수립
커버리지 검증
CI 파이프라인 관리



협업 도구

GitHub Issues - 작업 관리 | PR - 코드 리뷰 | Teams/Slack - 소통 | 팀 빌드 서버 - CI/CD

코드 기능 단위테스트 교육 | 스프린트/역할 분담

TDD 계획서 템플릿

효과적인 TDD 프로젝트 수행을 위한 계획서 템플릿입니다.

1 테스트 리스트 작성

구현할 기능을 작은 단위로 분해하고, 각 단위의 테스트 목록을 작성합니다.
예: "빈 문자열 입력 시 0 반환", "단일 숫자 시 해당 값 반환"

2 우선순위 지정

테스트 케이스에 우선순위를 부여하고 단순한 케이스부터 시작합니다.
우선순위: High(필수), Medium(중요), Low(선택)

3 리스크/가정 문서화

프로젝트 진행 시 예상되는 리스크와 가정 사항을 명시합니다.
예: "외부 API는 200ms 이내 응답", "유효 입력만 가정"

4 완료 정의(DoD) 설정

테스트가 "완료"된 기준을 명확히 설정합니다.
DoD 예시: 모든 테스트 통과, 코드 커버리지 80%+, 리팩토링 완료, 코드 리뷰 승인

팁

TDD 계획서는 살아있는 문서입니다. 개발 중 새로운 테스트 케이스가 발견되면 지속적으로 업데이트하세요.

코드 리뷰 체크리스트

테스트 코드를 리뷰할 때 확인해야 할 핵심 품질 검증 항목입니다.

65 테스트 가독성 및 구조

테스트 명명 규칙

Method_Scenario_Expected 형식으로 일관되게 작성되었는가?

AAA 패턴 적용

Arrange-Act-Assert 구조가 명확히 분리되어 있는가?

중복 코드 제거

설정 코드가 헬퍼 메서드나 Fixture로 적절히 추출되었는가?

테스트 품질 및 커버리지

경계값/예외 케이스

경계 조건, 널값, 최대/최소값에 대한 테스트가 있는가?

적절한 Mock 사용

의존성이 올바르게 격리되고 행위 검증이 적절한가?

단일 책임 테스트

하나의 테스트가 하나의 시나리오만 검증하는가?

리뷰 프로세스

검증 로직이 의도를 정확히 반영하는지, 깨지기 쉬운 테스트가 아닌지 확인하세요.

데모/평가 루브릭

평가 항목	우수 (5점)	양호 (3점)	개선 필요 (1점)
기능 적합성30%	모든 요구사항을 완벽히 구현 엣지 케이스까지 고려	핵심 요구사항 대부분 구현 일부 엣지케이스 누락	기본 기능만 구현 중요 요구사항 일부 미구현
테스트 코드 품질40%	높은 커버리지(80%+) 모든 경로/예외 테스트	적절한 커버리지(60-80%) 주요 경로 테스트	낮은 커버리지(60% 미만) 핵심 기능 테스트 부재
CI/CD 자동화20%	완전한 자동화 구축 PR 품질 게이트 포함	기본 CI 파이프라인 기본 자동 테스트	불완전한 CI 구성 수동 테스트 의존
협업 및 발표10%	체계적 역할분담/코드리뷰 전문적 발표/데모	적절한 역할분담 이해가능한 발표	불명확한 역할분담 미흡한 발표준비

i 종합 평가

90점↑: A+, 80-89: A, 70-79: B, 60-69: C, 60점↓: D

가산점: TDD 완벽적용(+5), 혁신도구(+3), 코드품질(+2)

회고/배운 점

프로젝트 회고와 지속적 개선을 위한 핵심 방법론입니다. 효과적인 피드백 사이클을 구축하세요.

잘된 점 (What went well)

성공적인 테스트 자동화, 품질 향상 등 긍정적 측면을 식별하고 성공 요인을 분석하세요.

개선점 (To improve)

테스트 구현 속도, 커버리지 부족 등 개선이 필요한 영역을 성장 기회로 접근하세요.

액션 아이템 (Action items)

구체적이고 실행 가능한 개선 계획을 수립하고 담당자와 기한을 명확히 지정하세요.

지속 가능한 테스트 전략

테스트 자산의 지속적 개선, 지식 공유, 코드 리뷰 강화, 테스트 문화 정착을 추진하세요.

