WRITING MAINTAINABLE UNIT TESTS

MASTERING THE ART OF LOOSELY COUPLED UNIT TESTS

JAN VAN RYSWYCK

Writing Maintainable Unit Tests

Mastering the art of loosely coupled unit tests

Jan Van Ryswyck

This book is for sale at http://leanpub.com/writing-maintainable-unit-tests

This version was published on 2023-10-25

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

To my parents, Antoon and Etiennette for learning me the value of hard work. And to my wife Frouke and my three kids Len, Lisa and Laura for all their support.

Contents

Preface

Who Should Read This Book?

This book is for experienced software developers who want to improve upon their existing skills in writing unit tests. In order to get the most value out of this book, it's recommended that you're already familiar with at least one xUnit test framework for writing automated tests as well as the mechanics of Test-Driven Development.

This book is not aimed at beginners who are completely new to writing unit tests or Test-Driven Development. No content has been included to get you up-and-running with an xUnit test framework. That would be an entire book of it's own. If you're new to these topics, then there are several other learning resources that will provide you with the proper foundation.

This book will teach you how to build loosely coupled, highly maintainable and robust unit tests that are trustworthy and improve the overall code quality of your software applications. Although the examples in this book are written in C#, the principles and guidance are broadly applicable to other platforms and programming environments as well (Java, Python, JavaScript, … etc.). You will be able to universally apply this knowledge throughout the rest of your career.

How This Book Is Organised

This book contains six chapters.

- **Chapter 1** provides an overview of the different kinds of automated tests and how to apply a healthy mix in a code base. We also touch on the different flavours of Test-Driven Development.
- **Chapter 2** describes the characteristics and principles that make tests maintainable. We touch on a number of design principles like DRY, DAMP and the Single-Responsibility Principle.

- **Chapter 3** discusses the anatomy of automated tests and how a good structure is essential to keep them readable for our fellow software developers.
- **Chapter 4** demonstrates a number of patterns and techniques to keep tests decoupled from the production code. This is the longest chapter of the book.
- **Chapter 5** shows a number of patterns and techniques for writing clear assertions and observations.
- **Chapter 6** touches on some miscellaneous principles that are useful for writing maintainable and readable tests.

Standing On The Shoulders Of Giants

The content of this book is based on 15+ years of experience with Test-Driven Development. However, I would never been able to learn anything if it weren't for the many excellent books and articles that shaped my thinking over all those years, some of which are explicitly mentioned in the bibliography section.

Then there's the open-source tools and libraries for writing all kinds of automated tests which we take for granted. These days every software development ecosystem has several options available that we as developers can make use of in order to build quality software. I'm humbled by the fact that so many developers spend so much of their free time to deliver us these amazing testing tools and libraries.

Last but not least, there's also the countless interactions with colleagues at work, attendees at code retreats, open space and regular conferences from which I have learned so much.

A big thank you to all these people. I highly value all the things that I've learned from you throughout the years. My hope is that by writing this book, I'm able to give some of this knowledge back.

Source Code

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Chapter 1: Types of Automated Tests

Introduction

In the first chapter, we discuss the different types of tests that you might encounter out there in the field so that you're able to recognise and categorise when you encounter them. But first, let's discuss briefly why we need tests.

But Why?

You've probably already heard quite a number of reasons why a software developer should write all kinds of automated tests. Some of these reasons might include that tests:

- Increase the quality of the code base.
- Increase the maintainability of the code base.
- Drive the design of the software system.
- Are good documentation.

There might be plenty of other reasons to be found on the internet as well. But the real reason we write tests is "shipping value faster". This sounds rather counterintuitive, right? How can writing more code makes us deliver value any faster? And still, we claim that we can go faster with automated tests.

Let's start by telling a short story.

Imagine that you don't write automated tests. Suppose you're a C++ developer somewhere in the mid 1990's. Extreme Programming (XP) practices weren't known to the world of software development yet. You've been tasked to add a new feature

to a software system. So, you figure out what this feature should do by asking a bunch of questions to a business person or maybe an analyst who summarised the requirements for you. After that, you do a big upfront design of how this new feature will fit into the existing system. Remember we're in the mid-90's, so we're following a Waterfall[1] approach here.

Now you're good to go. You're ready to start coding. And churning out code is what you do. You're writing hundreds and hundreds of lines of code. You're making progress and you feel productive. You're on top of the world! And after just a few days, you're done. But not really done.

You still have a few compiler warnings here and there. Since you're an honourable software developer, you try to fix those. It seems that in order to fix one of these compiler warnings, you need to make some changes that take a bit longer than just a couple of minutes. So you start to make some changes to the design in order to make the compiler happy. And after a few hours, you're done. But not really done.

The code compiles, but does it work? So you try to run the application and something strange happens. It crashes during startup. But you haven't touched any code that is involved with the startup of the process. How can this be? So you set out some breakpoints and start debugging furiously. After some time debugging the code you find the culprit. Turns out that at some point you did make a minor change to the startup code. And you even marked it with a TODO comment to yourself, stating that you need to have a second look at this code. You're a good citizen, so you start your journey to make things right. After a couple of hours of more coding and debugging, you're finally able to start the application. You're finally done. But not really done.

Now you still need to walkthrough the feature that you've just added in order to see if it works correctly. You exercise this new part of the application, finding a few more subtle bugs here and there. In the meanwhile you're boss stops by your desk, asking whether this new feature will be ready on time for this trade show scheduled for next week. You tell her that you're almost done. Just a couple of minor bugs to fix. And you continue you're journey, debugging and fixing the code. And after one more day of immensely displaying your reputation of *bug slayer* to the world, you're finally done. But not really done.

You only have two more days left before a new build needs to be made for the trade show. The new feature still needs to be approved by the QA department. But this is

[1] https://en.wikipedia.org/wiki/Waterfall_model

just a formality, right? You hand everything over to a QA engineer and mention the urgency of only two days left before the final build. The QA engineer starts testing furiously, and you already start thinking about the next feature that needs to be added. After about half a day of testing the application, you receive a full list of bugs from the QA engineer. What happened? It seems that by adding the new feature, you also broke a couple of existing features. So you fire up the good old debugger. One more day to go. You start fixing bugs gloriously, working late into the night. You hand everything over to the QA engineer again the next morning. You're exhausted? But this is how software development is supposed to be. And, most importantly, you're done now right? But not really done.

After another half day of testing by the QA engineer, it seems that you've introduced a couple of minor new bugs while fixing the ones on the first list. So you get handed over a new list of bugs. Now you're really pressed for time. Your boss is getting anxious as well, stopping by your desk every hour or so. You're completely fed up with this and start taking shortcuts in order to fix those latest bugs, spending a couple of hours on debugging the code. Now you're done!

