

lab-01-tdd-with-copilot

- [Lab 1: Test-Driven Development with GitHub Copilot](#)
 - [Overview](#)
 - [Prerequisites](#)
 - [Step 1: Create Interface First \(Design Phase\)](#)
 - [1.1 Open Copilot Chat](#)
 - [1.2 Request Interface Generation](#)
 - [1.3 Review Generated Interface](#)
 - [1.4 Verify Design](#)
 - [Step 2: Write Tests FIRST \(Red Phase\)](#)
 - [2.1 Request Test Generation](#)
 - [2.2 Review Test Structure](#)
 - [2.3 Example Test \(SendEmailNotificationAsyncTests.cs\)](#)
 - [2.4 Run Tests \(Expect Failure - RED\)](#)
 - [2.5 Reflect on Test Design](#)
 - [Step 3: Implement Code \(Green Phase\)](#)
 - [3.1 Request Implementation](#)
 - [3.2 Review Generated Implementation](#)
 - [3.3 Verify Code Quality](#)
 - [3.4 Run Tests \(Expect Success - GREEN\)](#)
 - [Step 4: Observe & Reflect \(Refactor Phase\)](#)
 - [4.1 Review Architecture](#)
 - [4.2 Review Test Quality](#)
 - [4.3 Ask Copilot for Improvements](#)
 - [4.4 Optional Refactoring Exercise](#)
 - [Key Learning Points](#)
 - [TDD Benefits You Experienced](#)
 - [Copilot Instructions Impact](#)
 - [Common TDD Mistakes \(Avoid These!\)](#)
 - [Extension Exercises \(If Time Permits\)](#)
 - [Exercise 1: Add Email Validation](#)
 - [Exercise 2: Add OpenTelemetry Tracing](#)
 - [Exercise 3: Add Batch Notifications](#)
 - [Success Criteria](#)
 - [Troubleshooting](#)
 - [Tests Won't Compile](#)
 - [Tests Pass Immediately](#)
 - [Copilot Not Following Conventions](#)
 - [FakeItEasy Not Working](#)
 - [Next Steps](#)
 - [Additional Resources](#)

Lab 1: Test-Driven Development with GitHub Copilot

Duration: 30 minutes

Learning Objectives:

- Master the Red-Green-Refactor TDD cycle with AI assistance

- Use Copilot to generate tests before implementation
 - Apply repository Copilot Instructions for consistent code quality
 - Understand how TDD enforces better design decisions
-

Overview

In this lab, you'll create a `NotificationService` that sends task notifications via email and SMS. You'll follow strict Test-Driven Development (TDD) practices:

1. **Design** - Create the interface first
2. **Red** - Write failing tests
3. **Green** - Implement code to pass tests
4. **Refactor** - Improve and reflect

Why TDD? Writing tests first forces you to think about your API design, ensures testability, and provides living documentation of behavior.

Prerequisites

- Repository cloned and `main` branch checked out
 - VS Code open with GitHub Copilot enabled
 - `.github/copilot-instructions.md` automatically loaded
 - Initial build successful: `dotnet build && dotnet test`
-

Step 1: Create Interface First (Design Phase)

1.1 Open Copilot Chat

- Press `Ctrl+Alt+I` (Windows/Linux) or `Cmd+Shift+I` (Mac)
- This opens the Copilot Chat panel

1.2 Request Interface Generation

In the chat panel, enter:

Create an `INotificationService` interface in the Application layer for sending email and SMS notifications about tasks. Include methods for both individual and combined notifications.

1.3 Review Generated Interface

Copilot should generate something like:

```
namespace TaskManager.Application.Services;  
  
public interface INotificationService  
{
```

```

    Task SendEmailNotificationAsync(string recipient, string subject, string
        message, CancellationToken cancellationToken = default);

    Task SendSmsNotificationAsync(string phoneNumber, string message,
        CancellationToken cancellationToken = default);

    Task SendNotificationAsync(string recipient, string phoneNumber, string
        subject, string message, CancellationToken cancellationToken =
        default);
}


```

Expected Location: src/TaskManager.Application/Services/INotificationService.cs

1.4 Verify Design

Review the interface and ask yourself:

- Does it belong in the Application layer? (Yes - it's a service interface)
- Are method names descriptive and intention-revealing?
- Does it follow async/await patterns with CancellationToken?
- Is the API easy to use and understand?

If satisfied, accept the code. If not, refine your prompt.

Step 2: Write Tests FIRST (Red Phase)

Critical TDD Principle: Write tests BEFORE implementation. This is the "Red" phase - tests will fail because the implementation doesn't exist yet.

