My Personal Experience with TDD

its challenges and how to overcome them

JACOB GADE | Software Engineer at STRONGMINDS a Trifork Company

Table of Contents

Introduction	Page 3
The TDD Fundamentals	Page 4
Challenges in Adopting TDD	Page 6
Challenge 1: Where do I Even Begin	Page 11
Challenge 2: Generating Meaningful Test Cases	Page 17
Challenge 3: Testing Orchestrating Methods	Page 27
Challenge 4: Maintaining Tests During Refactoring	Page 37
Challenge 5: Breaking Down Complex Business Logic	Page 43
Challenge 6: Long-Term Test Maintenance	Page 67
Conclusion: Making TDD Work for You	Page 73
References	Page 77

Introduction

I used to hate writing tests, but now they're among my most powerful development tools. Let's backtrack a bit.

In today's software industry, with its focus on agile development, DevOps, and end-to-end team ownership, the development process often follows these activities:

- Prepare: Research the problem, understand its context and domain. Explore existing solutions and break down the problem into smaller actions.
- Implement: Build the solution by executing the actions identified during preparation.
- Test: Write tests to verify your solution works correctly, covering both happy paths and edge cases. Tests also demonstrate to others that your code functions as intended.
- Review: Submit your solution for peer review, incorporate feedback, and merge into the main branch.
- Deploy: Release the new version, including your feature, to production.
- Monitor: Track application telemetry to identify unforeseen issues, enabling rapid detection of bugs, performance impacts and necessary fixes.

Testing, particularly automated testing, is vital in software development. Tests ensure your solution meets requirements, reduce bugs, and build confidence in your code's correctness. When someone asks ”How do you know it works?”, you simply point to your tests. Tests also act as guardrails going forward. As the codebase evolves and grows with more features, tests can be run continuously to confirm that existing behavior keeps working as intended.

Despite these benefits, I used to dread writing tests. After implementing a feature and manually verifying it through the API or UI, I was eager to move on to the next challenge. Writing tests felt like an annoying chore, something I did only because senior developers insisted or my pragmatic inner voice demanded it. This is why TDD intrigued me when I first encountered it at university. Could it make unit testing more engaging and less of a burden?

The TDD Fundamentals

Now, this booklet is not about what TDD is, as I expect you to already have delved into it, but for the sake of establishing a common understanding, let's summarize its key aspects and benefits.

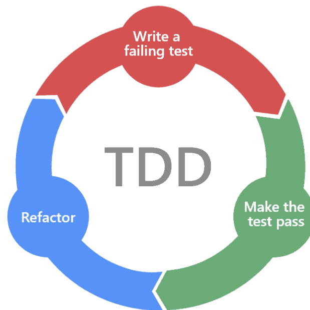


Figure 1: red-green-refactor cycle (*Why Test-Driven Development (TDD)* 2025)

Test-Driven Development (TDD) is a development process where you

write an automated test, verify it fails (confirming it tests the right thing), write minimal code to make it pass, then refactor for good design, generalization, and to eliminate duplication (Beck 2002). This flow, red-green-refactor (see Figure 1), means you write tests alongside production code, always starting with the test. This transforms testing from mere verification into a development and design tool. Importantly, TDD doesn't mean writing only unit tests, a common misconception in many TDD articles. Integration, component, and system tests remain vital. They answer different questions and catch errors unit tests miss. When TDD is applied at the integration or acceptance test level, it aligns closely with *Behaviour-Driven Development (BDD)* (North 2006), which expresses tests in a Given-When-Then (GWT) format analogous to Arrange-Act-Assert, framing behavior from a user's perspective. As we will see in Challenge 1, the outside-in approach to TDD shares this same user-centric focus.

Writing tests first forces you to test against requirements, not against an existing implementation. When writing tests after implementation, I'd often examine the code's methods and paths, then write tests to cover them. This essentially means testing the implementation rather than the requirements. This approach verifies the implementation works but doesn't validate it meets requirements. This is a crucial distinction that can lead to losing sight of original goals. Additional benefits I've discovered through TDD include:

- Using the API before implementing it provides early client perspective, leading to better design.
- Concurrent testing provides rapid feedback, preventing wasted effort on wrong approaches.
- Test difficulty reveals code smells early. Cumbersome setup often indicates Single Responsibility Principle (SRP) violations (Martin 2008) or inappropriate concrete dependencies.
- Tests serve as debugging entry points and sandboxes for exploring code and library behavior.

Despite these benefits, TDD remains controversial, with many claiming it hinders productivity. I believe this stems from people abandoning it too early, before overcoming the initial learning curve and challenges.

This booklet presents some of the challenges that I have met while adopting TDD, and what I have done to overcome them.

Challenges in Adopting TDD

Adopting TDD requires a fundamental mindset shift. Writing tests before implementation can feel counterintuitive. This paradigm change, combined with practical obstacles, makes TDD challenging to master. TDD guides often demonstrate simple examples like calculators with obvious input-output pairs. Real production codebases present far more complex challenges.

In the following sections, I'll demonstrate these challenges and solutions through a real-world example: building a logistics system for last-mile delivery. This example was chosen specifically because it contains the complexity you'll encounter in actual production code, including multiple interconnected services, business logic with various constraints, and a need for testable design. As we work through this case study, I'll highlight each challenge as it arises and show practical techniques to overcome it.

While I use OOP and C#/.NET, these principles apply across languages and paradigms.

Let's examine our example application and address the first challenge: figuring out where to begin.

Case Study: The Delivery Assignment Service

To better illustrate some of the challenges and their solutions, let's say we're building a logistics system for last-mile delivery companies. Multiple microservices constitute this system, each structured using the Onion Architecture (Palermo 2008). We're building the Delivery Assignment Service, which assigns deliveries to couriers.

This service should:

- Expose REST endpoints for delivery companies to register new deliveries
- Listen for events when couriers come online/go offline
- Publish events when a delivery is assigned to a courier
- Have business logic for matching deliveries to couriers based on factors like location, courier capacity, and delivery proximity clustering

The larger system also has services like:

- Courier Service (manages courier profiles, availability, real-time location)
- Delivery Tracking Service (tracks delivery status, ETAs)
- Customer Notification Service (sends updates to customers)
- Analytics Service (tracks delivery performance metrics)
- Route Optimization Service (plans optimal delivery routes)

Figure 2 shows how these services interact, with the highlighted event flow representing the delivery assignment process we'll be implementing. The REST endpoint for registering new deliveries already exists, however, we are tasked with implementing the first version of the delivery assignment feature. After design meetings, it is determined that the flow of delivery assignment goes like this:

1. A courier comes online (`CourierOnlineEvent` from Courier Service)
2. Packages are assigned based on
    - Courier's assigned warehouse
    - Package proximity clustering
    - Courier capacity optimization

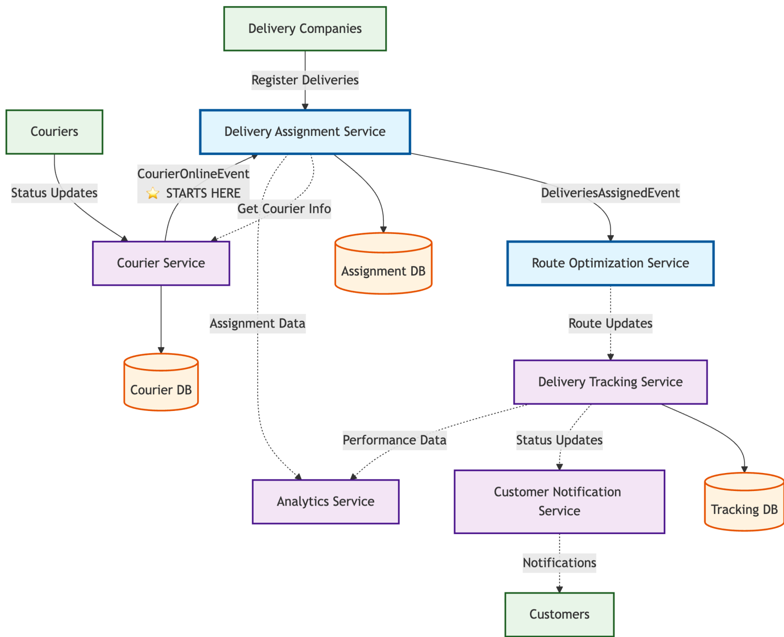


Figure 2: System architecture of the last-mile delivery system, illustrating how the services interact. The delivery assignment event flow is highlighted by making involved services blue and denoting the starting event with a star.

3. A `DeliveriesAssignedEvent` is published, triggering the Route Optimization Service to calculate the optimal route based on traffic etc.

Since we're using Clean Architecture (Onion Architecture) for our microservices, it's important to understand how our Delivery Assignment Service is structured internally:

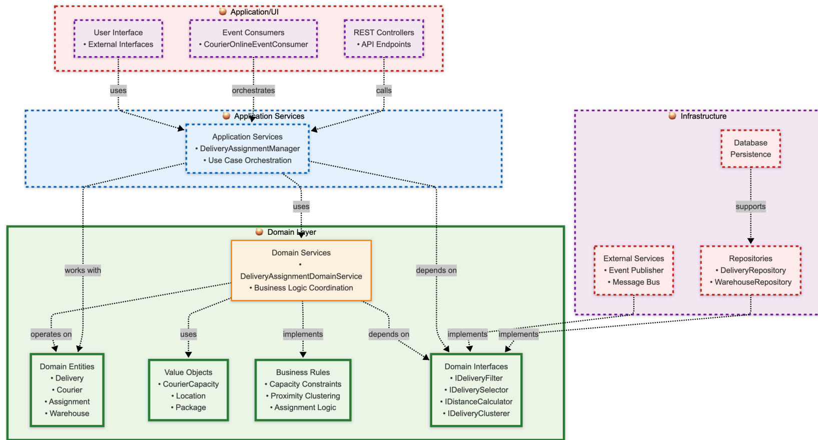


Figure 3: Showcasing how the Delivery Assignment Service is structured internally using clean or onion architecture.

This architectural structure will directly influence our TDD approach. Different layers present different testing challenges. These range from testing event consumers and REST controllers in the outer Application/UI layer that orchestrate work using services in the Application Services layer, to implementing complex business logic in the Domain core (Evans 2003). The dependency inversion principle means our domain interfaces define contracts that infrastructure components must implement, which affects how we set up mocks.

Now that we have our example application defined, and a new feature to implement, let's look at some of the TDD challenges we could face in the process.

Challenge 1

Where do I Even Begin?

We've been assigned the delivery assignment feature and determined our approach. But where do we start testing?

1.1 Solution

Two approaches exist:

1. Inside out
2. Outside in

Inside out: Start from the innermost layer (domain/business logic). This approach tackles core logic first, simplifying outer layer testing once you understand the underlying implementation.

Outside in: Start from the outermost layer (presentation/UI). This approach, which is my preference, drives design from the client's perspective. Since outer layers change more frequently (adding mobile apps, new UIs), designing for their needs creates better interfaces. This also mirrors how we typically read code.