The QA engineer quickly checks whether things got fixed, and you did. Now you're done. And close to collapse as well. But this is how things should go, right?

The common theme of this story is cycle time. Cycle time in this context means the time you spend between performing an action and receiving feedback about the consequences of that action. So looking back to our fictitious story, the cycle time is just way too long. Days pass by without any feedback. Does the code that's being written actually work? Did we break any existing features? Can we refactor a piece of code that needs improvement?

When we have automated tests, we have the potential of a short cycle time. We can start a new feature by writing a small test. We run the test, so that we can see it fail. We write no more production code than is sufficient to make the test pass. Then we refactor the code. We run all the tests to see if we didn't break anything. Rinse and repeat.

After only a few more of these short cycles, we push the code to the source control system. Only a few seconds later an automated build is started to compose the necessary artifacts of the entire system. This process also executes the entire suite of tests. After only a few minutes, the developer gets notified of the first results. And after a short time, everyone of the team knows whether we still have a working

software system or not. Trade shows or no trade shows, we always have the system ready for deployment. This way we can continuously provide value. Automated tests are the corner stone of the process to make this happen.

Let's dive into what types of automated tests exist.

A Taxonomy of Tests

It's more than fair to say that the terminology used in the world of automated tests can be a bit overwhelming. Software people have uncovered all sorts of tests in a wide variety of flavours.

For example, there are:

- Unit tests
- Integration tests
- API tests
- Database tests
- Acceptance tests
- UI tests
- Performance tests
- Regression tests
- And much more …

You might have heard about some of these kinds of tests. All of these do have their usefulness for the specific purpose that they are serving. Nonetheless, for modern day software developers it's sometimes quite hard to understand when a particular test falls into one or even multiple of these categories.

What's even more intimidating is how you determine which of these kinds of tests are applicable to your work and the specific use cases that apply for you. Personally, I don't feel that having a gazillion types of tests is very useful.

So having a more useful system for categorising tests seems to be in order. There has been a time where I've found the distinction between "fast" and "slow" tests to be useful. But coming up with a decent definition for both of these categories still remained to be somewhat non-deterministic.

Solitary and Sociable Tests

At some point I've come to adopt the terminology used in the excellent book Working Effectively with Unit Tests[2], written by Jay Fields. Here the author proposes two broad categories, **solitary** and **sociable** tests. These can be seen as the equivalent of tests that either run very fast, and tests that have a wider variety of slowness.

Solitary Tests

Solitary tests only have two constraints:

1. The code being exercised by these tests never cross the process boundary in which they are executed.
2. A single class or module is being tested. This is also called the Subject Under Test or *SUT* for short. Using this term in the code of solitary tests is a generally accepted practice.

The first constraint means that a solitary test never executes code that talks to a database, communicates across the network, touches the file system, etc. … This basically implies that a solitary test never requires any configuration whatsoever in order for it to execute correctly.

The second constraint is definitely the most controversial one. The most fundamental solitary test exercises a single class or module. Suppose that we have a couple of classes named A, B, C, D, E and F. In this particular case, class A uses B and C, and class B uses D and E, and so on.

[2] https://leanpub.com/wewut

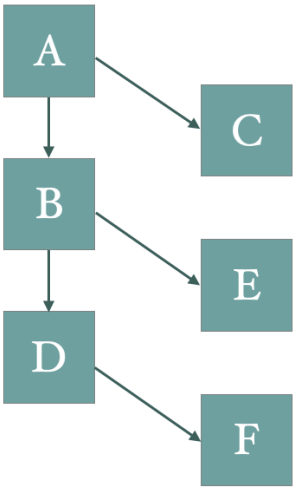

Figure 1.1 - A class hierarchy

So in this case, a couple of solitary tests verify the behaviour of class A in isolation and another couple of solitary tests verify the behaviour of class B in isolation, and so on.

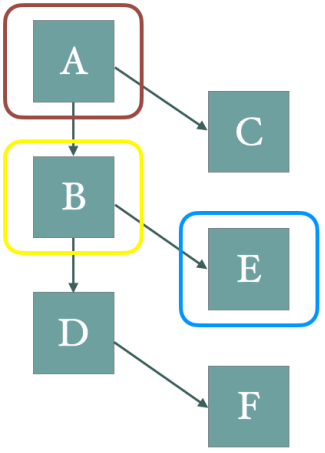

Figure 1.2 - Testing classes in isolation

Solitary tests focus on the individual parts, assuming that their collaborators work as expected. These are the cheapest tests that will cover the most ground. Solitary tests run very fast due to their nature of being very fine-grained. Therefore they are also called “unit tests”.

Sociable Tests

A sociable test is a test that cannot be classified as a solitary test. Tests that fall into this category are more course-grained as they usually exercise multiple classes or modules at the same time. They are more focused on the collaboration and integration of the different parts that make up a software system.

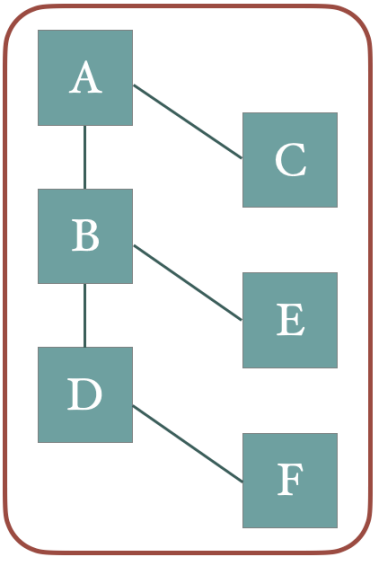

Figure 1.3 - Testing the integration of multiple classes

So in this case, a sociable test verifies the interactions between classes A, B, C, D, E and F, basically exercising their behaviour as a whole. The outcome of a sociable test depends on many different behaviours at once.

Sociable tests therefore execute more slowly than solitary tests, this due to the fact that they typically exercise more code and often cross their process boundary to communicate with other parts of the system like a database, a queue, the file system, the system clock, etc. ...

Sociable tests can be easily identified by their need for configuration. This is usually by means of configuration files where all sorts of connection strings or file locations are being stored that can be used by both the part of the application under test as well as the sociable tests themselves for verification of the outcome.

In order to build maintainable and high-quality software systems, we need both solitary as well as sociable tests, but not in an equal amount. Although tests from both of these categories have their strong points and weaknesses, it's better to have several solitary tests combined with only a few accompanying sociable tests.

Let's have a look at the test pyramid and discuss why this is useful.

The Test Pyramid

As we've mentioned in the previous section, it's better to have several solitary tests and only a few sociable tests that accompany them. This is where the test pyramid comes in. Initially described by Mike Cohn in the book Succeeding with Agile[3], the test pyramid provides a visual representation of a beneficial balance of automated tests.