2.1 Request Test Generation

In Copilot Chat, enter:

Create xUnit tests for NotificationService in the pattern specified in .github/copilot-instructions.md. Organize tests by method with separate test classes. Use FakeItEasy for mocking ILogger. Test happy path and all guard clauses.

2.2 Review Test Structure

Copilot should create a folder structure like:

```

tests/TaskManager.UnitTests/Services/NotificationServiceTests/
└── SendEmailNotificationAsyncTests.cs
└── SendSmsNotificationAsyncTests.cs
└── SendNotificationAsyncTests.cs

```

Each test class should contain:

- Tests for the happy path (valid inputs)
- Tests for guard clauses (null/empty parameters)

- Descriptive test method names (e.g., `SendEmailNotificationAsync_WithValidInputs_SendsEmail`)
- `FakeItEasy` mocks for `ILogger<NotificationService>`
- Async test methods with proper assertions

2.3 Example Test (`SendEmailNotificationAsyncTests.cs`)

```

namespace TaskManager.UnitTests.Services.NotificationServiceTests;

public sealed class SendEmailNotificationAsyncTests
{
    private readonly ILogger<NotificationService> _logger;
    private readonly NotificationService _sut;

    public SendEmailNotificationAsyncTests()
    {
        _logger = A.Fake<ILogger<NotificationService>>();
        _sut = new NotificationService(_logger);
    }

    [Fact]
    public async Task SendEmailNotificationAsync_WithValidInputs_SendsEmail()
    {
        // Arrange
        const string recipient = "user@example.com";
        const string subject = "Task Update";
        const string message = "Your task has been updated";

        // Act
        await _sut.SendEmailNotificationAsync(recipient, subject, message);

        // Assert
        // Verify logging occurred (implementation detail we'll check)
        A.CallTo(_logger).Where(call =>
            call.Method.Name == "Log" &&
            call.GetArgument<LogLevel>(0) == LogLevel.Information)
            .MustHaveHappened();
    }

    [Theory]
    [InlineData(null)]
    [InlineData("")]
    [InlineData(" ")]
    public async Task
        SendEmailNotificationAsync_WithInvalidRecipient_ThrowsArgumentException(string
            invalidRecipient)
    {
        // Arrange
        const string subject = "Test";
    }
}

```

```

    const string message = "Test message";

    // Act & Assert
    await Assert.ThrowsAsync<ArgumentException>(() =>
        _sut.SendEmailNotificationAsync(invalidRecipient, subject,
            message));
}

[Theory]
[InlineData(null)]
[InlineData("")]
[InlineData(" ")]
public async Task
    SendEmailNotificationAsync_WithInvalidSubject_ThrowsArgumentException(string
        invalidSubject)
{
    // Arrange
    const string recipient = "user@example.com";
    const string message = "Test message";

    // Act & Assert
    await Assert.ThrowsAsync<ArgumentException>(() =>
        _sut.SendEmailNotificationAsync(recipient, invalidSubject,
            message));
}

[Theory]
[InlineData(null)]
[InlineData("")]
[InlineData(" ")]
public async Task
    SendEmailNotificationAsync_WithInvalidMessage_ThrowsArgumentException(string
        invalidMessage)
{
    // Arrange
    const string recipient = "user@example.com";
    const string subject = "Test";

    // Act & Assert
    await Assert.ThrowsAsync<ArgumentException>(() =>
        _sut.SendEmailNotificationAsync(recipient, subject,
            invalidMessage));
}
}

```

2.4 Run Tests (Expect Failure - RED)

In the terminal, run:

```
dotnet test
```

Expected Result: ✗ Tests FAIL

You should see errors like:

```
error CS0246: The type or namespace name 'NotificationService' could not be found
```

This is GOOD! You're in the "Red" phase of TDD. The tests define what you need to build.

2.5 Reflect on Test Design

Before implementing, review:

- Do test names clearly describe behavior?
 - Are guard clause tests comprehensive?
 - Is the happy path covered?
 - Are tests organized by method?
-

Step 3: Implement Code (Green Phase)

Goal: Write the MINIMUM code needed to make tests pass. No more, no less.

3.1 Request Implementation

In Copilot Chat, enter:

Implement NotificationService that passes all the tests. Follow the coding style in .github/copilot-instructions.md: sealed class, file-scoped namespace, ILogger dependency injection, async/await, guard clauses with nameof.