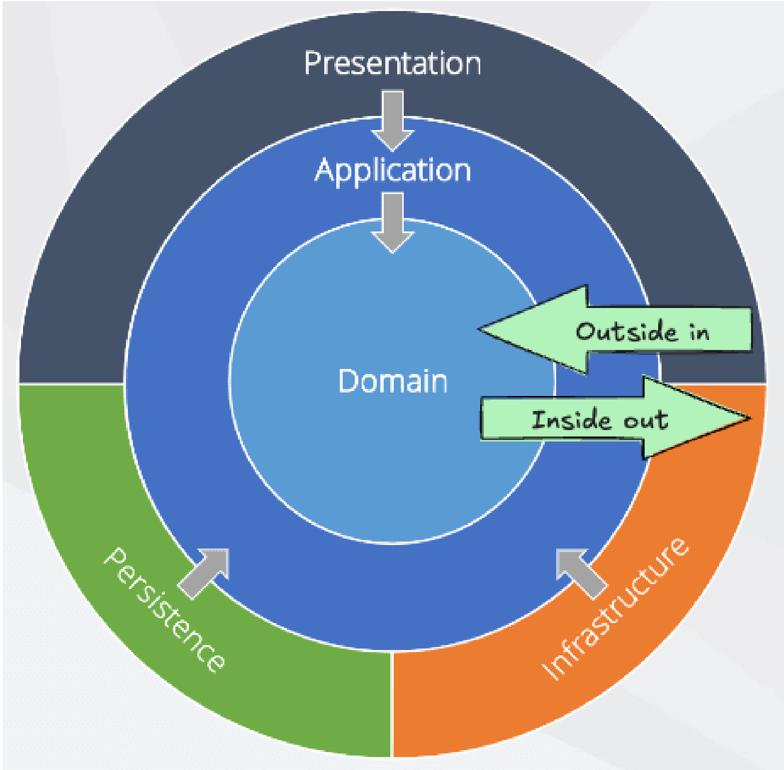


Figure 1.1: Inside out vs outside in testing

Note

Be cautious that outside-in doesn't overly shape your domain model around specific UI needs rather than core business rules. Keep domain concepts pure and refactor as needed.

1.2 Practical Example

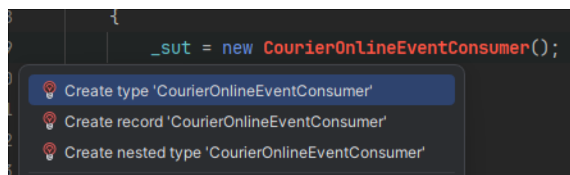
Assuming that we start from the outermost layer, let's start implementing our new feature.

From our planning/design session, we know that the entry point to our new delivery assignment feature is a new `CourierOnlineEvent` consumer.

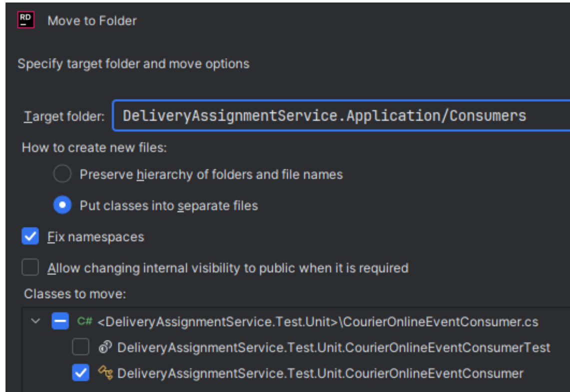
```
1  namespace DeliveryAssignmentService.Test.Unit;
2
3  public class CourierOnlineEventConsumerTest
4  {
5      private CourierOnlineEventConsumer _sut;
6
7      public CourierOnlineEventConsumerTest()
8      {
9          _sut = new CourierOnlineEventConsumer();
10     }
11
12     [Fact]
13     public void Test1()
14     {
15     }
16 }
```

Since `CourierOnlineEventConsumer` doesn't exist yet, we'll create it. Most IDEs make this easy with built-in refactoring tools:

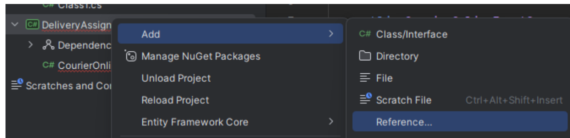
1. Create a class for `CourierOnlineEventConsumer`



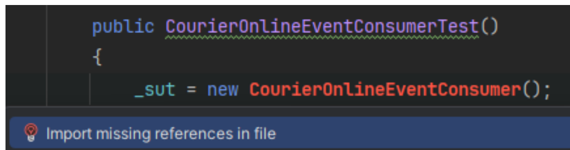
2. Move the new type



3. Add reference to assembly



4. Import class namespace



1.3 Key Takeaways: Where do I Even Begin?

The Challenge: Faced with a new feature request, it's unclear where to start writing tests when using TDD.

The Solution: Choose between Outside-In (starting from the API/UI layer) or Inside-Out (starting from the domain layer or inner-most/lowest layer) based on your needs.

Quick Reference:

- **Outside-In**: Better for driving design from user's perspective and creating readable interfaces
- **Inside-Out**: Better for focusing on core business logic first and understanding critical infrastructure early
- Start with the entry point for your feature (in our case, the `CourierOnlineEventConsumer`)
- Let the compiler guide you by writing the test first, then creating the classes to make it compile

Next up: Now that we have code that compiles and are ready to write our first tests, how do we come up with the right tests? What should we start with?

Challenge 2

Generating Meaningful Test Cases

Determining your first test and generating meaningful test cases can be surprisingly difficult. Without an existing implementation to examine, you must work solely from requirements and design. This constraint is actually beneficial because it forces requirement-based tests rather than implementation-based ones. You also end up shaping the interface through test usage.

2.1 Solution

One thing I like to do when using TDD, is maintaining a running list of test cases and concerns for the current method. Every idea or edge case gets noted for eventual implementation. You'll be surprised about how many test cases you come up with as you start implementing the solution and consider various scenarios and edge cases.

I create this list before writing any tests. Then I note down all the test cases I can come up with from the top of my head. These typically include test cases that assert the ”Happy Path”, meaning what do I expect

when everything works as expected. This list eliminates decision paralysis between tests and lets me order tests to support incremental development. Starting with simple tests creates fast feedback loops and natural progression. Each test builds on the previous, avoiding complex tests that require full implementation upfront.

When creating this list it is important to keep **Scope** in mind: What is it that we are trying to test? If it is a unit test then you're only testing that class and its behavior. I know this is a basic fact about unit testing, but I sometimes need to remind myself to stay focused whenever I start pondering further, outside the scope.

Let's make this test-list. The first cases that come to mind are:

```
Testing the Consume method of
    ↪ CourierOnlineEventConsumer

- It should call a DeliveryAssignmentService to
    ↪ assign packages to the given courier
- When the DeliveryAssignmentService successfully
    ↪  assigns packages, it should publish a
    ↪ DeliveriesAssignedEvent
```

These happy-path tests are a start. Here are three techniques for generating more comprehensive test cases:

Getting the thoughts flowing: Write in plain language what the class/method should do. This narrative reveals hidden test cases and clarifies the class's purpose.

Branching (paths): Identify all execution paths, including happy path, edge cases, and error scenarios. Writing pseudocode helps reveal these paths early.

Focusing on the input-output pairing: Focus on black-box testing. What outputs should given inputs produce? This approach tests behavior, not implementation, which aligns with requirements. It also allows for easy refactoring of internals.

Using the first technique we get:

> *The `CourierOnlineEventConsumer` consumes `CourierOnlineEvents` that contain the ID of the courier who got online. Using this ID the consumer will try to get the courier. If the courier does not exist, it should log an error and not attempt to assign deliveries...*

From just the first few sentences, we discover a new concept, the courier ID. This leads to new logic, i.e. that we should check to see if the courier exists or not. Let's update the list.

```
Testing the Consume method of
    ↪ CourierOnlineEventConsumer

- (Updated) When receiving a CourierOnlineEvent
    ↪ with a valid courier ID, it should call a
    ↪ DeliveryAssignmentService to assign
    ↪ packages to the given courier
- When the DeliveryAssignmentService successfully
    ↪  assigns packages, it should publish a
    ↪ DeliveriesAssignedEvent
- (New) When receiving a CourierOnlineEvent with
    ↪ a valid courier ID, it should call the
    ↪ CourierService to get the courier in
    ↪ question.
- (NEW) When receiving a CourierOnlineEvent with
    ↪ an invalid/unknown courier ID, it should
    ↪ not attempt to assign packages.
```

Note: I skipped testing logging and made the first test more specific.

When focusing on input-output pairs, we should consider not just the main event flow but also our dependencies. What happens when `DeliveryAssignmentService` and `CourierService` receive valid or invalid inputs? What if `DeliveryAssignmentService` returns zero packages assigned? Let's capture these scenarios in our test list.

```
Testing the Consume method of
    ↪ CourierOnlineEventConsumer


...
```

```
- When the DeliveryAssignmentService returns
    ↪ empty assignments (no packages available),
    ↪ it should not publish anything
```

Another output could be an exception. What should happen if an exception occurs? In our case, we will avoid implementing our services to throw exceptions, and since we won't be using any third party libraries that do, we can skip it here.

We could discover more test cases by writing pseudocode, but we have enough to start. We'll add more tests as we discover them during implementation.

I've sorted these tests to build naturally on each other, progressing from simple to complex implementation.

```
Testing the Consume method of
    ↪ CourierOnlineEventConsumer

1. When receiving a CourierOnlineEvent with a
    ↪ valid courier ID, it should call the
    ↪ CourierService to get the courier in
    ↪ question.
2. When receiving a CourierOnlineEvent with a
    ↪ valid courier ID, it should call a
    ↪ DeliveryAssignmentService to assign
    ↪ packages to the given courier
3. When receiving a CourierOnlineEvent with an
    ↪ invalid/unknown courier ID, it should not
    ↪ attempt to assign packages.
4. When the DeliveryAssignmentService
    ↪ successfully assigns packages, it should
    ↪ publish a DeliveriesAssignedEvent
5. When the DeliveryAssignmentService returns
    ↪ empty assignments (no packages available),
    ↪ it should not publish anything
```

2.2 Practical Example

Here's our first test case, using Osherove's naming convention (Osherove 2005) (`Method_Scenario_ExpectedResult`) and the Arrange-Act-Assert (AAA) pattern, conventions we will use throughout:

```
using DeliveryAssignmentService.Application
  .Consumers;
using DeliveryAssignmentService
  .ApplicationServices;
using DeliveryAssignmentService.Domain;
using Moq;

namespace DeliveryAssignmentService.Test.Unit;

public class CourierOnlineEventConsumerTest
{
    private readonly CourierOnlineEventConsumer _sut;
    private readonly Mock<ICourierService>
        ↪ _courierServiceMock;

    public CourierOnlineEventConsumerTest()
    {
        _courierServiceMock = new Mock<ICourierService>();
        _sut = new CourierOnlineEventConsumer(
          _courierServiceMock.Object
        );
    }

    [Fact]
    public void Consume
      _WhenReceivingACourierOnlineEvent
      ForAValidCourier
      _TheCourierServiceShouldBeCalledToGetCourier()
    {
        // Arrange
        var courierId = Guid.NewGuid();
        var courierOnlineEvent = new CourierOnlineEvent
        {
            CourierId = courierId
        };
        // Act
        _sut.Consume(courierOnlineEvent);

        // Assert
        _courierServiceMock.Verify(x => x.GetCourierById(
            ↪ courierId), Times.Once);
    }
}
```

After making the code compile, we run the test and see it fail.