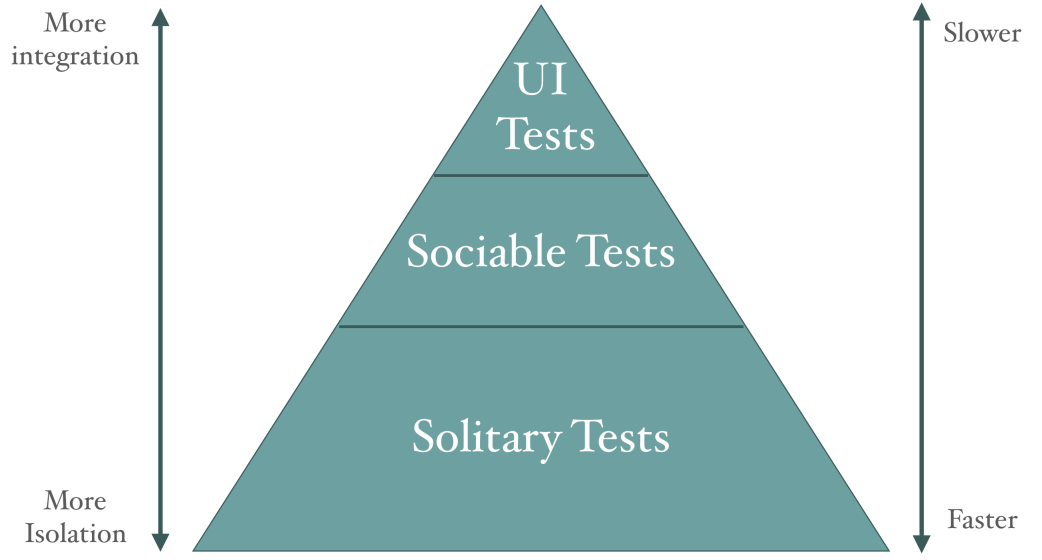

Figure 1.4 - The test pyramid

The concept of the test pyramid is made up from the following parts:

- At the base, we have solitary tests. These should make up the largest amount of automated tests in the software system.
- In the middle, we have sociable tests. These should definitely be part of the suite of automated tests, but in a significantly lesser amount than the number of sociable tests.
- At the top, we have a handful to a dozen broad system tests. This is usually in the form of UI or API tests that exercise the most important parts of the system.

[3] https://bit.ly/succeeding-with-agile2

> Try to resist the urge to create lots of these kind of tests as they can quickly turn against you and soon become a maintenance nightmare.

At the bottom of the test pyramid, we have the most isolation and also the fastest performance. This is where we get the most feedback about the design of the system. The more we move up the test pyramid, the more integration is employed and verified which results in slower tests and less feedback.

I personally consider the test pyramid to be more of a spectrum and less as a pile of discrete buckets. The moment a test exercises code of more than a single concrete class, it moves up the pyramid towards the area where the sociable tests live. How much the test rises depends on a couple of factors:

- Does a cluster of classes all live within the same dependency inversion boundary?
- Is there a single class within this cluster that takes up the role as the main entry point?
- Are the other classes in the cluster only used by the main entry class?
- Are all these classes part of a single conceptual whole?

If the answer to these questions are all positive, then I would argue that a test which exercises code of such a cluster of classes is still a solitary test and not a sociable test. However I do recognize that this hugely depends on the person you ask the question. There just isn't a wide consensus on this particular topic.

Never place all your bets on just a single category of automated tests! Applying all of these categories to the system is the best way to achieve a well-tested code-base. However, a surprisingly large number of development teams have made the mistake of applying a testing pyramid that is upside down. Such an "ice cream cone" shaped automated testing strategy implies that there are more sociable tests than solitary tests. This anti-pattern usually stems from an overreliance on manual testing and/or a lack of applying the Test-Driven Development process.

There are four primary reasons as to why we prefer that the majority of tests are solitary tests:

1. Sociable tests can be very slow and nondeterministic. This is due to the fact that they usually cross their process boundary. They make failure diagnostics more difficult because they are further removed from the cause of the failure.

2. Sociable tests can therefore be overly verbose and can require a lot of code in order to setup parts of the system under test. It requires more effort and therefore takes more time to write.
3. Solitary tests apply a certain pressure on the design of the system. They often indicate the design flaws that might exist. Sociable tests on the other hand don't provide such useful feedback about the design of the system because they are inherently farther removed from the details.
4. Sociable tests are highly susceptible to cascading failures. Let's take a moment to explain what this means.

Cascading Failures

As soon as we move away from testing a single concrete class or Subject Under Test and start considering collaborations between several concrete implementations, we're bound to encounter **cascading failures**. This means that a slightest change of the production code or a bug can result in a high number of failing tests that, from a conceptual point of view, don't have a direct relation to the changed code.

Let's have a look at an example.

```
public class Customer
{
    public int Id { get; }
    public string Email { get; private set; }
    public CustomerType Type { get; private set; }

    public Customer(int id, string email)
    {
        Id = id;
    }

    public void MakePreferred()
    {
        Type = CustomerType.Preferred;
    }
```

```
    public void ChangeEmail(string newEmail)
    {
        Email = newEmail;
    }
}

public enum CustomerType
{
    Regular = 0,
    Preferred = 1
}
```

Here we have a part of an application that manages customers. Users of the system can make a customer a preferred customer or change its email. A preferred customer receives some additional discounts and faster shipping. This is the implementation of the customer entity.