3.2 Review Generated Implementation

Copilot should generate src/TaskManager.Application/Services/NotificationService.cs:

```
namespace TaskManager.Application.Services;

public sealed class NotificationService : INotificationService
{
    private readonly ILogger<NotificationService> _logger;

    public NotificationService(ILogger<NotificationService> logger)
    {
        _logger = logger ?? throw new ArgumentNullException(nameof(logger));
    }

    public async Task SendEmailNotificationAsync(
        string recipient,
        string subject,
```

```
        string message,
        CancellationToken cancellationToken = default)
    {
        if (string.IsNullOrWhiteSpace(recipient))
            throw new ArgumentException("Recipient cannot be null or empty",
                nameof(recipient));

        if (string.IsNullOrWhiteSpace(subject))
            throw new ArgumentException("Subject cannot be null or empty",
                nameof(subject));

        if (string.IsNullOrWhiteSpace(message))
            throw new ArgumentException("Message cannot be null or empty",
                nameof(message));

        _logger.LogInformation(
            "Sending email notification to {Recipient} with subject
            {Subject}",
            recipient,
            subject);

        // Simulate email sending
        await Task.Delay(100, cancellationToken);

        _logger.LogInformation(
            "Email notification sent successfully to {Recipient}",
            recipient);
    }

    public async Task SendSmsNotificationAsync(
        string phoneNumber,
        string message,
        CancellationToken cancellationToken = default)
    {
        if (string.IsNullOrWhiteSpace(phoneNumber))
            throw new ArgumentException("Phone number cannot be null or
empty", nameof(phoneNumber));

        if (string.IsNullOrWhiteSpace(message))
            throw new ArgumentException("Message cannot be null or empty",
                nameof(message));

        _logger.LogInformation(
            "Sending SMS notification to {PhoneNumber}",
            phoneNumber);

        // Simulate SMS sending
        await Task.Delay(100, cancellationToken);
    }
}
```

```

        _logger.LogInformation(
            "SMS notification sent successfully to {PhoneNumber}",
            phoneNumber);
    }

    public async Task SendNotificationAsync(
        string recipient,
        string phoneNumber,
        string subject,
        string message,
        CancellationToken cancellationToken = default)
    {
        if (string.IsNullOrWhiteSpace(recipient))
            throw new ArgumentException("Recipient cannot be null or empty",
                nameof(recipient));

        if (string.IsNullOrWhiteSpace(phoneNumber))
            throw new ArgumentException("Phone number cannot be null or
empty", nameof(phoneNumber));

        if (string.IsNullOrWhiteSpace(subject))
            throw new ArgumentException("Subject cannot be null or empty",
                nameof(subject));

        if (string.IsNullOrWhiteSpace(message))
            throw new ArgumentException("Message cannot be null or empty",
                nameof(message));

        _logger.LogInformation(
            "Sending combined notification to email {Recipient} and phone
{PhoneNumber}",
            recipient,
            phoneNumber);

        await SendEmailNotificationAsync(recipient, subject, message,
            cancellationToken);
        await SendSmsNotificationAsync(phoneNumber, message,
            cancellationToken);

        _logger.LogInformation(
            "Combined notification sent successfully");
    }
}

```

3.3 Verify Code Quality

Check that the implementation follows all conventions:

- **sealed class** - Class cannot be inherited (defensive design)

- **File-scoped namespace** - namespace TaskManager.Application.Services;
- **Constructor validation** - logger ?? throw new ArgumentNullException(nameof(logger))
- **Guard clauses** - All parameters validated at method start
- **nameof() operator** - Used in all exceptions for refactoring safety
- **Async/await** - All methods properly async with CancellationToken
- **Structured logging** - Parameters passed to logger, not string interpolation
- **No else statements** - Guard clauses enable "fail fast" pattern
- **Single responsibility** - Class only handles notifications

3.4 Run Tests (Expect Success - GREEN)

In the terminal, run:

```
dotnet test
```

Expected Result: Tests PASS

You should see:

```
Passed!  - Failed:      0, Passed:     12, Skipped:      0, Total:     12
```

Congratulations! You've completed the Red-Green cycle.

Step 4: Observe & Reflect (Refactor Phase)

Goal: Improve code quality without changing behavior. Tests should still pass.

4.1 Review Architecture

Ask yourself:

- **Layer Separation:** Is NotificationService correctly in the Application layer?
 - Yes - it's a use case/service, not domain logic or infrastructure
- **Dependencies:** Does it only depend on ILogger (infrastructure concern)?
 - Yes - clean dependency injection
- **Domain Logic:** Is there any domain logic here?
 - No - this is pure application service orchestration

4.2 Review Test Quality

Ask yourself:

- **Test Organization:** Are tests organized by method in separate files?
- **Descriptive Names:** Can you understand behavior just by reading test names?
- **Test Coverage:** Are all edge cases covered (null, empty, whitespace)?
- **Test Independence:** Does each test run independently?