Consume_WhenReceivingACourierOnlineEventForAValidCourier_TheCourierServiceShouldBeCalledToGetCourier Failed

This confirms our test actually works. It fails when it should. The minimal implementation to make it pass is:

```
1  public class CourierOnlineEventConsumer : IConsumer<
       ↪ CourierOnlineEvent>
2  {
3      public void Consume(CourierOnlineEvent
           ↪ courierOnlineEvent)
4      {
5          throw new NotImplementedException();
6      }
7  }
8
9  -->
10
11 public class CourierOnlineEventConsumer(ICourierService
       ↪ courierService) : IConsumer<CourierOnlineEvent>
12 {
13     public void Consume(CourierOnlineEvent
           ↪ courierOnlineEvent)
14     {
15         courierService.GetCourierById(courierOnlineEvent.
               ↪ CourierId);
16     }
17 }
```

Running the test again, it passes.

Consume_WhenReceivingACourierOnlineEventForAValidCourier_TheCourierServiceShouldBeCalledToGetCourier Success

As we work through our test list, a pattern emerges. By the fourth test, we're writing something like this:

```
1  [Fact]
2  public void Consume
3     _WhenDeliveriesAreSuccessfullyAssigned
4     _AnEventShouldBePublished()
5  {
6      // Arrange
7      var courierId = Guid.NewGuid();
8      var courierOnlineEvent = new CourierOnlineEvent
9      {
10         CourierId = courierId
11     };
```

```
12
13          _courierServiceMock
14              .Setup(x => x.GetCourierById(It.Is<Guid>(guid =>
                    ↪ guid.Equals(courierId))))
15              .Returns(
16                  new Courier
17                  {
18                      Id = courierId,
19                      Status = CourierStatus.Online
20                  }
21              );
22
23          _deliveryAssignmentService
24              .Setup(x => x
25                  .AssignDeliveries(It.Is<Courier>(courier =>
                        ↪ courier.Id == courierId)))
26              .Returns(
27                  new DeliveryAssignment
28                  {
29                      Courier = new Courier
30                      {
31                          Id = courierId,
32                          Status = CourierStatus.Online
33                      },
34                      Date = DateOnly.FromDateTime(DateTime.Now),
35                      Deliveries = new List<Delivery>
36                      {
37                          new()
38                          {
39                              Id = Guid.NewGuid(),
40                              Address = "Address1",
41                              IsAssigned = true,
42                              Package = new Package
43                              {
44                                  Id = Guid.NewGuid(),
45                                  Weight = 1.5f,
46                                  Receiver = "Receiver1",
47                                  Sender = "Sender1"
48                              }
49                          },
50                          new()
51                          {
52                              Id = Guid.NewGuid(),
53                              Address = "Address2",
54                              IsAssigned = true,
55                              Package = new Package
56                              {
57                                  Id = Guid.NewGuid(),
58                                  Weight = 2.5f,
59                                  Receiver = "Receiver2",
```

```
60                              Sender = "Sender2"
61                          }
62                      }
63                  }
64              }
65          );
66
67      // Act
68      _sut.Consume(courierOnlineEvent);
69
70      // Assert
71      _courierServiceMock.Verify(x => x.GetCourierById(
            ↪ courierId), Times.Once);
72      _deliveryAssignmentService
73          .Verify(x => x
74              .AssignDeliveries(It.Is<Courier>(courier =>
                    ↪ courier.Id == courierId)),
75               Times.Once);
76      _eventPublisherMock
77          .Verify(x => x.PublishAsync(It.IsAny<
                ↪ DeliveriesAssignedEvent>()),
78          Times.Once);
79  }
```

2.3 Key Takeaways: Generating Meaningful Test Cases

The Challenge: Without existing implementation to guide you, it's difficult to identify what test cases to write.

The Solution: Create a test list using three techniques: Getting thoughts flowing, Branching (paths), and Input-Output pairing.

Quick Reference:

- **Create a test list before coding**: jot down test ideas as they come
- **Getting thoughts flowing**: Write in plain language what the method should do
- **Branching**: Identify different paths (happy path, edge cases, error cases)

- **Input-Output pairing**: Focus on black-box testing to determine what outputs for which inputs
- **Start with simple cases**: Start with simple cases and order them to build incrementally

Next Up: Looking at the code above, we see that the arrange part has become very long, taking up most of the test case code. This happens as we need to set up and arrange more and more dependencies of our consumer. Such long arrange parts discourage us from writing more tests as they are tiring and cumbersome to work with. They often pop up whenever we write tests for API controllers, services, consumers and the like, as they commonly use many dependencies.

The next challenge deals with exactly this problem.

Challenge 3

Testing Orchestrating Methods

Some methods are particularly cumbersome to test. I call these 'orchestrating methods'. They delegate work to dependencies, passing outputs between them to coordinate complex operations. You'll typically find them in outer layers like controllers, consumers, and services.

Testing these methods presents two problems: you must arrange all dependencies for each test case, and when you add new service calls, you must update the arrange section of existing tests. This slows development considerably. Additionally, arranging mocks requires knowing how the method uses each dependency. This couples your tests to the implementation. Refactoring can break tests even when behavior remains unchanged.

Our fourth test `Consume_WhenDeliveriesAreSuccessfullyAssigned_AnEventShouldBePublished` exemplifies this problem. The arrange part sets up two dependencies, asserting on three. Imagine that we needed to arrange 3-4 dependencies plus additional setup steps to set the system in a specific state. What makes this test even worse is the complexity of building the `DeliveryAssignment` object.

3.1 Solution

Luckily, several strategies can help manage orchestrating method tests.

3.1.1 Defaulting to the Happy Path

Usually your tests will either expect dependencies to ”just work”, or a single dependency will be arranged to return some error value. You can simplify by arranging mocks for the happy path in your setup method. For .NET xUnit (*xUnit.net* 2025) this is the test class’ constructor.

> **Note**
>
> If you need to share the same teardown code as well, make the test class IDisposable and implement the Dispose method.

The downside to this approach is that understanding one’s test requires knowledge of what’s happening in the constructor, meaning the complete test context isn’t contained within the test case itself. A solution to this is to name your test classes after the setup-context. E.g. for our `CourierOnlineEventConsumerTest` class we could add a nested class `InvalidCourier` which contains tests where the setup-context is arranged to match the fact that we are working with an invalid courier ID.

```
1  public class CourierOnlineEventConsumerTest
2  {
3      public class InvalidCourierId
4      {
5          private readonly CourierOnlineEventConsumer _sut;
6
7          public InvalidCourierId()
8          {
9              // Setup dependencies for invalid courier ID
10         }
11     }
12
13     // Add other nested classes for other setup-contexts
14 }
```

3.1.2 Factories or Builders for Mocks

Factory and builder patterns can make test writing easier and more readable, especially for complex mocks or test data.

Factories can return configured mocks or test data, with parameters for customization.

Builders offer more configuration flexibility than factories but require more implementation effort. They're worthwhile when you need highly customizable test objects across many scenarios.

3.1.3 Auto Fixture and Mock

Libraries and tools exist to dramatically simplify test data creation and mock setup. In .NET we have `AutoFixture` (*AutoFixture* 2025) and various Auto-Mockers for different libraries.

`AutoFixture` can create virtually any type of value without having to explicitly specify the exact value. This helps us keep the arrange part lean. It also helps us be more productive during TDD, as we don't have to worry about concrete values for test data unless we have to.

`Moq.AutoMock` (*Moq.AutoMock* 2025) is used to automate the process of creating and resolving mocks. This tool works as an auto mocking IoC container, that lets you ask for the system under test without worrying about its constructor, and changes to it. The IoC container will resolve all arguments automatically and insert the mocks.

3.1.4 Refactor the Method's Design

When orchestrating methods become too difficult to test, consider refactoring. You might extract strategies (using the strategy pattern) and test them independently. A single integration test can then verify they work together correctly.

3.2 Practical Example

Let's apply these strategies to make our test smaller and more readable, while preparing for the fifth test from our list.

First, using `AutoFixture` simplifies our test:

```
using AutoFixture;

...

public class CourierOnlineEventConsumerTest {

    private readonly Fixture _fixture = new();

    ...

    [Fact]
    public void Consume
    _WhenDeliveriesAreSuccessfullyAssigned
    _AnEventShouldBePublished()
    {
        // Arrange
        _fixture.Register(() => DateOnly.FromDateTime(new
            ↪ DateTime(2023, 1, 1)));
        var courier = _fixture.Create<Courier>();
        var courierOnlineEvent = _fixture.Build<
            ↪ CourierOnlineEvent>()
            .With(x => x.CourierId, courier.Id).Create();
        var deliveryAssignment = _fixture.Build<
            ↪ DeliveryAssignment>()
            .With(x => x.Courier, courier).Create();

        _courierServiceMock
            .Setup(x => x.GetCourierById(courier.Id)).
                ↪ Returns(courier);
        _deliveryAssignmentService
            .Setup(x => x.AssignDeliveries(courier)).
                ↪ Returns(deliveryAssignment);

        // Act
        _sut.Consume(courierOnlineEvent);

        // Assert - same as before
    }
}
```

In order for `Auto` `Fixture` to know how to create a valid `DateOnly`

struct, I had to register a creator. Such customization code could be placed in a base class (e.g., `WithAutoFixture`), that test classes can then inherit from to gain access to a pre-configured fixture. Using `AutoFixture` we have successfully made the arrange part much more concise, and easier to read.

Next, let's improve our mock creation with `Moq.AutoMock`:

```
using AutoFixture;
using Moq;
using Moq.AutoMock;

...

public class CourierOnlineEventConsumerTest
{
    private readonly CourierOnlineEventConsumer _sut;
    private readonly Fixture _fixture = new();
    private readonly AutoMocker _autoMocker = new
        ↪ CustomAutoMocker();

    public CourierOnlineEventConsumerTest()
    {
        _sut = _autoMocker.CreateInstance<
            ↪ CourierOnlineEventConsumer>();
    }

    ...

    [Fact]
    public void Consume
    _WhenDeliveriesAreSuccessfullyAssigned
    _AnEventShouldBePublished()
    {
        // Arrange
        _fixture.Register(() => DateOnly.FromDateTime(new
            ↪ DateTime(2023, 1, 1)));
        var courierOnlineEvent = _fixture.Create<
            ↪ CourierOnlineEvent>();

        // Act
        _sut.Consume(courierOnlineEvent);

        // Assert
        _autoMocker.GetMock<ICourierService>()
            .Verify(x => x.GetCourierById(
                ↪ courierOnlineEvent.CourierId),
            Times.Once);
        _autoMocker.GetMock<IDeliveryAssignmentManager>()
```

```
37                  .Verify(x => x.AssignDeliveries(It.Is<Courier>(
                        ↪ courierToAssign => courierToAssign.Id
                        ↪ == courierOnlineEvent.CourierId)),
38                  Times.Once);
39              _autoMocker.GetMock<IEventPublisher>()
40                  .Verify(x => x.PublishAsync(
41                      It.Is<DeliveriesAssignedEvent>(
                            ↪ assignedEvent => assignedEvent.
                            ↪ DeliveryAssignment.Courier.Id.
                            ↪ Equals(courierOnlineEvent.CourierId
                            ↪ ))),
42                  Times.Once);
43          }
44      }
45
46      // In CustomAutoMocker.cs - custom automocker to set
            ↪ default setup code for mocks
47      using AutoFixture;
48      using DeliveryAssignmentService.ApplicationService;
49      using DeliveryAssignmentService.Domain.Models;
50      using Moq;
51      using Moq.AutoMock;
52
53      namespace DeliveryAssignmentService.Test.Unit;
54
55      public class CustomAutoMocker : AutoMocker
56      {
57          private readonly Fixture _fixture = new();
58
59          public CustomAutoMocker()
60          {
61              SetupAutoFixture();
62              var courierServiceMock = GetMock<ICourierService>()
                    ↪ ;
63              courierServiceMock
64                  .Setup(x => x.GetCourierById(It.IsAny<Guid>()))
65                  .Returns<Guid>(id => _fixture.Build<Courier>()
66                      .With(c => c.Id, id)
67                      .Create());
68
69              var deliveryAssignmentManagerMock = GetMock<
                    ↪ IDeliveryAssignmentManager>();
70              deliveryAssignmentManagerMock
71                  .Setup(x => x.AssignDeliveries(It.IsAny<Courier
                        ↪ >()))
72                  .Returns<Courier>(courier => _fixture.Build<
                        ↪ DeliveryAssignment>()
73                      .With(da => da.Courier, courier)
74                      .Create());
75          }
```

```
76
77      private void SetupAutoFixture()
78      {
79          _fixture.Register(() => DateOnly.FromDateTime(
              ↪ DateTime.Now));
80      }
81  }
```