We also have two handler classes that receive commands for either making a customer preferred or changing its email.

```
//
// Make a customer preferred
//
public class MakeCustomerPreferredHandler
{
    private readonly AuthorizationService _authorizationService;
    private readonly ICustomerRepository _customerRepository;

    public MakeCustomerPreferredHandler(
        AuthorizationService authorizationService,
        ICustomerRepository customerRepository)
    {
        _authorizationService = authorizationService;
        _customerRepository = customerRepository;
    }
```

```
    public void Handle(MakeCustomerPreferred command)
    {
        if(! _authorizationService.IsAllowed(command.Action))
            ThrowUnauthorizedException(command.CustomerId);

        var customer = _customerRepository
            .Get(command.CustomerId);

        if(null == customer)
            ThrowUnknownCustomerException(command.CustomerId);

        customer.MakePreferred();
        _customerRepository.Save(customer);
    }

    private static void ThrowUnauthorizedException(int customerId)
    {
        var errorMessage = "Not authorized to make customer " +
            $"(ID: {customerId}) a preferred customer.";
        throw new UnauthorizedException(errorMessage);
    }

    private static void ThrowUnknownCustomerException(
        int customerId)
    {
        var errorMessage = $"The customer with ID {customerId} " +
            "is not known by the system and therefore could " +
            "not be made a preferred customer.";
        throw new UnknownCustomerException(errorMessage);
    }
}

public class MakeCustomerPreferred
{
    public CustomerAction Action { get; }
    public int CustomerId { get; }
```

```
    public MakeCustomerPreferred(int customerId)
    {
        Action = CustomerAction.MakePreferred;
        CustomerId = customerId;
    }
}

//
// Change the email of a customer
//
public class ChangeCustomerEmailHandler
{
    private readonly AuthorizationService _authorizationService;
    private readonly ICustomerRepository _customerRepository;

    public ChangeCustomerEmailHandler(
        AuthorizationService authorizationService,
        ICustomerRepository customerRepository)
    {
        _authorizationService = authorizationService;
        _customerRepository = customerRepository;
    }

    public void Handle(ChangeCustomerEmail command)
    {
        if(! _authorizationService.IsAllowed(command.Action))
            ThrowUnauthorizedException(command.CustomerId);

        var customer = _customerRepository
            .Get(command.CustomerId);

        if(null == customer)
            ThrowUnknownCustomerException(command.CustomerId);

        customer.ChangeEmail(command.NewEmail);
```

```
        _customerRepository.Save(customer);
    }

    private static void ThrowUnknownCustomerException(
        int customerId)
    {
        var errorMessage = $"The customer with ID {customerId} " +
            "is not known by the system and therefore it's " +
            "email could not be changed.";
        throw new UnknownCustomerException(errorMessage);
    }

    private static void ThrowUnauthorizedException(int customerId)
    {
        var errorMessage = "Not authorized to make customer " +
            $"(ID: {customerId}) a preferred customer.";
        throw new UnauthorizedException(errorMessage);
    }
}

public class ChangeCustomerEmail
{
    public CustomerAction Action { get; }
    public int CustomerId { get; }
    public string NewEmail { get; }

    public ChangeCustomerEmail(int customerId, string newEmail)
    {
        Action = CustomerAction.ChangeEmail;
        CustomerId = customerId;
        NewEmail = newEmail;
    }
}
```

Notice that these handler classes make use of an authorisation service that verifies whether the user is allowed to perform the operation.

We have two types of users in the system; help desk staff and back-office managers.

```
public class UserContext
{
    public UserRole Role { get; }

    public UserContext(UserRole role)
    {
        Role = role;
    }
}

public enum UserRole
{
    Unknown = 0,
    HelpDeskStaff = 1,
    BackOfficeManager = 2
}
```

Currently both types of users are allowed to make customers preferred or change their email. Only users that are not known by the system are disallowed all actions.

```
public class AuthorizationService
{
    private readonly UserContext _userContext;

    public AuthorizationService(UserContext userContext)
    {
        _userContext = userContext;
    }

    public bool IsAllowed(CustomerAction customerAction)
    {
        if(_userContext.Role == UserRole.Unknown)
            return false;
```

```
        // ...

        return true;
    }
}

public enum CustomerAction
{
    ChangeEmail = 0,
    MakePreferred = 1,
    // ...
}
```

Now the business has come up with a new requirement. From now on, only back-office managers are allowed to make customers preferred. Help desk staff are still allowed to change the email of our customers.

At this point, we make a change to the authorisation service to reflect this new requirement.

```
public bool IsAllowed(CustomerAction customerAction)
{
    if(_userContext.Role == UserRole.Unknown)
        return false;

    if(_userContext.Role == UserRole.HelpDeskStaff &&
       customerAction == CustomerAction.MakePreferred) {

        return false;
    }

    // ...

    return true;
}
```

What we see now is that suddenly some tests for the *MakeCustomerPreferredHandler*

start failing due to this change in the *AuthorizationService*. When we take a closer look, we see that the failing tests are using a concrete instance of the *AuthorizationService*.

```
[TestFixture]
public class When_an_authorized_user_makes_a_customer_preferred
{
    [Test]
    public void Then_the_specified_customer_should_be_made_a_
        preferred_customer()
    {
        var customer = new Customer(354, "john@doe.com");
        var command = new MakeCustomerPreferred(354);

        // Instantiate a concrete instance of the
        // AuthorizationService
        var userContext = new UserContext(UserRole.HelpDeskStaff);
        var authorizationService =
            new AuthorizationService(userContext);

        var customerRepository =
            Substitute.For<ICustomerRepository>();
        customerRepository.Get(Arg.Any<int>()).Returns(customer);

        var sut = new MakeCustomerPreferredHandler(
            authorizationService, customerRepository);
        sut.Handle(command);

        Assert.That(customer.Type,
            Is.EqualTo(CustomerType.Preferred));
    }

    [Test]
    public void Then_the_specified_customer_should_henceforth_be_
        treated_as_a_preferred_customer_by_the_system()
    {
```

```
        var customer = new Customer(354, "john@doe.com");
        var command = new MakeCustomerPreferred(354);

        // Instantiate a concrete instance of the
        // AuthorizationService
        var userContext = new UserContext(UserRole.HelpDeskStaff);
        var authorizationService =
            new AuthorizationService(userContext);

        var customerRepository =
            Substitute.For<ICustomerRepository>();
        customerRepository.Get(Arg.Any<int>()).Returns(customer);

        var sut = new MakeCustomerPreferredHandler(
            authorizationService, customerRepository);
        sut.Handle(command);

        customerRepository.Received().Save(customer);
    }
}

[TestFixture]
public class When_an_unauthorized_user_attempts_to_make_a_customer_
    preferred
{
    [Test]
    public void Then_an_exception_should_be_thrown()
    {
        var command = new MakeCustomerPreferred(354);

        // Instantiate a concrete instance of the
        // AuthorizationService
        var userContext = new UserContext(UserRole.Unknown);
        var authorizationService =
            new AuthorizationService(userContext);
```

```
        var customerRepository =
            Substitute.For<ICustomerRepository>();

        var sut = new MakeCustomerPreferredHandler(
            authorizationService, customerRepository);

        TestDelegate makeCustomerPreferred =
            () => sut.Handle(command);

        Assert.That(makeCustomerPreferred,
            Throws.InstanceOf<UnauthorizedException>());
    }
}

[TestFixture]
public class When_an_authorized_user_attempts_to_make_a_customer_
    preferred_that_is_not_known_by_the_system
{
    [Test]
    public void Then_an_exception_should_be_thrown()
    {
        var command = new MakeCustomerPreferred(354);

        // Instantiate a concrete instance of the
        // AuthorizationService
        var userContext =
            new UserContext(UserRole.BackOfficeManager);
        var authorizationService =
            new AuthorizationService(userContext);

        var customerRepository =
            Substitute.For<ICustomerRepository>();
        customerRepository.Get(Arg.Any<int>()).ReturnsNull();

        var sut = new MakeCustomerPreferredHandler(
            authorizationService, customerRepository);
```

```
        TestDelegate makeCustomerPreferred =
            () => sut.Handle(command);

        Assert.That(makeCustomerPreferred,
            Throws.InstanceOf<UnknownCustomerException>());
    }
}
```

This means that these tests are not testing the handler in isolation. They include, and are therefore dependent on, the implementation of the *AuthorizationService*. Therefore these are not 100% solitary tests.