4.3 Ask Copilot for Improvements

Reusable Prompt:

Use the /check slash command in Copilot Chat to get code review and improvement suggestions:

```
/check Review the NotificationService implementation and tests. Are there any improvements we could make while keeping the same behavior?
```

Copilot might suggest:

- **Extract validation logic** into a helper method (reduce duplication)
- **Add more specific exception types** (e.g., InvalidEmailException)
- **Add integration tests** for actual email/SMS providers
- **Add telemetry/tracing** with OpenTelemetry (workshop bonus!)

4.4 Optional Refactoring Exercise

If time permits, try extracting parameter validation:

```
private static void ValidateParameter(string value, string parameterName)
{
    if (string.IsNullOrWhiteSpace(value))
        throw new ArgumentException($"{parameterName} cannot be null or empty", parameterName);
}
```

Then refactor methods to use:

```
ValidateParameter(recipient, nameof(recipient));
ValidateParameter(subject, nameof(subject));
ValidateParameter(message, nameof(message));
```

Run tests again: dotnet test - Should still pass!

Key Learning Points

TDD Benefits You Experienced

1. **Design First:** Interface and tests forced you to think about the API before writing code
2. **Clear Requirements:** Tests document exactly what the service should do
3. **Confidence:** Every change is validated by tests
4. **Refactoring Safety:** Can improve code structure without fear of breaking behavior
5. **No Overengineering:** Only wrote code needed to pass tests

Copilot Instructions Impact

1. **Consistency:** All generated code follows the same conventions
2. **Quality:** Guard clauses, async/await, logging automatically included
3. **Best Practices:** Sealed classes, nameof, structured logging enforced
4. **Test Patterns:** xUnit + FakeItEasy patterns consistently applied

⚠ Common TDD Mistakes (Avoid These!)

1. ✗ Writing implementation before tests - You lose design feedback
 2. ✗ Writing tests after implementation - Tests tend to just verify existing code, not drive design
 3. ✗ Skipping the "Red" phase - You don't know if tests actually test anything
 4. ✗ Making tests pass by changing tests - Tests define requirements; don't cheat!
 5. ✗ Ignoring failing tests - Red → Green → Refactor, always in that order
-

Extension Exercises (If Time Permits)

Exercise 1: Add Email Validation

1. Write a test that verifies email format validation
2. Implement email validation in `SendEmailNotificationAsync`
3. Ensure tests pass

Exercise 2: Add OpenTelemetry Tracing

1. Research OpenTelemetry in the workshop instructions
2. Add activity tracing to notification methods
3. Write tests that verify traces are created

Exercise 3: Add Batch Notifications

1. Design an interface for `SendBatchNotificationsAsync`
 2. Write tests for batch sending (multiple recipients)
 3. Implement batch notification logic
-

Success Criteria

You've completed this lab successfully when:

- `INotificationService` interface created in Application layer
 - Test suite created with 12+ passing tests
 - `NotificationService` implementation follows all Copilot Instructions conventions
 - You followed Red-Green-Refactor cycle (saw tests fail, then pass)
 - Code is clean, readable, and well-organized
 - You understand why TDD leads to better design
-

Troubleshooting

Tests Won't Compile

Problem: `NotificationService` type not found

Solution: This is expected in the Red phase! Implement the service in Step 3.

Tests Pass Immediately

Problem: Tests pass even though no implementation exists

Solution: Your tests might be too lenient. Review test assertions.

Copilot Not Following Conventions

Problem: Generated code doesn't use sealed classes, nameof, etc.

Solution:

1. Verify `.github/copilot-instructions.md` exists in repo
2. Restart VS Code to reload instructions
3. Be explicit in prompts: "Follow `.github/copilot-instructions.md`"

FakeItEasy Not Working

Problem: Can't create fakes or verify calls

Solution:

1. Ensure using directive: `using FakeItEasy;`
 2. Check NuGet package is installed in test project
 3. Review `FakeItEasy` syntax in existing tests
-

Next Steps

Move on to [**Lab 2: Requirements → Backlog → Code**](#) where you'll:

- Convert user stories into backlog items with Copilot
 - Generate acceptance criteria
 - Build features from requirements
 - Practice the full development workflow
-

Additional Resources

- [xUnit Documentation](#)
- [FakeItEasy Documentation](#)
- [GitHub Copilot Documentation](#)
- [Clean Architecture in .NET](#)