This dramatically improves our test. The arrange section drops from 58 to 2 lines. Following the suggestion of setting up mocks so that they convey to the happy-path, I added a `CustomAutoMocker`. This configures default happy-path behavior for all mocks, and `AutoFixture` simplifies the setup code itself.

Implementing the fifth and last test we get:

```
1   [Fact]
2   public void Consume
3     _WhenDeliveryAssignmentIsEmpty
4     _AnEventShouldNotBePublished()
5   {
6       // Arrange
7       _fixture.Register(() => DateOnly.FromDateTime(new
          ↪ DateTime(2023, 1, 1)));
8       var courierOnlineEvent = _fixture.Create<
          ↪ CourierOnlineEvent>();
9
10      _autoMocker.GetMock<IDeliveryAssignmentManager>()
11      .Setup(x => x.AssignDeliveries(It.IsAny<Courier>()))
12          .Returns(_fixture
13              .Build<DeliveryAssignment>()
14              .With(x => x.Deliveries, new List<Delivery>())
15              .Create());
16
17      // Act
18      _sut.Consume(courierOnlineEvent);
19
20      // Assert
21      _autoMocker.GetMock<ICourierService>()
22          .Verify(x => x.GetCourierById(courierOnlineEvent.
              ↪ CourierId), Times.Once);
23      _autoMocker.GetMock<IDeliveryAssignmentManager>()
24          .Verify(x => x.AssignDeliveries(It.Is<Courier>(
              ↪ courierToAssign => courierToAssign.Id ==
              ↪ courierOnlineEvent.CourierId)), Times.Once)
              ↪ ;
25      _autoMocker.GetMock<IEventPublisher>()
```

```
26            .Verify(x => x.PublishAsync(It.IsAny<
                  ↪ DeliveriesAssignedEvent>()), Times.Never);
27    }
```

Again, we have a concise arrange section. We only configure the mock that deviates from the happy path.

With logging added, our final `Consume` implementation looks like this:

```
1   public class CourierOnlineEventConsumer(
2       ILogger<CourierOnlineEventConsumer> logger,
3       ICourierService courierService,
4       IDeliveryAssignmentManager deliveryAssignmentManager,
5       IEventPublisher eventPublisher) : IConsumer<
            ↪ CourierOnlineEvent>
6   {
7
8       public void Consume(CourierOnlineEvent
            ↪ courierOnlineEvent)
9       {
10          var courier = courierService.GetCourierById(
                ↪ courierOnlineEvent.CourierId);
11          if (courier == null)
12          {
13              logger.LogWarning("Courier with id {CourierId}
                    ↪ not found",
14              courierOnlineEvent.CourierId);
15              return;
16          }
17
18          var deliveryAssignment = deliveryAssignmentManager
19              .AssignDeliveries(courier);
20
21          if (deliveryAssignment.Deliveries.Count == 0)
22          {
23              logger.LogInformation("No deliveries to assign
                    ↪ to courier {CourierId}", courier.Id);
24              return;
25          }
26
27          eventPublisher.PublishAsync(new
                ↪ DeliveriesAssignedEvent
28          {
29              DeliveryAssignment = deliveryAssignment
30          });
31      }
32  }
```

Our consumer now contains use case logic. It knows which services to call

and how to handle empty assignments. This violates Onion Architecture principles: use case logic belongs in the application service layer, while business rules (like handling zero deliveries) belong in the domain layer. This refactoring is exactly what drives the next challenge.

3.3 Key Takeaways: Testing Orchestrating Methods

The Challenge: Orchestrating methods require extensive mock setup that makes tests cumbersome, hard to read, and fragile.

The Solution: Use intelligent test fixtures, auto-mocking, and builder patterns to minimize setup code, make it flexible to change and improve maintainability. Consider redesigning the method's implementation if it becomes too cumbersome to test.

Quick Reference:

- **Default to happy path**: Configure mocks for success cases in test constructor or otherwise
- **Use AutoFixture**: Automatically generate test data without specifying exact values
- **Implement auto-mocking**: Use tools like Moq.AutoMocker (*Moq* 2025) to handle dependency injection automatically
- **Create builders/factories**: For complex test objects that need specific configurations
- **Consider test scope**: Not everything needs unit testing. Use integration tests for volatile orchestration code
- **Redesign method**: When the pain of testing the method becomes too great, consider if its design is really the issue.

Next Up: When we start to refactor our code, moving responsibilities around and changing the design, we run into the next hurdle of TDD. Tests start breaking. How can we mitigate this?

Challenge 4

Maintaining Tests During Refactoring

Refactoring production code, moving responsibilities between layers, redesigning interfaces, or extracting logic into new classes, is a natural part of TDD. But it often causes tests to break, even when the external behavior is unchanged. This can make refactoring feel costly and, over time, discourages it.

After moving use case and business logic to deeper layers, our redesigned Consume method might look like this:

```
1   public void Consume(CourierOnlineEvent courierOnlineEvent)
2   {
3       var courierId = courierOnlineEvent.CourierId;
4       var deliveryAssignmentResult =
5           deliveryAssignmentManager.AssignDeliveries(
                ↪ courierId);
6
7       if (deliveryAssignmentResult.IsFailure)
8       {
9           logger.LogError("Failed to assign deliveries to
                ↪ courier {CourierId}. Error:
10          {Error}", courierId, deliveryAssignmentResult.Error
                ↪ );
11          return;
12      }
```

```
13
14          eventPublisher.PublishAsync(new DeliveriesAssignedEvent
15          {
16              DeliveryAssignment = deliveryAssignmentResult.Value
17          });
18      }
```

The consumer no longer fetches couriers or checks for empty assignments. It simply attempts delivery assignment and either publishes an event on success or logs an error on failure. Note the use of a `Result` type to handle success/failure scenarios cleanly.

This refactoring will result in various compile time errors and most of our tests won't compile, e.g., because the `AssignDeliveries` does not take a `Courier` anymore but instead a `GUID`.

```
1   // Assert
2   _autoMocker.GetMock<IDeliveryAssignmentManager>()
3       .Verify(x =>
4           x.AssignDeliveries(It.Is<Courier>(courier =>
                ↪ courier.Id == courierId)),
5           Times.Once);
```

We must also update `CustomAutoMocker` to handle the new signatures using `Result` types. It should be noted that our auto mocking investments paid off. We did not need to change how our `CourierOnlineEventConsumer` is created or remove any mock creation code for the `ICourierService` as all of this is done automatically. Finally, we must remove assertions and setup for `CourierService` from our tests, including entire tests focused on courier fetching. This refactoring reduces our `Consume` tests from 5 to 3.

It's tempting to think we could have avoided this work by writing tests after implementation. However, this test maintenance isn't entirely negative. Let me explain why, then share strategies to minimize refactoring pain.

Breaking tests actually provide valuable feedback:

- They confirm we've actually changed something
- They prove our tests detect changes effectively

We can also let broken tests guide our refactoring process. For non-trivial changes, update tests incrementally to reflect new behavior. This essentially means using TDD to drive the refactoring itself. For major redesigns, consider implementing the new version in parallel with TDD, then switching over once it's complete.

4.1 Solution

Next are some strategies to minimize test maintenance during refactoring.

4.1.1 Test behavior not implementation

I mentioned this earlier for the challenge "Having a hard time coming up with tests" where I said to focus on the input-output pairs when trying to come up with good test cases. This rule also applies when we want tests that don't break during refactoring. As long as the contract is intact, we can change the internal implementation as much as we like. This works well for pure methods with clear inputs and outputs. However, orchestrating methods like `Consume` that return void and coordinate dependencies require testing some implementation details. From the outside, `Consume` 's contract remained unchanged, but internally we redesigned it completely. Since our tests verified internal interactions, they broke. The next strategy addresses this specific challenge.

4.1.2 Strategic Test Coverage

Test critical domain logic and complex algorithms thoroughly with unit tests. They benefit from the confidence and design feedback. For volatile orchestration code, consider integration or component tests instead. These test from a higher level without mocking every dependency of the system under test. This makes them more capable of proper black box testing, and therefore more resilient to internal refactoring.

4.1.3 Single Assertion Tests

When fixing our tests after refactoring, we had to remove `CourierService` assertions from every test. Only one test actually needed this assertion: ”When receiving a `CourierOnlineEvent` with a valid courier ID, it should call the `CourierService` to get the courier in question”. This single test would be enough to tell us if the `CourierService` was not called properly anymore. We should only assert what's relevant to each specific test. This clarifies test intentions and improves maintainability. While some advocate one assertion statement per test, I prefer asserting one concept per test, which might require multiple related assertions.

4.1.4 Single Responsibility Principle

Why do I mention this well known **SOLID** principle all of a sudden. Well the idea here is that if we adhere to the **SRP** we gain two benefits that will mitigate test code maintenance during refactoring.

- Focused classes lead to focused test suites. Only tests for the specific responsibility being changed will break.

- Focused test suites tend to be simpler or smaller, making them more maintainable.

To understand these benefits consider the opposite: a `CourierService` handling everything courier-related, including delivery assignment. This class would have many methods, many dependencies, and many tests. When one responsibility changes, all tests might break due to shared setup code. The impact cascades across unrelated areas. Following SRP contains changes to relevant tests only.

After implementing `DeliveryAssignmentManager`, we move to the core domain logic, delivery assignment. This brings us to our next challenge.

4.2 Key Takeaways: Maintaining Tests During Refactoring

The Challenge: Tests often break during refactoring even when behavior remains the same, making refactoring painful and time-consuming especially during TDD.

The Solution: Write tests that focus on behavior rather than implementation, and follow design principles that minimize test fragility.