Suppose that we have a few dozen handler classes like these with corresponding tests that each use a concrete instance of the *AuthorizationService*. Now this small change to the implementation of the *AuthorizationService* results in lots of failing tests. A simple change requested by the business can make these tests into a maintenance nightmare.

In this particular case we can easily fix these tests by changing the role that we pass to the *UserContext*. However, we can also take the path of further decoupling the tests.

We can introduce an interface for the *AuthorizationService*.

```
public interface IAuthorizationService
{
    bool IsAllowed(CustomerAction customerAction);
}

public class AuthorizationService : IAuthorizationService
{
    private readonly UserContext _userContext;

    public AuthorizationServiceV2(UserContext userContext)
    {
        _userContext = userContext;
```

```
    }

    public bool IsAllowed(CustomerAction customerAction)
    {
        if(_userContext.Role == UserRole.Unknown)
            return false;

        if(_userContext.Role == UserRole.HelpDeskStaff &&
           customerAction == CustomerAction.MakePreferred) {

            return false;
        }

        // ...

        return true;
    }
}
```

Now we are able to use a test double, either written manually or generated dynamically using a mocking framework like we already did for the *CustomerRepository*. In our example we've used the NSubstitute[4] library.

```
[TestFixture]
public class When_an_unauthorized_user_attempts_to_make_a_
    customer_preferred
{
    [Test]
    public void Then_an_exception_should_be_thrown()
    {
        var command = new MakeCustomerPreferred(354);

        // Instantiate a test double for the AuthorizationService
        // and disallow the action
```

[4] https://nsubstitute.github.io

```
        var authorizationService =
            Substitute.For<IAuthorizationService>();

        authorizationService
            .IsAllowed(CustomerAction.MakePreferred)
            .Returns(false);

        var customerRepository =
            Substitute.For<ICustomerRepository>();

        var sut = new MakeCustomerPreferredHandler(
            authorizationService, customerRepository);

        TestDelegate makeCustomerPreferred =
            () => sut.Handle(command);

        Assert.That(makeCustomerPreferred,
            Throws.InstanceOf<UnauthorizedException>());
    }
}
```

This way we have completely isolated the Subject Under Test and converted the tests for the handler classes to 100% solitary tests. This way we can make more changes to the *AuthorizationService* without breaking other tests (e.g. help desk staff is no longer allowed to change the email of a customer either).

Whenever a Subject Under Test requires collaborators for achieving its promised functionality,we basically have three options:

- Use an instance of a concrete class.
- Use an instance of a handwritten test double.
- Use an instance provided by a test double library or framework.

If we choose the first option, we're all set up for cascading failures. Cascading failures, as shown in this example, is one of the most important reasons why people lose faith in their tests. There are few things that can kill productivity and motivation faster

than cascading failures. The goal should always be to have a loosely coupled design for both the production code as well as the test code that accompanies it.

Therefore we should favour more fine-grained, solitary tests over course-grained, sociable tests. This way, changes to the Subject Under Test will only create failures in tests that are directly associated with it. Failures can be resolved more easily and quickly with fine-grained tests, sometimes even by just looking at the code.

Failures of course-grained tests more frequently involves debugging the code under test. Debugging code requires a lot more mental cycles from software developers, which results in more time spent trying to find the offending code. And time takes money.

We can reduce the effects of cascading failures by replacing cross-boundary collaborators with test doubles, converting all tests into solitary tests. But do make sure to take a pragmatic approach. I like to emphasise that it's still fine to involve the collaborators of a Subject Under Test when they live within the same boundary. An example of this is when writing solitary tests for an aggregate root[5] inside the domain of an application.

The term "Subject Under Test" is a reminder to think carefully about the granularity of the unit of code that we want to design and therefore consume.

Test Stages During Continuous Integration

As we've already learned by now - we need a healthy mix of both solitary and sociable tests in order to build high-quality systems. We need solitary tests for fast feedback, especially when following the principles of Test-Driven Development. And we need sociable tests for verifying the correctness of the interactions between the different parts of the system.

Because of the different characteristics of both solitary and socially tests, they are also applied differently during the stages the development lifecycle as well as the CI build pipeline. Such a staged process gives developers the confidence they need to efficiently add new features or make changes to the system, to feel productive and have fun.

[5] https://www.martinfowler.com/bliki/DDD_Aggregate.html

Therefore, during a CI build, the solitary tests are usually executed first because they are the fastest automated tests in the spectrum of the test pyramid. This way we can get feedback as quickly as possible. Solitary tests are executed most often compared to any other category of tests.

If all of the solitary tests pass, then the sociable tests are executed, after which the broad system tests exercise the system as a whole. Usually the execution of the sociable tests and broad system tests are separate steps in the CI build pipeline. I've also seen setups where both categories of tests are executed in separate builds of their own. This all depends heavily on the features and capabilities of the continuous integration software used by the team.

Test Duplication

We should be aware of the costs that our tests impose and therefore try to avoid test duplication throughout the layers of the test pyramid as much as possible. Our goal should always be to keep our tests as far down the test pyramid as possible. Therefore we shouldn't feel bad to remove sociable tests that exercise the same functionality that is already covered by solitary tests.

Suppose that we have a method on a web controller that represents a REST endpoint in our application. This controller method uses other parts of the system to fulfil its required functionality.

A good approach is to have a number of solitary tests that exercise the behaviour of the controller method. But these solitary tests are not able to verify whether the controller method can actually respond to HTTP requests.

Therefore we need a sociable test to perform this verification so that we're confident that all the individual parts of the system are working together correctly. This sociable test usually exercises the so-called "happy path", but nothing more. This way we have the right amount of tests at each level of the test pyramid without the potential of having overlapping failures.

From here on out we narrow our focus to developing maintainable and high-quality solitary tests. As we will no longer be discussing sociable tests, do bear in mind that some of the same principles and practices that apply to solitary tests can also help and guide you when writing sociable tests as well.

State and Behaviour Verification

There are generally two different styles that are being used for solitary tests:

- Solitary tests that perform state verification.
- Solitary tests that perform behaviour verification.