효과적인 회고 진행법

정기적인(2-4주) 회고, 심리적 안전감이 보장된 환경, '4Ls' 방법 또는 '스타-스톱-계속' 등 다양한 기법을 활용하세요. 식별된 문제점이 실질적인 개선으로 이어지는 액션 아이템을 도출하는 것이 핵심입니다.



Part 10

부록

FAQ, 참고 자료, 용어집,
추가 연습문제 및 학습 리소스

FAQ

i 자주 묻는 질문

- 💡 단위테스트와 관련하여 자주 질문되는 내용입니다.
- ✓ 더 많은 질문은 과정 Q&A 시간에 문의하세요.

? 핵심 질문과 답변

Q1

코드 커버리지 목표를 어느 수준으로 잡는 것이 좋을까요?

핵심 비즈니스 로직은 80-90%, UI/인프라 코드는 50-70% 정도가 적절합니다. 단순 수치보다 핵심 시나리오 검증이 중요합니다.

Q2

Mocking을 과도하게 사용하는 것이 좋을까요?

Mock은 외부 의존성 분리에 유용하나, 과도한 사용은 테스트를 구현에 강결합시킵니다. 통합 테스트와 균형을 맞추세요.

Q3

레거시 코드에 어떻게 테스트를 추가해야 할까요?

현재 동작을 특성화 테스트로 문서화하고, 점진적으로 의존성 주입이 가능한 Seam을 찾아 리팩토링하세요. 변경 필요 부분부터 우선 적용하세요.

i 추가 질문이 있으신가요?

실습 시간이나 팀별 과제 진행 시 코치에게 문의하세요. 단위 테스트는 지속적인 연습으로 습득하는 기술입니다.

참고 자료

공식 문서

.NET 단위 테스트 모범 사례

learn.microsoft.com/ko-kr/dotnet/core/testing

Microsoft 공식 단위 테스트 가이드

NUnit/xUnit 공식 문서

docs.nunit.org / xunit.net/docs

테스트 프레임워크 레퍼런스

테스트 도구

Moq 라이브러리

github.com/moq/moq4

인기 C# 목킹 프레임워크

ReportGenerator

reportgenerator.io

코드 커버리지 시각화

CI/CD 자동화

GitHub Actions - .NET

docs.github.com/ko/actions

GitHub CI/CD 파이프라인 설정

Azure DevOps

azure.microsoft.com/ko-kr/products/devops

MS 클라우드 CI/CD 솔루션

학습 자료

The Art of Unit Testing (Roy Osherove)

단위 테스트 원칙과 패턴

교육 과정 예제 코드

github.com/SDC-SW/unit-test-examples

실습 예제 리포지토리

추가 학습 팁

실제 프로젝트에 적용하여 점진적으로 학습하세요. 코드 리뷰와 페어 프로그래밍으로 테스트 기법을 공유하는 것이 효과적입니다.

용어집 및 추가 연습

■ 주요 용어 정리

AAA 패턴

Arrange(준비), Act(실행), Assert(검증) 단계로 테스트를 구조화하는 패턴

Fixture

테스트에 필요한 객체나 상태의 고정된 기준점, 테스트 간 공유되는 설정

코드 커버리지

테스트가 프로덕션 코드를 실행하는 비율. 라인, 브랜치, 함수 커버리지

TDD

Test-Driven Development. 테스트를 먼저 작성한 후 코드를 구현하는 방법론

Test Double

실제 객체 대신 사용하는 테스트용 객체. Mock, Stub, Fake, Spy 등

FIRST 원칙

Fast, Independent, Repeatable, Self-validating, Timely의 단위 테스트 특성

☰ 추가 연습 과제

레거시 코드에 테스트 추가

테스트가 없는 모듈을 찾아 단위 테스트를 작성하세요:

1. 의존성 분리로 테스트 가능하게 리팩토링
2. 핵심 기능 테스트 케이스 작성
3. 경계 조건과 예외 처리 테스트
4. 코드 커버리지 80% 이상 달성

TDD로 새 기능 개발

다음 요구사항을 TDD 방식으로 구현하세요:

- 문자열 파서 (CSV, JSON)
- 검색 알고리즘 (이진 검색, 정렬)
- API 요청 처리기 (비동기)

모든 단계에서 Red-Green-Refactor 사이클 적용