Quick Reference:

- Test behavior, not implementation: Focus on input-output relationships, not internal method calls, i.e. black box over white box testing
- Strategic test coverage: Test critical domain logic thoroughly, orchestration logic less so
- Single assertion principle: Each test should verify one thing to minimize update scope
- Follow SRP: Smaller, focused classes mean fewer tests affected by changes
- Use refactoring as opportunity: Let breaking tests guide you through the refactoring process

Next Up: Using TDD for developing complex logic that you don't know the solution to yet can be difficult, but used correctly it can help you break it down into manageable steps and improve the result.

Challenge 5

Breaking Down Complex Business Logic

At this stage, the `AssignDeliveries` method of the application service `DeliveryAssignmentManager` might look like this:

```
1  public class DeliveryAssignmentManager(
2      ICourierService courierService,
3      IDeliveryRepository deliveryRepository,
4      IDeliveryAssignmentDomainService assignmentService,
5      ILogger<DeliveryAssignmentManager> logger)
6      : IDeliveryAssignmentManager
7  {
8
9      public Result<DeliveryAssignment> AssignDeliveries(Guid
           ↪ courierId)
10     {
11         // Get the courier
12         var courierResult = courierService.GetCourierById(
               ↪ courierId);
13         if (courierResult.IsFailure)
14         {
15             logger.LogWarning("Failed to assign deliveries:
                   ↪ Courier with ID {CourierId} not found.
                   ↪ Error: {Error}",
16                 courierId, courierResult.Error);
17
```

```
18              return Result.Failure<DeliveryAssignment>(
                    ↪ courierResult.Error);
19          }
20
21          var courier = courierResult.Value;
22          var warehouseId = courier.WarehouseId;
23          var deliveryCandidates = deliveryRepository.
                ↪ GetDeliveryCandidatesForCourier(
24              warehouseId,
25              DateTime.Today,
26              courier.MaxCapacity,
27              courier.CurrentLocation
28          );
29
30          if (deliveryCandidates.Count == 0)
31          {
32              logger.LogInformation("No suitable deliveries
                    ↪ available for courier {CourierId}",
                    ↪ courierId);
33
34              return Result.Success(new DeliveryAssignment
35              {
36                  Courier = courier,
37                  Date = DateOnly.FromDateTime(DateTime.Today
                        ↪ ),
38                  Deliveries = new List<Delivery>()
39              });
40          }
41
42          // Use domain service to assign deliveries to
                ↪ courier
43          var assignedDeliveries =
44              assignmentService.AssignDeliveriesToCourier(
                    ↪ courier, deliveryCandidates);
45
46          // Create delivery assignment
47          var deliveryAssignment = new DeliveryAssignment
48          {
49              Courier = courier,
50              Date = DateOnly.FromDateTime(DateTime.Today),
51              Deliveries = assignedDeliveries
52          };
53
54          // Update deliveries as assigned in a batch
                ↪ operation
55          if (assignedDeliveries.Count > 0)
56          {
57              deliveryRepository.MarkDeliveriesAsAssigned(
                    ↪ assignedDeliveries
58                  .Select(d => d.Id).ToList(), courier.Id);
```

```
59            }
60
61            logger.LogInformation("Successfully assigned {Count
                ↪ } deliveries to courier {CourierId}",
62              assignedDeliveries.Count, courierId);
63
64            return Result.Success(deliveryAssignment);
65        }
66    }
```

Comments have been added for readability. We see some of the logic that was previously in the `CourierOnlineEventConsumer` in this service instead, specifically the fetching of the courier. Similar to the `Consume` method, the `AssignDeliveries` is also an orchestrating method.

Note

Notice that `DeliveryAssignmentManager` decides when to mark deliveries as assigned (lines 55–63). One could argue this belongs to the `Delivery` domain entity itself, e.g., `delivery.MarkAsAssigned(courier)`, with the repository simply persisting the change. When application or domain services take over behaviour that logically belongs to domain objects, the result is an *anemic domain model* (Fowler 2003): objects that are little more than data bags. Whether this responsibility should live in the entity or the service is a worthwhile design conversation, and the right answer depends on your team's conventions and the complexity of the domain.

Next, we implement the `AssignDeliveriesToCourier` method of the `IDeliveryAssignmentDomainService`. This critical business logic requires thorough testing. It is also an area in which I am no expert, that is last mile delivery assignment algorithms, so we need to navigate a lot of unknowns and learn on the way. All of this makes it the perfect candidate to showcase one of the most important techniques in TDD: breaking down complex logic into manageable increments.

5.1 Solution

Before implementing `AssignDeliveriesToCourier`, let's first discuss test granularity in TDD. In *Test-Driven Development by Example* (Beck 2002), Beck describes adjustable "step sizes": you should be able to write tests that encourage one line of logic and a handful of refactorings, or tests that encourage 100 lines of logic and hours of refactoring. The former is driven by unit tests and the latter by application level tests. The idea is that you should be able to adjust your step size based on your circumstances. If you're highly confident in your ability to write correct code for a given method, extensive testing might not be necessary. However, when facing uncertainty, writing a lot of tests, each building on top of each other in small increments, makes sense. These small increments are what Beck calls "baby steps" (Beck 2002), and is what we will practice next.

Test step sizes work in two dimensions:

- Test-to-test progression: How much new logic each test drives
- Refactoring increments: how much code you change during refactoring between test runs

Both step sizes matter and should be adjusted according to your confidence level with the code you're writing.

5.2 Practical Example

5.2.1 Step 1: Setup Test Class

```
1  public class DeliveryAssignmentDomainServiceTest
2  {
3      private readonly DeliveryAssignmentDomainService _sut;
4      private readonly Fixture _fixture = new();
5      private readonly AutoMocker _autoMocker = new
           ↪ CustomAutoMocker();
6
7      public DeliveryAssignmentDomainServiceTest()
8      {
9          _sut = _autoMocker.CreateInstance<
               ↪ DeliveryAssignmentDomainService>();
```

```
10      }
11
12      // Tests...
13  }
```

5.2.2 Step 2: Write the first and simplest test

Note

In order to determine what tests to write and in what order, we could make a test list as demonstrated in Challenge 2.

The absolute simplest case that I can think of is, what happens when there are no deliveries to assign?

```
1   [Fact]
2   public void AssignDeliveriesToCourier
3     _WithNoAvailableDeliveries
4     _ReturnsEmptyList()
5   {
6       // Arrange
7       var courier = _fixture.Create<Courier>();
8       var availableDeliveries = new List<Delivery>();
9
10      // Act
11      var result = _sut.AssignDeliveriesToCourier(courier,
            ↪ availableDeliveries);
12
13      // Assert
14      Assert.Empty(result);
15  }
```

The simplest implementation that will make the test pass is:

```
1   public List<Delivery> AssignDeliveriesToCourier(Courier
        ↪ courier, List<Delivery> availableDeliveries)
2   {
3       return new List<Delivery>();
4   }
```

This passes by always returning an empty list. Obviously incorrect, but it drives us to the next test.

5.2.3 Step 3: Testing a single delivery

```
1  [Fact]
2  public void AssignDeliveriesToCourier
3    _WithOneAvailableDelivery
4    _ReturnsTheDelivery()
5  {
6      // Arrange
7      var courier = _fixture.Create<Courier>();
8      var delivery = _fixture.Create<Delivery>();
9      var availableDeliveries = new List<Delivery> { delivery
       ↪  };
10
11     // Act
12     var result = _sut.AssignDeliveriesToCourier(courier,
       ↪ availableDeliveries);
13
14     // Assert
15     Assert.Single(result);
16     Assert.Equal(delivery, result.First());
17 }
```

The simplest update to pass the test is:

```
1  public List<Delivery> AssignDeliveriesToCourier(Courier
     ↪ courier, List<Delivery> availableDeliveries)
2  {
3      if (availableDeliveries.Count == 0)
4      {
5          return new List<Delivery>();
6      }
7
8      return new List<Delivery> { availableDeliveries.First()
       ↪  };
9  }
```

We have taken two very small steps using two tests. Our implementation can now handle cases where there are either no, or just a single available delivery to assign. We haven't done any refactoring yet, and our implementation is clearly not the final, general implementation that can handle all types of input. But before refactoring, let's add one more test.

5.2.4 Step 4: Testing for multiple deliveries

```
1  [Fact]
2  public void AssignDeliveriesToCourier
3    _WithMultipleAvailableDeliveries
```

```
4     _ReturnsAllDeliveries()
5  {
6      // Arrange
7      var courier = _fixture.Create<Courier>();
8      var delivery1 = _fixture.Create<Delivery>();
9      var delivery2 = _fixture.Create<Delivery>();
10     var delivery3 = _fixture.Create<Delivery>();
11     var availableDeliveries = new List<Delivery>
12         { delivery1, delivery2, delivery3 };
13
14     // Act
15     var result = _sut.AssignDeliveriesToCourier(courier,
           ↪ availableDeliveries);
16
17     // Assert
18     Assert.Equal(3, result.Count);
19     Assert.Contains(delivery1, result);
20     Assert.Contains(delivery2, result);
21     Assert.Contains(delivery3, result);
22 }
```

Our implementation becomes,

```
1  public List<Delivery> AssignDeliveriesToCourier(Courier
       ↪ courier, List<Delivery> availableDeliveries)
2  {
3      if (availableDeliveries.Count > 1)
4      {
5          return availableDeliveries;
6      }
7
8      if (availableDeliveries.Count == 0)
9      {
10         return new List<Delivery>();
11     }
12
13     return new List<Delivery> { availableDeliveries.First()
           ↪ };
14 }
```

With tests for zero, one, and many deliveries, we can refactor toward a general solution. This technique of using multiple tests to reveal patterns is called triangulation (Beck 2002). The pattern emerges, leading to this general implementation.

```
1  public List<Delivery> AssignDeliveriesToCourier(Courier
       ↪ courier, List<Delivery> availableDeliveries)
2  {
```

```
3        return availableDeliveries;
4    }
```

Multiple tests before refactoring provide confidence that your general solution is correct. With only one test, you can't verify the generalization. Though this example's pattern was fairly obvious, the triangulation technique becomes invaluable for complex logic.

We will continue to use this triangulation strategy going forward.