State Verification

Applying state verification means that we first exercise the Subject Under Test by calling one or more methods on an object. Then we use assertions to verify the state of the object. We may also verify any results returned by the method.

This is also known as the **Detroit School of TDD**. This nickname comes from its origins out of Extreme Programming, a well known development methodology used by Chrysler's C3 project at the end of the 1990s. Kent Beck's book Test-Driven Development By Example[6] best describes this approach.

Let's have a look at an example of state verification.

```
public class Resume
{
    private readonly IList<Experience> _experiences;
    public IEnumerable<Experience> Experiences => _experiences;

    public Resume()
    {
        _experiences = new List<Experience>();
    }

    public void AddExperience(string employer, string role,
        DateTime from, DateTime until)
    {
        var newExperience = new Experience(employer, role,
```

[6] https://bit.ly/tdd-by-example2

```
            from, until);
        _experiences.Add(newExperience);
    }
}

public class Experience
{
    public string Employer { get; }
    public string Role { get; }
    public DateTime From { get; }
    public DateTime Until { get; }

    public Experience(string employer, string role, DateTime from,
        DateTime until)
    {
        Employer = employer;
        Role = role;
        From = from;
        Until = until;
    }
}
```

Here we have a system for managing online résumés. A user can add one or more experiences to a résumé. Therefore we have a *Resume* class with a method *AddExperience*. This method adds an *Experience* object to a collection.

```
[TestFixture]
public class When_adding_experience_to_a_resume
{
    [Test]
    public void Then_the_specified_experience_should_now_appear_
        on_the_resume()
    {
        var experienceFrom = new DateTime(2014, 09, 12);
        var experienceUntil = new DateTime(2017, 12, 31);
```

```
        var resume = new Resume();
        resume.AddExperience("Google", "Data analyst",
            experienceFrom, experienceUntil);

        var addedExperience = resume.Experiences
            .SingleOrDefault();

        Assert.That(addedExperience,
            Is.Not.Null);
        Assert.That(addedExperience.Employer,
            Is.EqualTo("Google"));
        Assert.That(addedExperience.Role,
            Is.EqualTo("Data analyst"));
        Assert.That(addedExperience.From,
            Is.EqualTo(experienceFrom));
        Assert.That(addedExperience.Until,
            Is.EqualTo(experienceUntil));
    }
}
```

Looking at the implementation of the corresponding solitary test, we see that a *Resume* object is created and that the *AddExperience* method is being called. Then we verify whether a correct *Experience* object has been added to the collection of experiences. With this test we verify the new state of the *Resume* object.

State verification tests do not instrument the Subject Under Test to verify its interactions with other parts of the system. We only inspect the observable state of an object and the direct outputs of its methods. This approach tests the least possible implementation detail. It has the most notable advantage that these tests will continue to pass even if the internals of the SUT's methods are changed without altering their observable behaviour.

There are also two slightly different styles of state verification, namely **Procedural State Verification** and **Object State Verification**. We'll cover these two styles more in-depth in chapter 5 - "Assertions and Observations".

Behaviour Verification

Verifying the behaviour of the Subject Under Test implies the ability for instrumenting and verifying its interactions with other objects or other parts of the system. This is mostly done by using test doubles, either written manually or by using a framework.

This is also known as the **London School of TDD**. This nickname comes from the practices applied by the Extreme Programming community in London. The concepts of this process is most clearly described by the book Growing Object Oriented Software Guided By Tests[7], written by Steve Freeman and Nat Pryce, also known as the *GOOS* book.

Let's have a look at an example of behaviour verification.

```
public class RegisterUserHandler
{
    private readonly IUserRepository _userRepository;
    private readonly IEmailSender _emailSender;

    public RegisterUserHandler(
        IUserRepository userRepository,
        IEmailSender emailSender)
    {
        _userRepository = userRepository;
        _emailSender = emailSender;
    }

    public void Handle(RegisterUser command)
    {
        var user = new User(command.Email);
        _userRepository.Save(user);

        var emailMessage = new EmailMessage(user.Email,
            "Confirm email", "...");
        _emailSender.Send(emailMessage);
```

[7] https://bit.ly/tdd-goos2

```
    }
}

public class RegisterUser
{
    public string Email { get; }

    public RegisterUser(string email)
    {
        Email = email;
    }
}
```

Here we have a system that manages user registration. We have a class named *RegisterUserHandler* that creates and saves a new user. Also a confirmation email is sent to verify the existence of the specified email address.

```
[TestFixture]
public class When_registering_a_new_user
{
    [Test]
    public void Then_the_new_user_should_be_registered_in_the_
        system()
    {
        var userRepository = Substitute.For<IUserRepository>();
        var emailSender = Substitute.For<IEmailSender>();

        var sut = new RegisterUserHandler(userRepository,
            emailSender);

        var command = new RegisterUser("john@doe.com");
        sut.Handle(command);

        userRepository.Received().Save(Arg.Any<User>());
    }
```

```
    [Test]
    public void Then_a_confirmation_email_should_be_sent()
    {
        var userRepository = Substitute.For<IUserRepository>();
        var emailSender = Substitute.For<IEmailSender>();

        var sut = new RegisterUserHandler(userRepository,
            emailSender);

        var command = new RegisterUser("john@doe.com");
        sut.Handle(command);

        emailSender.Received().Send(Arg.Any<EmailMessage>());
    }
}
```

Looking at the implementation of the solitary tests, notice that we're using the NSubstitute[8] mocking library to verify that the collaborators of the *RegisterUserHandler* class are being called.

We pass those test doubles to the *RegisterUserHandler* class using constructor injection. For more information on Dependency Injection, you can check out the excellent book Dependency Injection[9] by Mark Seemann. With these tests we verify the behaviour of the *RegisterUserHandler* class.

Applying this approach is a bit more complicated and harder to reason about than using state verification. These tests are also more brittle as they imply that the test code has intimate knowledge about the implementation details of the Subject Under Test. Changing the implementation details without changing the overall functionality will more likely result in failing tests.

There are a few techniques and design principles that can somewhat reduce this brittleness. The most important one is to strive for the least amount of collaborators. This way interactions can be verified effectively without sacrificing maintainability. We'll cover this more in-depth in chapter 4 - "Decoupling Patterns".

[8] https://nsubstitute.github.io
[9] https://bit.ly/dep-inj2

From this explanation, we might come to the conclusion that only applying state verification is the best approach for writing tests. And to a certain degree this is true. We should favour state verification over behaviour verification most of the time. But in practice, we actually need both kinds of verifications. Not every object in our system has the same role or responsibility. Some are more algorithmic and involves logic based code, while others involve more interactions between objects.