5.2.5 Step 5: Triangulating on Courier Capacity

Let's add capacity constraints using triangulation, handling weight and volume separately.

```
1    [Fact]
2    public void AssignDeliveriesToCourier
3      _WhenCapacityCanOnlyFitOnePackage
4      _ReturnsThatOneDelivery()
5    {
6        // Arrange
7        var courier = _fixture.Build<Courier>()
8            .With(c => c.MaxCapacity, new CourierCapacity(1,
                 ↪ 0.1))
9            .Create(); // Capacity is 1 kilo, 0.1 cubic meter
10
11       var delivery1 = _fixture
12           .Build<Delivery>()
13               .With(x => x.Package,
14                   _fixture
15                       .Build<Package>()
16                       .With(x => x.Weight, 1)
17                       .With(x => x.Volume, 0.1)
18                       .Create())
19           .Create();
20       var delivery2 = _fixture.Create<Delivery>();
21       var availableDeliveries = new List<Delivery> {
             ↪ delivery1, delivery2 };
22
23       // Act
24       var result = _sut.AssignDeliveriesToCourier(courier,
             ↪ availableDeliveries);
25
26       // Assert
27       Assert.Single(result);
28   }
```

Our implementation becomes,

```
1   public List<Delivery> AssignDeliveriesToCourier(Courier
        ↪ courier, List<Delivery> availableDeliveries)
2   {
3       if (courier.MaxCapacity.MaxWeight.Equals(1))
4       {
5           return new List<Delivery> { availableDeliveries.
                ↪ First() };
6       }
7
8       return availableDeliveries;
9   }
```

We add one more test for a different capacity.

```
1   [Fact]
2   public void AssignDeliveriesToCourier
3     _WhenCapacityWeightCanFitTwoPackages
4     _ReturnsThemBoth()
5   {
6       // Arrange
7       var courier = _fixture.Build<Courier>()
8           .With(c => c.MaxCapacity, new CourierCapacity(2,
                ↪ 0.2))
9           .Create();
10      var delivery1 = _fixture
11          .Build<Delivery>()
12              .With(x => x.Package,
13                  _fixture
14                      .Build<Package>()
15                      .With(x => x.Weight, 1)
16                      .With(x => x.Volume, 0.1)
17                      .Create())
18          .Create();
19      var delivery2 = _fixture.Build<Delivery>()
20          .With(x => x.Package, delivery1.Package)
21          .Create();
22
23      var availableDeliveries = new List<Delivery> {
            ↪ delivery1, delivery2 };
24
25      // Act
26      var result = _sut.AssignDeliveriesToCourier(courier,
            ↪ availableDeliveries);
27
28      // Assert
29      Assert.Equal(2, result.Count);
30      Assert.Contains(delivery1, result);
31      Assert.Contains(delivery2, result);
32  }
```

This leads to,

```
1   public List<Delivery> AssignDeliveriesToCourier(Courier
        ↪ courier, List<Delivery> availableDeliveries)
2   {
3       if (courier.MaxCapacity.MaxWeight.Equals(2))
4       {
5           return new List<Delivery> { availableDeliveries[0],
6               availableDeliveries[1] };
7       }
8
9       if (courier.MaxCapacity.MaxWeight.Equals(1))
10      {
11          return new List<Delivery> { availableDeliveries.
                ↪ First() };
12      }
13
14      return availableDeliveries;
15  }
```

Writing the last test in our triangulation we get,

```
1   [Fact]
2   public void AssignDeliveriesToCourier
3     _WhenCapacityWeightCanFitNoPackages
4     _ReturnsNoDeliveries()
5   {
6       // Arrange
7       var courier = _fixture.Build<Courier>()
8           .With(c => c.MaxCapacity, new CourierCapacity(1,
                ↪ 0.1))
9           .Create();
10      var delivery1 = _fixture
11          .Build<Delivery>()
12          .With(x => x.Package,
13              _fixture
14                  .Build<Package>()
15                  .With(x => x.Weight, 10)
16                  .With(x => x.Volume, 1)
17                  .Create())
18          .Create();
19      var delivery2 = _fixture.Build<Delivery>()
20          .With(x => x.Package, delivery1.Package)
21          .Create();
22      var availableDeliveries = new List<Delivery> {
            ↪ delivery1, delivery2 };
23
24      // Act
25      var result = _sut.AssignDeliveriesToCourier(courier,
            ↪ availableDeliveries);
26
```

```
27        // Assert
28        Assert.Empty(result);
29    }
```

Following the zero-one-many pattern, we get,

```
1   public List<Delivery> AssignDeliveriesToCourier(Courier
        ↪ courier, List<Delivery> availableDeliveries)
2   {
3       if (availableDeliveries
4           .All(x => x.Package.Weight > courier.MaxCapacity.
                ↪ MaxWeight))
5       {
6           return new List<Delivery>();
7       }
8
9       if (courier.MaxCapacity.MaxWeight.Equals(2))
10      {
11          return new List<Delivery> { availableDeliveries[0],
12              availableDeliveries[1] };
13      }
14
15      if (courier.MaxCapacity.MaxWeight.Equals(1))
16      {
17          return new List<Delivery> { availableDeliveries.
                ↪ First() };
18      }
19
20      return availableDeliveries;
21  }
```

We then refactor to the general implementation, updating any tests that fail because AutoFixture generates random capacity values:

```
1   public List<Delivery> AssignDeliveriesToCourier(Courier
        ↪ courier, List<Delivery> availableDeliveries)
2   {
3       var result = new List<Delivery>();
4       var remainingWeight = courier.MaxCapacity.MaxWeight;
5
6       foreach (var delivery in availableDeliveries)
7       {
8           if (delivery.Package.Weight <= remainingWeight)
9           {
10              result.Add(delivery);
11              remainingWeight -= delivery.Package.Weight;
12          }
13      }
14
```

```
15        return result;
16    }
```

With these test helper methods,

```
1   private Courier GetCourierWithMaxCapacity(double maxWeight,
        ↪ double maxVolume)
2   {
3       return _fixture.Build<Courier>()
4           .With(c => c.MaxCapacity, new CourierCapacity(
                ↪ maxWeight, maxVolume))
5           .Create();
6   }
7
8   private Delivery GetDeliveryWithPackage(double weight,
        ↪ double volume)
9   {
10      return _fixture.Build<Delivery>()
11          .With(x => x.Package,
12              _fixture
13                  .Build<Package>()
14                  .With(x => x.Weight, weight)
15                  .With(x => x.Volume, volume)
16                  .Create())
17          .Create();
18  }
```

5.2.6 Step 6: Adding Volume Capacity

Volume constraints work similarly to weight, so one test suffices given our increased confidence.

```
1   [Fact]
2   public void AssignDeliveriesToCourier
3     _WhenDeliveriesExceedVolumeCapacity
4     _RespectVolumeLimit()
5   {
6       // Arrange
7       var courier = GetCourierWithMaxCapacity(10, 10);
8       var smallDelivery1 = GetDeliveryWithPackage(1, 1);
9       var smallDelivery2 = GetDeliveryWithPackage(2, 3);
10      var largeDelivery = GetDeliveryWithPackage(5, 8);
11      var availableDeliveries = new List<Delivery> {
            ↪ smallDelivery1, smallDelivery2,
12          largeDelivery };
13
14      // Act
15      var result = _sut.AssignDeliveriesToCourier(courier,
            ↪ availableDeliveries);
```

```
16
17        // Assert
18        Assert.Equal(2, result.Count);
19        Assert.Contains(smallDelivery1, result);
20        Assert.Contains(smallDelivery2, result);
21    }
```

This results in,

```
1   public List<Delivery> AssignDeliveriesToCourier(Courier
        ↪ courier, List<Delivery> availableDeliveries)
2   {
3       var result = new List<Delivery>();
4       var remainingWeight = courier.MaxCapacity.MaxWeight;
5       var remainingVolume = courier.MaxCapacity.MaxVolume;
6
7       foreach (var delivery in availableDeliveries)
8       {
9           if (delivery.Package.Weight <= remainingWeight
10              && delivery.Package.Volume <= remainingVolume)
11          {
12              result.Add(delivery);
13              remainingWeight -= delivery.Package.Weight;
14              remainingVolume -= delivery.Package.Volume;
15          }
16      }
17
18      return result;
19  }
```

5.2.7 Step 7: Proximity Optimization

Let's add proximity clustering. Deliveries close to each other take precedence over isolated ones. When multiple clusters exist, we prioritize the furthest clusters since closer deliveries can be assigned later. In other words, handle the most distant routes first, as nearby deliveries can always be picked up on a future assignment. (In the meantime, I've also added warehouse validation, ensuring couriers only deliver from their assigned warehouse).

Let's start with a simple proximity test.

```
1   [Fact]
2   public void AssignDeliveriesToCourier
3     _OptimizesForProximityWithinCapacityLimit()
4   {
```

```
5        // Arrange
6        var warehouseId = Guid.Empty; // Warehouse Repository
             ↪ mock set to return this
7                                      // warehouse ID per
                                           ↪ default
8        var courier = GetCourierWithCapacityAndLocation(
9            maxWeight: 10,
10           maxVolume: 10,
11           latitude: 0,
12           longitude: 0,
13           warehouseId: warehouseId
14       );
15       var closeDelivery1 = GetDeliveryWithPackageAndLocation(
16           weight: 5,
17           volume: 5,
18           latitude: 0.1,
19           longitude: 0.1,
20           warehouseId: warehouseId
21       );
22       var farDelivery1 =
23           GetDeliveryWithPackageAndLocation(5, 5, 100, 100,
                 ↪ warehouseId);
24       var farDelivery2 =
25           GetDeliveryWithPackageAndLocation(1, 1, 101, 101,
                 ↪ warehouseId);
26
27        var availableDeliveries =
28            new List<Delivery> { closeDelivery1, farDelivery1,
                 ↪  farDelivery2 };
29
30       // Act
31       var result = _sut.AssignDeliveriesToCourier(courier,
             ↪ availableDeliveries);
32
33       // Assert
34       Assert.Equal(2, result.Count);
35       Assert.Contains(farDelivery1, result);
36       Assert.Contains(farDelivery2, result);
37   }
```

Updating our implementation we get,

```
1   public List<Delivery> AssignDeliveriesToCourier(Courier
        ↪ courier, List<Delivery> availableDeliveries)
2   {
3        var result = new List<Delivery>();
4
5        var eligibleDeliveries = availableDeliveries
6            .Where(d => d.WarehouseId == courier.WarehouseId)
7            .ToList();
```

```
8
9       if (!eligibleDeliveries.Any())
10          return result;
11
12      var warehouse = warehouseRepository.GetWarehouseById(
            ↪ courier.WarehouseId);
13
14      var eligibleDeliveriesSorted = eligibleDeliveries
15          .OrderBy(d => CalculateDistance(warehouse.Location,
                ↪  d.Address.Location))
16          .ToList();
17
18      var remainingWeight = courier.MaxCapacity.MaxWeight;
19      var remainingVolume = courier.MaxCapacity.MaxVolume;
20
21      foreach (var delivery in eligibleDeliveriesSorted)
22      {
23
24          if (delivery.Package.Weight <= remainingWeight
25              && delivery.Package.Volume <= remainingVolume)
26          {
27              result.Add(delivery);
28              remainingWeight -= delivery.Package.Weight;
29              remainingVolume -= delivery.Package.Volume;
30          }
31      }
32
33      return result;
34  }
35
36  private double CalculateDistance(Location firstLocation,
        ↪ Location secondLocation)
37  {
38      return Math.Sqrt(
39          Math.Pow(firstLocation.Latitude - secondLocation.
                ↪ Latitude, 2)
40          + Math.Pow(firstLocation.Longitude - secondLocation
                ↪ .Longitude, 2));
41  }
42
43  // In CustomAutoMocker.cs
44  public CustomAutoMocker()
45  {
46  ...
47      var warehouseRepositoryMock = GetMock<
            ↪ IWarehouseRepository>();
48      warehouseRepositoryMock
49          .Setup(x => x.GetWarehouseById(It.IsAny<Guid>()))
50          .Returns(_fixture.Build<Warehouse>()
51              .With(x => x.Id, Guid.Empty)
```

```
52                  .With(x => x.Location, new Location(0, 0))
53                  .Create());
54  }
```

I've added `IWarehouseRepository` to get warehouse locations for distance calculations. The distance calculation uses simple 2D Euclidean distance, though production code would use actual route distances.