As you might have guessed, state verification is most useful for verifying algorithms and business logic, like domain objects, validators, static functions like helper or extension methods, etc. … Behaviour verification is most useful for verifying interactions, like services, controllers, gateways, etc. …

Being aware about state and behaviour verification is the first step to learn about how to write maintainable solitary tests.

Test-Driven Development

Test-Driven Development is a discipline that exists for about two decades now. Unfortunately, to this very day, it is still not without controversy. Most professional developers know that writing some form of automated tests can be quite beneficial for any type of codebase. What still seems to be quite controversial is whether to write a test before or after the production code has been laid out. This discussion seems to stir its head every other day on many discussion forums and on social media.

When I use the term TDD, I mean the repeating process of first writing a failing test, then writing as little production code as possible to make the test pass, after which the important step of refactoring the code ensues. This process is also commonly referred to as "Red, Green, Refactor".

Some people firmly follow this mantra of "Red, Green, Refactor". Others don't like to follow this strict process for whatever reason and prefer to write tests after they've completed the implementation. Personally I like to write tests before the production code, and I highly encourage anyone to adopt this practice if they haven't already.

Nevertheless, I would like to touch on two different approaches to Test-Driven Development that are practiced throughout the world of software development, namely:

- Inside-Out TDD
- Outside-In TDD

Inside-Out TDD

As its names implies, this approach allows us to start with the smallest unit of code - exercising either an individual class or module - by following the "Red, Green, Refactor" cycle. During this process, the design of the implementation happens during the "Refactor" step. This step is quite important and becomes more involved compared to the "Red" and "Green" steps. Usually there's not much upfront design involved, which can be considered a positive thing as it constrains overengineering.

Each entity of the system is created in a TDD fashion using solitary tests until the whole feature is built up. Then a few sociable tests are added to verify that all the parts are working together as expected. Inside-Out TDD is basically the approach where you start at the bottom of the test pyramid and work your way up.

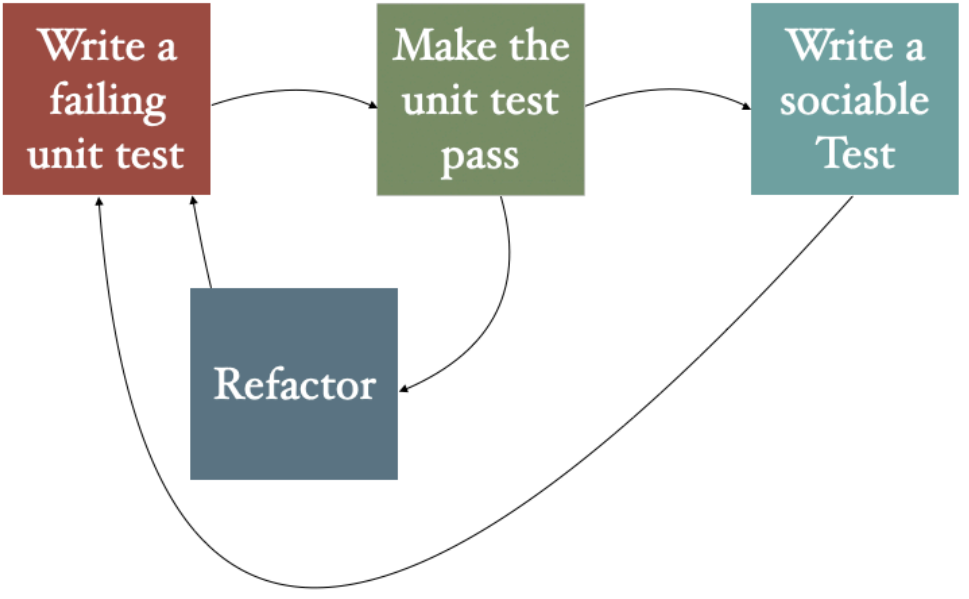

Figure 1.5 - The process of Inside-Out TDD

The most notable advantage by first focusing on the individual parts of the system is the ability to work in very small increments. This also enables that the development work can be parallelised within a software team.

The downside of Inside-Out TDD is that by initially focusing on the individual parts, there's a higher risk of these entities not working together correctly with the possibility of rework.

Outside In TDD

This approach focuses on creating a complete flow between the larger parts of the system right from the start. All the entities that make up a feature of the system are being created from the get-go, immediately verifying the wiring and interactions between them. The design of the system happens upfront during the "Red" step of "Red, Green, Refactor". This results in the "Refactor" step becoming much more shallow. The different parts that make up a feature of the system are put in place while writing a failing sociable test. Such a thin slice of real functionality is also referred to as a **walking skeleton** (see Growing Object Oriented Software Guided By Tests[10]). This failing sociable test serves as a beacon, making sure that no implementation is being forgotten.

[10] https://bit.ly/tdd-goos2

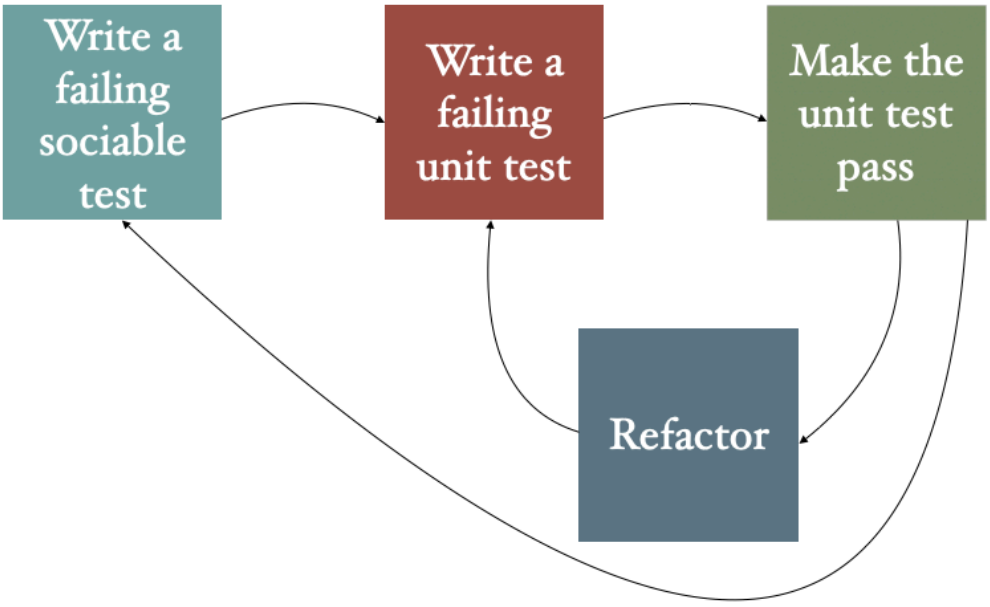

Figure 1.6 - The process of Outside-In TDD

The first test being written usually exercises a controller or a service at the system boundary as the starting point. A sociable test is usually employed here instead of a solitary test. Some parts of the code need to be swapped out by fake implementations that mimic the behaviour of the real-world implementations.

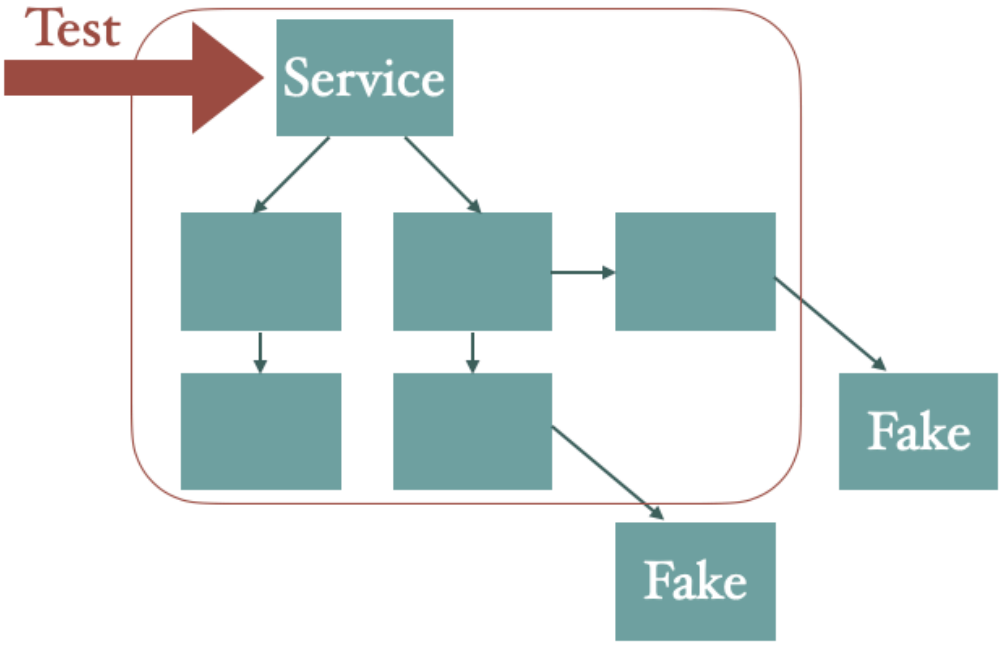

Figure 1.7 - A walking skeleton

When every piece of the puzzle is in place, solitary tests are then used to further flesh out the concrete implementations of these individual classes or modules.

Outside-In TDD is basically the approach where you start at the top of the test pyramid and work your way down.

The advantage of this approach is that it feels more exploratory, and is ideal for those kinds of situations where the high-level parts of the system are known without committing to the more fine-grained implementation details. This results in a "think like the client" mindset well before we start thinking as a software developer.

The downside of Outside-In TDD is that the design of the larger feature and its different parts should be known right from the start as opposed to driving the design of the smaller parts of the system. This usually takes significantly more time to write a first failing sociable test.

One might come to the conclusion that it's somehow important to choose one approach over the other. However, this is definitely not the case. Please note that

Inside-Out TDD and Outside-In TDD are not mutually exclusive. There's no point in choosing one approach over the other and rigorously sticking to a particular choice. We should practice and master both ways of writing automated tests in order to develop a "gut instinct" for applying a certain approach.

For more detailed information regarding Inside-Out TDD and Outside-In TDD and their implications, I would like to refer to the article Does TDD Really Lead To Good Design?[11] by Sandro Mancuso.

Summary

In this chapter we described the different kinds of automated tests that you might encounter in a code base. We discussed the pros and cons of both solitary and sociable tests. We've taken a close look at the test pyramid and why applying a healthy mix of different kinds of tests is very important.

We then narrowed our focus towards solitary tests. We talked about verifying state versus verifying behaviour. We then finished this chapter with an explanation of Inside Out TDD and Outside In TDD.

[11] https://codurance.com/2015/05/12/does-tdd-lead-to-good-design/

Chapter 2: Maintainable Solitary Tests

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Introduction

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Clean Solitary Tests

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

The DRY Principle

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

The Single-Responsibility Principle (SRP)

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

The DAMP Principle

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Other Characteristics Of Maintainable Solitary Tests

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Chapter 3: The Anatomy of Solitary Tests

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Introduction

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Arrange, Act, Assert

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Stage 1: Arrange

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Stage 2: Act

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Stage 3: Assert

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Examples

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

AAA Per Test Method

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Single Assert Per Test

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Avoid SetUp / TearDown

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

AAA Per Test Class

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Assert Last Principle

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Naming Unit Tests

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Roy Osherove naming style

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Behaviour specification naming style

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Which one?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Chapter 4: Decoupling Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Introduction

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Only Test Through Public Interfaces

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Object Mother

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Test Data Builder

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Test Data Builder Guidelines

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Explicit Test Data Values

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Default Test Data Values

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Apply Cautiously

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Touching Point With Production Code

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Keep It Simple

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Immutable Test Data Builder

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

State and Behaviour Verification (Again)

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

State Verification

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Behaviour Verification

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Indirect Inputs and Outputs

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Test Doubles

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Dummy

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Stub

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Spy

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Mock

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Fake

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Overview

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Manual Test Doubles Versus Test Double Frameworks

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Test Double Heuristics

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Avoid Excessive Specification of Test Doubles

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Avoid Using Test Doubles For Types That You Don't Own

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Avoid Test Doubles For Concrete Classes

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Don't Let Test Doubles Return Other Test Doubles

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Don't Implement Behaviour In Test Doubles

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Reduce The Number Of Collaborators

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Subject Under Test Builder

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Auto Mocking Container

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Fixture Object

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Chapter 5: Assertions and Observations

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Introduction

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Making Clear Observations

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Only Asserts Should Cause Failing Tests

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Single Assert Per Test

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Procedural Versus Object State Verification

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Procedural State Verification

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Object State Verification

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Equality

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Comparer

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Custom Assert

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Deep Equal

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Resembling Objects

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Chapter 6: Principles For Solitary Tests

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Introduction

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Avoid Inheritance For Test Classes

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

TDD Requires Design Skills

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Avoid The Self-Shunt Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Avoid Using The System Clock In Solitary Tests

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Prevent Domain Knowledge From Sneaking Into Solitary Tests

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Solitary Tests For Logging

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Closing Thoughts

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

About The Author

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.

Bibliography

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/writing-maintainable-unit-tests.