Continuing in this manner, we can add tests one by one, further developing our proximity implementation. With a couple of tests behind us we can also start to refactor the implementation to improve it. One idea I have is to use density-based spatial clustering of applications with noise (**DBSCAN**) (Ester et al. 1996) to identify clusters in our eligible deliveries dataset. From there we can prioritize the clusters by distance to the warehouse as we did above and then pick deliveries one by one from the cluster(s) depending on their proximity to each other and not only to the warehouse. To keep this somewhat brief, I won't show all the tests that got me there one by one, instead I will show their titles/names and the final implementation.

Two key tests shaped the algorithm:

1. Clusters with the furthest center point take precedence (better to handle many distant deliveries than one outlier)

2. Deliveries can span multiple clusters based on proximity (starting from the furthest delivery, we add the next closest regardless of cluster boundaries)

```
✓ AssignDeliveriesToCourier_PrioritizesCloserDeliveriesThanWhichClusterTheyBelongTo Success
✓ AssignDeliveriesToCourier_PrioritizesClusterWithCenterMostFarAway Success
```

The implementation now looks like this. The DBSCAN is implemented in the `GeoCluster` class using the `ClusterDeliveries` method (with a distance threshold of 2 km between items in a cluster).

```
1  public List<Delivery> AssignDeliveriesToCourier(Courier
       ↪ courier, List<Delivery> availableDeliveries)
2  {
3      var result = new List<Delivery>();
4
```

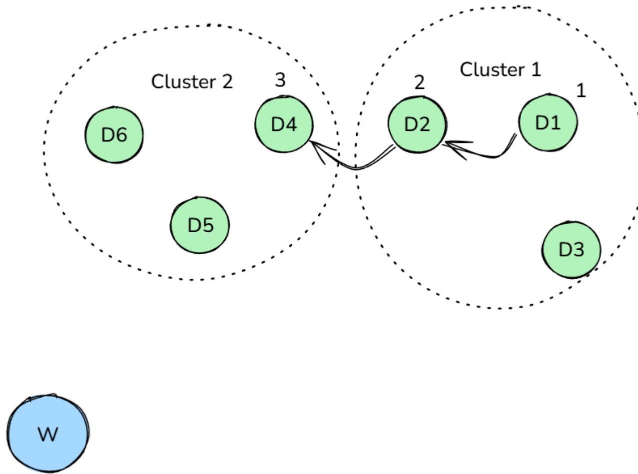


Figure 5.1: How deliveries are assigned couriers based on their relative distance to each other

```
5        var eligibleDeliveries = availableDeliveries
6            .Where(d => d.WarehouseId == courier.WarehouseId)
7            .ToList();
8
9        if (!eligibleDeliveries.Any())
10           return result;
11
12       var remainingWeight = courier.MaxCapacity.MaxWeight;
13       var remainingVolume = courier.MaxCapacity.MaxVolume;
14       var clusters = GeoCluster.ClusterDeliveries(
             ↪ eligibleDeliveries, 2);
15       var warehouse = warehouseRepository.GetWarehouseById(
             ↪ courier.WarehouseId);
16       var closestCluster =
17           clusters.MaxBy(x => CalculateDistance(warehouse.
                 ↪ Location, x.Centroid));
18       var currentDelivery = closestCluster?.Deliveries
19           .MaxBy(x => CalculateDistance(warehouse.Location, x
                 ↪ .Address.Location));
20
21       if (currentDelivery == null)
22           return result;
23
24       result.Add(currentDelivery);
25       eligibleDeliveries.Remove(currentDelivery);
```

```
26        remainingWeight -= currentDelivery.Package.Weight;
27        remainingVolume -= currentDelivery.Package.Volume;
28
29        while (remainingVolume > 0 && remainingWeight > 0)
30        {
31
32            var nextDelivery = eligibleDeliveries
33                .Where(d => d.Package.Weight <= remainingWeight
34                        && d.Package.Volume <= remainingVolume)
35                .MinBy(x => CalculateDistance(currentDelivery.
                    ↪ Address.Location,
36                        x.Address.Location));
37
38            if (nextDelivery == null)
39                break;
40
41            result.Add(nextDelivery);
42            remainingWeight -= nextDelivery.Package.Weight;
43            remainingVolume -= nextDelivery.Package.Volume;
44            currentDelivery = nextDelivery;
45            eligibleDeliveries.Remove(nextDelivery);
46        }
47
48        return result;
49    }
```

Our `AssignDeliveriesToCourier` method has become pretty hard to read and perhaps maintain for anyone else then ourselves, so as a last step, let's refactor it to make it more readable running our tests along the way to ensure that everything still works as intended.

5.2.8 Step 8: Final refactoring

There are two ways we could refactor the method. Either we simply extract private methods, or we, with the now added knowledge, break up the implementation into multiple classes, behind interfaces, to make it more modular and maintainable. Coming this far, let's go with the latter option as this also follows SRP which was advocated previously.

```
1   public class DeliveryAssignmentService(
2       IWarehouseRepository warehouseRepository,
3       IDeliveryFilter deliveryFilter,
4       IDistanceCalculator distanceCalculator,
5       IDeliveryClusterer deliveryClusterer,
6       IDeliverySelector deliverySelector)
7   {
```

```
8      public List<Delivery> AssignDeliveriesToCourier(Courier
           ↪  courier, List<Delivery> availableDeliveries)
9      {
10         var result = new List<Delivery>();
11
12         var eligibleDeliveries = deliveryFilter
13             .FilterEligibleDeliveries(availableDeliveries,
                   ↪ courier);
14
15         if (!eligibleDeliveries.Any())
16             return result;
17
18
19         var remainingVolume = courier.MaxCapacity.MaxVolume
               ↪ ;
20         var remainingWeight = courier.MaxCapacity.MaxWeight
               ↪ ;
21
22         var warehouse = warehouseRepository.
               ↪ GetWarehouseById(courier.WarehouseId);
23         var clusters = deliveryClusterer.ClusterDeliveries(
               ↪ eligibleDeliveries, 2);
24         var closestCluster = clusters.MaxBy(x =>
25             distanceCalculator.CalculateDistance(warehouse.
                   ↪ Location, x.Centroid));
26
27         var currentDelivery = closestCluster?.Deliveries.
               ↪ MaxBy(x =>
28             distanceCalculator.CalculateDistance(warehouse.
                   ↪ Location,
29                 x.Address.Location));
30
31         if (currentDelivery == null)
32             return result;
33
34         AddDeliveryToAssignment(result, currentDelivery,
               ↪ ref remainingWeight,
35             ref remainingVolume);
36         eligibleDeliveries.Remove(currentDelivery);
37
38         while (HasRemainingCapacity(remainingWeight,
               ↪ remainingVolume))
39         {
40             var nextDelivery = deliverySelector
41                 .SelectNextDelivery(
42                     eligibleDeliveries,
43                     currentDelivery.Address.Location,
44                     remainingWeight,
45                     remainingVolume
46                 );
```

```
47
48                if (nextDelivery == null)
49                    break;
50
51                AddDeliveryToAssignment(result, nextDelivery,
                    ↪ ref remainingWeight,
52                    ref remainingVolume);
53
54                currentDelivery = nextDelivery;
55                eligibleDeliveries.Remove(nextDelivery);
56            }
57
58            return result;
59        }
60
61        private void AddDeliveryToAssignment(List<Delivery>
            ↪ assignments, Delivery delivery, ref double
            ↪ remainingWeight, ref double remainingVolume) {
62            assignments.Add(delivery);
63            remainingWeight -= delivery.Package.Weight;
64            remainingVolume -= delivery.Package.Volume;
65        }
66
67        private bool HasRemainingCapacity(double
            ↪ remainingWeight, double remainingVolume) {
68            return remainingVolume > 0 && remainingWeight > 0;
69        }
70    }
71
72    // IDistanceCalculator.cs
73    public interface IDistanceCalculator
74    {
75        double CalculateDistance(Location first, Location
            ↪ second);
76    }
77
78    // DistanceCalculator.cs
79    public class EuclideanDistanceCalculator :
        ↪ IDistanceCalculator
80    {
81        public double CalculateDistance(Location first,
            ↪ Location second)
82        {
83            return Math.Sqrt(
84                Math.Pow(first.Latitude - second.Latitude, 2) +
85                Math.Pow(first.Longitude - second.Longitude, 2)
86            );
87        }
88    }
89
```

```
90  // IDeliveryFilter.cs
91  public interface IDeliveryFilter
92  {
93      List<Delivery> FilterEligibleDeliveries(IEnumerable<
        ↪ Delivery> deliveries, Courier courier);
94  }
95
96  // DeliveryFilter.cs
97  public class WarehouseDeliveryFilter : IDeliveryFilter
98  {
99      public List<Delivery> FilterEligibleDeliveries(
        ↪ IEnumerable<Delivery> deliveries, Courier
        ↪ courier)
100     {
101         return deliveries.
102             Where(d => d.WarehouseId == courier.WarehouseId
                ↪ )
103             .ToList();
104     }
105 }
106
107 // IDeliveryClusterer.cs
108 public interface IDeliveryClusterer // Implemented by our
    ↪ GeoCluster class
109 {
110     List<DeliveryCluster> ClusterDeliveries(List<Delivery>
        ↪ deliveries, double distanceThresholdKm, int
        ↪ minClusterSize = 1);
111 }
112
113 // IDeliverySelector.cs
114 public interface IDeliverySelector
115 {
116     Delivery? SelectNextDelivery(List<Delivery> deliveries,
        ↪  Location currentLocation, double
        ↪ remainingWeight, double remainingVolume)
117 }
118
119 // ProximityDeliverySelector.cs
120 public class ProximityDeliverySelector(IDistanceCalculator
    ↪ distanceCalculator)
121 : IDeliverySelector
122 {
123
124     public Delivery? SelectNextDelivery(List<Delivery>
        ↪ deliveries, Location currentLocation, double
        ↪ remainingWeight, double remainingVolume)
125     {
126         return deliveries
127             .Where(d => d.Package.Weight <= remainingWeight
```

```
128                     && d.Package.Volume <= remainingVolume)
129                 .MinBy(d => distanceCalculator
130                     .CalculateDistance(currentLocation, d.
                        ↪ Address.Location));
131     }
132 }
```

Extracting logic behind interfaces breaks our tests since dependencies aren't configured. We can fix this by updating `CustomAutoMocker` with appropriate setups. Since we thoroughly tested this logic already, we don't need extensive new tests for each extracted component, though the modular structure makes individual testing easier if desired.

In hindsight, identifying these separate responsibilities earlier would have simplified testing. But we often lack this understanding upfront. What matters is that we refactor when patterns emerge, guided by our comprehensive test suite.

As you practice TDD, you will become better at identifying how and when to split complex logic, e.g., following SRP, into testable components. A learning from this example could be: when tests become complex, consider whether the system under test should be decomposed into simpler parts.

We have now implemented the most complex part of our feature request. We did this by utilizing small step sizes, writing focused test, one at a time, and adding the logic that it encouraged. Triangulation helped discover patterns and refactor to the general implementation. We tackled the complexity by incrementally adding logic to the algorithm backed by tests. During the development we learned more about the final implementation and in the final refactoring we ensured that the implementation would be readable for other developers and extendable. E.g. by implementing a new `IDeliveryFilter` and `IDeliverySelector` we could start taking expected delivery time into account in the next version.

The ability to adjust step sizes, and dare to do so, is one of the most important skills when it comes to TDD. It is a skill that you develop with experience and by forcing yourself to do very small steps. I am certain that in our example above, we could have taken even smaller steps and thereby

have guided the implementation in even smaller increments, however, I too am still early in my TDD journey.

To better grasp the final design of the feature we just implemented using TDD, I have created an UML class diagram.

5.3 Key Takeaways: Breaking Down Complex Business Logic

The Challenge: Complex business logic like delivery assignment algorithms can be overwhelming to implement all at once using TDD.

The Solution: Use baby steps and triangulation to incrementally build complex logic, starting with the simplest cases.

Quick Reference:

- Start with the simplest case: Empty list, single item, then multiple items
- Use triangulation: Write 3+ tests to identify patterns before refactoring to the general solution
- Adjust step size: Take smaller steps when uncertain, larger steps when confident
- Build incrementally: Each test should encourage just a small addition to the implementation
- Refactor after patterns emerge: Don't generalize too early. Wait until you have multiple examples to guide you

Next Up: This challenge marks the end of our `DeliveryAssignmentService` example. The next and final challenge will talk more generally about the maintenance of test code.

Figure 5.2: UML class diagram of the Delivery Assignment feature developed using TDD in this booklet

Challenge 6

Long-Term Test Maintenance

The final challenge in TDD adoption is test code maintainability. Like production code, test code requires ongoing maintenance.

Test code needs updates when:

- Interfaces change during refactoring
- Domain objects evolve
- Business logic changes
- Bug fixes require new assertions
- External dependencies change
- Test data becomes outdated

And probably in many other cases too. Test maintenance is inevitable as systems evolve, so the question becomes, how do we minimize this maintenance burden?

6.1 Solution

6.1.1 DRY

While DRY (Don't Repeat Yourself) is fundamental in production code, apply it selectively in tests.

If you find yourself writing the same setup code, extract it into helper methods, base classes, or test fixtures so that it can be shared and is not repeated.

If you find yourself writing creation code for domain entities in many places and it can't be extracted into a test fixture as test cases need it to be created differently, consider creating reusable builders or factories for that domain entity.

Consolidating common code means updates happen in one place and new tests become easier to write. However, overly DRY tests can be harder to understand in isolation since setup details live elsewhere. Balance is key.

6.1.2 Intelligent Fixtures

Create adaptive test fixtures using tools like `AutoFixture` and automockers (e.g., `Moq.AutoMocker`).

Configure `AutoFixture` in a base test fixture to handle object creation across your system. Set up your auto-mocker with happy-path defaults, so you only arrange mocks when testing failure scenarios.

`AutoFixture` adapts to property changes automatically, while auto-mockers handle constructor signature changes gracefully. You'll still need to update fixtures occasionally for new requirements, but the overall maintenance burden decreases significantly.

One important note is, try to avoid having to write new fixtures from scratch for every test class. If you do this, you might as well not use fixtures and have every setup code, utilities etc. in your test class.

6.1.3 Limited Test Scope

Following unit testing fundamentals reduces maintenance:

- Mock external dependencies
- Avoid testing implementation, i.e. prefer black box over white box testing unless necessary
- Test only the public interface, e.g., public methods not private methods

Single-responsibility tests are easier to maintain. Recall how we extracted logic from `AssignDeliveriesToCourier` into separate classes in Challenge 5? Each component became independently testable. Testing `IDeliverySelector.SelectNextDelivery` in isolation is simpler than testing it through `AssignDeliveriesToCourier`. Limited test scope applies to both test design and system design. Smaller, focused components lead to maintainable tests. Lastly, make sure that your tests only assert on one thing and that it is clearly articulated. This will not only help clarify your tests' expectations but also minimize the maintenance efforts necessary when, e.g., one of the mocks you assert on changes.

6.1.4 The Right Level of Testing

I emphasize that not everything needs unit testing. While some industries require extreme confidence through comprehensive testing, most projects benefit from a more strategic approach. In TDD, unit tests are usually what drives development, but skip them for trivial, non-critical code you've implemented many times before. One common complaint about TDD is that it slows you down, and I agree it does take time. Hopefully that extra time and thoroughness also means that you win a lot of time in the long run. If you are frustrated that everything seems to progress much slower after adopting TDD, then follow these rules when it comes to what to unit test.

Don't unit test:

- Trivial code you've written many times

- Non-critical features with minimal failure impact
- Obvious implementations

Do test:

- Domain business logic
- Core application logic (use case flows)
- Object mappers
- Repository/ORM behavior

Test granularity should match your confidence and the code's criticality. You don't always need baby steps. Sometimes the happy path plus obvious edge cases cover 90

Supplement unit tests with integration and E2E tests. These cover gaps in unit testing and can also drive feature development alongside unit tests.

6.1.5 Testable Design

Perhaps the most impactful way to reduce test maintenance is designing for testability:

- Use dependency injection for easy mocking
- Separate concerns for focused, single-purpose classes
- Create clean interfaces with clear contracts
- Avoid static methods and global state

Code that is easy to test also makes for tests that are easy to maintain. If a test was easy to write in the first place, updating it will also be easier. With logic separated into focused classes, the impact of changes are isolated to smaller and more focused test cases.

Finally, we need a paradigm shift. Treat test code as production code. Test maintenance, utilities, and configurations deserve the same priority as features. Quality tests enable rapid, confident delivery. They're not

overhead, they're infrastructure. In addition to tests, we also need strong safety rails to quickly detect and report bugs or insufficiencies, and setups that allow us to know exactly what is happening inside our application when it runs. This is where observability and telemetry come into play, but that is a topic for another time.

6.2 Key Takeaways: Long-Term Test Maintenance

The Challenge: Test code requires ongoing maintenance as the system evolves, creating a hidden cost that can slow down development.

The Solution: Apply software engineering principles to test code and create sustainable testing practices.

Quick Reference:

- Apply DRY selectively: Share setup code but keep tests readable in isolation
- Use intelligent fixtures: `AutoFixture` and auto-mockers can adapt to code changes automatically
- Limit test scope: Mock dependencies, test public interfaces only, focus on single responsibilities
- Right level of testing: Not everything needs unit tests. Be strategic about what to test and use other levels of testing
- Testable design: Use dependency injection, separate concerns, and create clean interfaces
- Treat tests as production code: Maintain, refactor, and improve test code continuously

Next Up: The conclusion ties together the practical lessons from all six challenges and offers guidance on developing a sustainable personal approach to TDD.

Conclusion

Making TDD Work for You

When first adopting TDD, you'll encounter many challenges and pain points. We have gone over some of them in this booklet, but they're not exhaustive. Learning and becoming comfortable with TDD is a long process and it requires a fundamental shift in mindset, but if you keep at it, the benefits are worthwhile.

Tests will become one of your strongest tools in developing high quality and robust solutions that you can be confident in. TDD elevates automated tests, and makes them much more than just assertions on expected behavior, they become design tools, a playground for learning, and a tool for breaking down complexity into digestible, incremental pieces. TDD is more than just writing tests first, it's a different way of approaching development.

As we saw in this booklet, each challenge we faced when adopting TDD for developing the delivery assignment feature had practical solutions that come with experience. As we use TDD and overcome challenges, we will become better and better at tackling these challenges making them trivial.

The key practical takeaways from this booklet are:

- Start with ”outside-in” TDD to drive better design from a user's perspective
- Use tools like AutoFixture and auto-mocking to dramatically reduce test arrangement code and maintenance burden
- Apply triangulation and baby steps for complex logic in order to break down complexity and learn what patterns exist
- Break down complex classes into smaller, focused components following SRP to improve testability and reduce maintenance
- Focus testing efforts strategically. Not everything needs thorough unit testing. Prioritize critical, core and complex components
- Design for testability from the start with dependency injection, clear interfaces, and separation of concerns

The main reason I started looking into TDD was my dislike for writing unit tests. TDD offered a process that would elevate unit tests' purpose and value. This made them go from being annoying end-of-feature todos to valuable tools for developing quality software. However, adopting TDD came with a lot of challenges, some of which this booklet examines.

The learning that made the biggest difference for my adoption of TDD, was accepting that not every class or method needed to be tested 100% for it to be TDD. The acceptance of testing some logic thoroughly, and other less so, or not even using unit tests, made it much easier to adopt. With this, TDD took less time, while still allowing me to test core business rules or complex code where my confidence level was low. Furthermore, core business logic are often times more straight forward to test as the expectations are clearer and the code pure, making the testing experience better.

Going forward, I will continue using TDD as a practical tool in my development process for tackling complexity. By breaking down challenging problems into testable increments, I've found that TDD not only strengthens the reliability of my code but also enhances my understanding of the problems I'm solving. What I appreciate most about my evolved

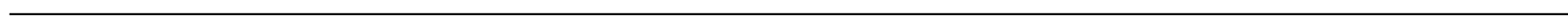
relationship with TDD is its flexibility. I now confidently apply it where it adds the most value, for critical business logic, complex algorithms, and code with high maintenance needs. For straightforward implementations or rapid prototypes, I've learned that forcing TDD can be counterproductive. This balanced approach makes TDD sustainable for me in my day-to-day work.

My final recommendations to developers who either want to start experimenting with TDD or who have already started and faced some of its challenges, are to start small and build confidence. Start by driving only the implementation of core business logic using TDD. This will both bring a lot of benefits in that the most important logic is well tested, and it is often easier to test this type of code due to its clear expectations which will help build confidence. Then when you feel confident enough, try implementing TDD in other parts of the codebase and experiment with different step sizes focusing on how to take very small steps.

At the end of the day, I think you should develop and refine your own approach to TDD that fits your context and workflow. This will ultimately be the thing that makes TDD a tool you'll keep using.

If you have any other challenges that were not covered in this booklet and like to share, please feel free to reach out.

Cheers!



References

AutoFixture (2025). *Minimizes the 'Arrange' phase of unit tests*. URL: https://github.com/AutoFixture/AutoFixture (visited on 01/26/2025).

Beck, Kent (2002). *Test-Driven Development: By Example*. Addison-Wesley Educational Publishers Inc.

Ester, M. et al. (1996). "A density-based algorithm for discovering clusters in large spatial databases with noise". In: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, pp. 226–231.

Evans, Eric (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Educational Publishers Inc.

Fowler, Martin (2003). *Anemic Domain Model*. URL: https://martinfowler.com/bliki/AnemicDomainModel.html (visited on 02/28/2026).

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

Moq (2025). *The most popular mocking framework for .NET*. URL: https://github.com/devlooped/moq (visited on 01/26/2025).

Moq.AutoMock (2025). *Auto-mocking container for Moq*. URL: https://github.com/moq/Moq.AutoMocker (visited on 01/26/2025).

North, Dan (2006). *Introducing BDD*. URL: https://dannorth.net/blog/introducing-bdd/ (visited on 02/28/2026).

Osherove, Roy (2005). *Naming Standards for Unit Tests*. URL: https://osherove.com/blog/2005/4/3/naming-standards-for-unit-tests.html (visited on 02/28/2026).

Palermo, Jeffrey (2008). *The Onion Architecture: Part 1*. URL: https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/ (visited on 01/26/2025).

Why Test-Driven Development (TDD) (2025). Figure 1 source. Marsner Technologies. URL: https://marsner.com/blog/why-test-driven-development-tdd/ (visited on 01/26/2025).

xUnit.net (2025). *Testing framework for .NET*. URL: https://xunit.net (visited on 01/26/2025).

JACOB GADE | Software Engineer at STRONGMINDS a Trifork Company